Міністерство освіти і науки України
Харківський національний університет радіоелектроніки

Факультет Комп'ютерних наук
(повна назва)
Кафедра Системотехніки
(повна назва)

КВАЛІФІКАЦІЙНА РОБОТА
Пояснювальна записка

рівень вищої освіти другий (магістерський)

Розробка мікро-сервісної архітектури систем пошуку спеціалізованої літератури
(тема)

Виконав:
студент 2 курсу групи СПРм-19-2
Говдерчак О.П.
(прізвище, ініціали)

Спеціальність 122 Комп'ютерні науки
(код і повна назва спеціальності)

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

Освітня програма Системне проектування
(повна назва спеціалізації)

Керівник професор Ситніков Д.Е.
(посада, прізвище, ініціали)

Допускається до захисту

Зав. кафедри системотехніки _____ професор Гребеннік І.В.
(підпис) (прізвище, ініціали)

2021 р.

Харківський національний університет радіоелектроніки

Факультет Комп'ютерних наук
Кафедра Системотехніки
Рівень вищої освіти другий(магістерський)
Спеціальність 122 Комп'ютерні науки
Тип програми освітньо-наукова
Освітня програма Системне проектування
(повна назва)

ЗАТВЕРДЖУЮ:
Зав. кафедри

_____
(підпис)

«____»_____ 20 ___ р.

ЗАВДАННЯ
НА АТЕСТАЦІЙНУ РОБОТУ

студентові Говдерчаку Олексію Павловичу
(прізвище, ім'я, по батькові)

1. Тема роботи Розробка мікро-сервісної архітектури систем пошуку спеціалізованої літератури
затверджена наказом по університету від «29» березня 2021р. № 389 Ст
2. Термін подання студентом роботи (проекту) 18 травня 2021р.
3. Вихідні дані до роботи (проекту) *Архітектура розроблюваної системи: мікросервісна; цільова аудиторія: підприємства із базою даних літературних джерел; інфраструктура системи: хмарна; критерії оцінювання архітектури системи: швидкодія на великих об'ємах даних, масштабованість, стійкість до помилок; функції системи: пошук / пропонування літературних джерел за декількома характеристиками. 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. Зміст пояснювальної записки (перелік питань, що потрібно розробити)
4.1 Вступ. 4.2 Аналіз предметної області. 4.2.1 Опис об'єкта дослідження. 4.2.2 Аналіз монолітної архітектури. 4.2.3 Аналіз мікросервісної архітектури. 4.2.4 Аналіз вимог до системи. 4.2.5 Дослідження інфраструктури системи пошуку. 4.2.6 Дослідження щодо роботи базою даних підприємства. 4.2.7 Дослідження оптимізації запитів до бази даних за допомогою кешування. 4.2.8 Постановка завдань будування архітектури системи. 4.3 Опис прийнятих проектних рішень. 4.3.1 Обґрунтування вибору мікросервісної архітектури. 4.3.2 Обґрунтування використання хмарних обчислень. 4.3.3 Обґрунтування вибору провайдеру хмарних обчислень. 4.3.4 Вибір необхідного стеку хмарних сервісів. 4.3.5 Використання AWS Step Functions. 4.4 Будування архітектури системи пошуку. 4.4.1 Приклад бази даних підприємства із спеціалізованою літературою. 4.4.2 Будування архітектури системи пошуку як окремого цілого сервісу. 4.4.3 Будування архітектури системи пошуку як ланки окремих сервісів. 4.4.4 Будування архітектури як автомату стану. 4.4.5 Порівняння підходів. 4.5 Висновки. 4.6 Перелік посилань.
5. Перелік графічного матеріалу (з точним зазначенням обов'язкових креслеників, плакатів) *5.1 Принципи монолітної та мікросервісної архітектури; 5.2 Діаграма послідовності системи*

пошуку спеціалізованої літератури; 5.3 Система підприємства монолітного зразка з системою пошуку всередині та коли система пошуку – це окремий мікросервіс; 5.4 Популярність використання хмарних технологій та прогнозування прибутку провайдерів хмарних обчислень; 5.5 Популярність Amazon серед інших постачальників хмарних послуг; 5.6 Порівняння безсерверних функцій Амазону та його віртуального серверу; 5.7 Фізична модель бази даних прикладу підприємства; 5.8 Необхідні таблиці для системи пошуку та створення прикладу реплікації їх та міграції в єдину DynamoDB таблицю; 5.9 Взаємодія системи підприємства (ХНУРЕ) із системою пошуку як окремим суцільним сервісом; 5.10 Взаємодія системи підприємства (ХНУРЕ) із системою пошуку як окремим суцільним сервісом; 5.11 Механізм розгортки суцільного монолітного додатку системи пошуку; 5.12 Діаграма послідовності роботи системи пошуку, розбитої на декілька сервісів (функцій); 5.13 Граф роботи системи пошуку, розробленої за допомогою автомату стану Step Functions та працюючий алгоритм системи; 5.14 Працюючі архітектури з фізичним сервером та ланцюгом безсерверних функцій.

6. Консультанти розділів роботи (проекту)

Найменування розділу	Консультант (посада, прізвище, ім'я, по батькові)	Позначка консультанта про виконання розділу	
		підпис	дата
Розділи спеціальної частини	проф. Ситніков Д.Е		

КАЛЕНДАРНИЙ ПЛАН

Пор. №	Назва етапів атестаційної роботи	Термін виконання етапів роботи	Примітка
1.	Отримання завдання атестаційної роботи	29.03.21	
2.	Аналіз завдання, літератури та аналогів з теми атестаційної роботи	10.04 — 17.04.21	
3.	Вибір засобів для розробки технічних вимог до програми	18.04 — 21.04.21	
4.	Структурне проектування	22.04 – 25.04.21	
5.	Вибір інфраструктури розгортки системи	26.04 – 01.05.21	
6.	Розробка варіантів архітектур	02.05 — 9.05.21	
7.	Оформлення пояснювальної записки	9.05 — 13.05.21	
8.	Перевірка на плагіат	14.05.21	
9.	Представлення на рецензування	15.05.21	
10.	Занесення роботи в електронний архів	15.05.21	
11.	Підготовка презентації та доповіді	16.05.21	
12.	Попередній захист атестаційної роботи	17.05.21	

Дата видачі завдання 29 березня 2021

Студент _____ Говдерчак О.П.

(підпис)

Керівник роботи (проекту) _____ професор Ситніков Д.Е

(підпис) (посада, прізвище, ініціали)

РЕФЕРАТ

Атестаційна робота: 86 стор., 23 рис., 5 табл., 3 додатки, 20 джерел. Графічний матеріал атестаційної роботи містить 14 плакатів.

Об’єктом дослідження виступає розробка архітектури системи пошуку спеціалізованої літератури.

Предметом розробки є побудова архітектури додатку за принципом мікросервісів для забезпечення можливості системи пошуку відтворити процес пропонування літературних джерел користувачу.

Метою роботи є поліпшення роботи пошуку електронних книжкових джерел підприємства, що має базу спеціалізованої літератури.

Результат роботи представлений у вигляді різних варіантів будування архітектури системи пошук-у, що може приймати за вхідні дані необмежену кількість літературних джерел та фільтрувати їх за характеристиками, наданими підприємством під час інтеграції або запиту до системи.

Областю застосування може виступати будь-яке підприємство, що має власний список співробітників, направленостей та літературних джерел. Прикладом може являтися університет, де студенти навчаються в групах різним дисциплінам, що належать до різних направлень та факультетів або компанія, що займається декількома проектами одразу, що ведуть різні команди та кожний проект має свої спеціалізовані документи. Правильно реалізована архітектура може полегшити, прискорити та гарантувати надійність процесу знаходження потрібної літератури.

МІКРОСЕРВІСИ, ПІДСИСТЕМА ПОШУКУ, СПЕЦІАЛІЗОВАНА ЛІТЕРАТУРА, МОНОЛІТ, ХМАРНІ ОБЧИСЛЕННЯ, БАЗА ДАНИХ, AMAZON

WEB SERVICES, AWS, LAMBDA, STEP FUNCTIONS, ІНФРАСТРУКТУРА, БЕЗСЕРВЕРНІ ФУНКЦІЇ, SERVERLESS

РЕФЕРАТ

Аттестационная работа: 86 стр., 23 рис., 5 табл., 3 приложения, 20 источника. Графический материал аттестационной работы содержит 14 плакатов.

Объектом исследования выступает разработка архитектуры системы поиска специализированной литературы.

Предметом разработки является строительство архитектуры приложения по принципу микросервисив для обеспечения способность системы поиска воспроизвести процесс предложение литературных источников пользователю.

Целью работы является улучшение работы поиска электронных книжных источников предприятия, имеющего базу специализированной литературы.

Результат работы представлен в виде различных вариантов построения архитектуры системы поиска, может принимать за входящие данные неограниченное количество литературных источников и фильтровать их по характеристикам, предоставленным предприятием при интеграции или запроса к системе.

Областью применения может выступать любое предприятие, имеющее собственный список сотрудников, направлений и литературных источников. Примером выступает университет, где студенты учатся в группах многообразно дисциплинам, принадлежащих к разным направлений и факультетов или компания, занимающаяся несколькими проектами сразу, и каждый проект имеет свои документы. Правильно реализована архитектура может облегчить, ускорить и гарантировать надежность процесса нахождения нужной литературы.

МИКРОСЕРВИСЫ, ПОДСИСТЕМА ПОИСКА, AWS, LAMBDA, СПЕЦИАЛИЗИРОВАННАЯ ЛИТЕРАТУРА, МОНОЛИТ, облачные вычисления,

БАЗА ДАННЫХ, AMAZON WEB SERVICES, STEP FUNCTIONS, ИНФРАСТРУКТУРА, БЕЗСЕРВЕРНИ ФУНКЦИИ, SERVERLESS

ABSTRACT

Certification work: 86 pages, 23 figures, 5 tables, 3 appendices, 20 sources. The graphic material of the attestation work contains 14 posters.

The object of research is the development of the architecture of the search system of specialized literature.

The subject of development is to build the architecture of the application on the principle of microservices to ensure the ability of the search engine to reproduce the process of offering literature to the user.

The aim of the work is to improve the search for electronic book sources of the enterprise, which has a base of specialized literature.

The result is presented in the form of various options for building a search engine architecture, which can take as input an unlimited number of literature sources and filter them by the characteristics provided by the company during the integration or query to the system.

The scope can be any company that has its own list of employees, areas and literature sources. An example would be a university where students study in groups in different disciplines belonging to different fields and faculties or a company that deals with several projects at once, leading different teams and each project has its own specialized documents. Properly implemented search engine architecture can greatly facilitate, speed up and guarantee the reliability of the process of finding the right literature.

MICROSERVICE, SEARCH SUBSYSTEM, SPECIALIZED LITERATURE, MONOLITH, CLOUD CALCULATION, DATABASE, AMAZON WEB SERVICES, AWS, LAMBRADE, STEPFUNCTION FUNCTION

ЗМІСТ

1 АНАЛІЗ ПРЕДМЕТНОЇ ОБЛАСТІ....13
1.1Опис об’єкта дослідження....13
1.2 Аналіз монолітної архітектури....14
1.3 Аналіз мікро-сервісної архітектури....16
1.4 Аналіз вимог до системи....19
1.5 Дослідження інфраструктури системи пошуку....22
1.6 Дослідження щодо роботи з базою даних підприємства....24
1.7 Дослідження оптимізації запитів до бази за допомогою кешування....26
1.8 Постановка завдань будування архітектури системи....29
2 ОПИС ПРИЙНЯТИХ ПРОЕКТНИХ РІШЕНЬ....30
2.1 Обґрунтування вибору мікросервісної архітектури....31
2.2 Обґрунтування використання хмарних обчислень....37
2.3 Обґрунтування вибору провайдера хмарних обчислень....41
2.4 Вибір необхідного стеку хмарних сервісів 47
2.5 Використання Amazon Step Functions....52
3 БУДУВАННЯ АРХІТЕКТУРИ СИСТЕМИ ПОШУКУ....59
3.1 Приклад бази даних підприємства із спеціалізованою літературою....59
3.2 Будування архітектури системи пошуку як окремого цілого сервісу....62
3.3 Будування архітектури системи пошуку як ланки окремих сервісів....70
3.4 Будування архітектури системи пошуку як автомату стану....73
3.5 Порівняння підходів....75
ВИСНОВКИ....82
ПЕРЕЛІК ПОСИЛАНЬ....84
ДОДАТОК А....87
ДОДАТОК Б....104

1 МОНОЛІТНА АРХІТЕКТУРА ОКРЕМОГО СЕРВІСУ СИСТЕМИ ПОШКУКУ 108
1.1 Розробка прошарку доступу до даних.....108
1.2 Розробка прошарку бізнес-логіки.....109
1.3 Розробка прошарку контролерів.....110
2 СИСТЕМА ПОШУКУ, РОЗБИТА НА БЕЗСЕРВЕРНІ ФУНКЦІЇ AWS LAMBDAS.....112
2.1 Розробка функції запуску алгоритму пошуку.....112
2.2 Розробка функцій фільтрувань.....112
3 БЕЗСЕРВЕРНІ ФУНКЦІЇ СИСТЕМИ ПОШУКУ, ОБ'ЄДНАНІ АВТОМАТОМ СТАНУ AWS STEP FUNCTIONS.....114
3.1 Розробка функції запуску алгоритму пошуку.....115
3.2 Розробка функцій фільтрувань.....115
3.3 Розробка Step Function State Machine.....117
ДОДАТОК В.....119

ПЕРЕЛІК УМОВНИХ ПОЗНАЧЕНЬ, СИМВОЛІВ, ОДИНИЦЬ, СКОРОЧЕНЬ ТА ТЕРМІНІВ

БД – база даних;

AWS – Amazon Web Services;

GCP – Google Cloud Platform;

S3 – Simple Storage Service;

SQL – Structured Query Language (мова структурованих запитів);

HTTP – HyperText Transfer Protocol (протокол передачі гіпертекстових файлів);

MVC – Model-View-Controller (архітектурний шаблон підходу до створення об’єктно-орієнтованих веб-додатків);

MS – Microsoft;

ERD – Entity Relationship Diagram (модель даних, що дозволяє описувати концептуальні схеми предметної області);

DAO – Data Access Object (об’єкт доступу до даних);

EC2 – Elastic Cloud Computing

ВСТУП

У наш час вже досить важко уявити якісь сервіси надання послуг, що не мають свого додатку чи веб-сайту, тобто не пов’язані із Інтернетом. Глобальна мережа охоплює усі сфери інтересів людей, тому електронні літературні ресурси не являються винятком.

Відмінною рисою сучасного етапу розвитку суспільства виступає існування переважної більшості інформації в електронній формі, існує досить багато сервісів, що надають людям змогу шукати статті, журнали, книжки тощо, але, якщо торкнутися спеціалізованої літератури, то можна побачити, що цей напрям не надто доступний в Інтернет-мережі. Однак, враховуючи антипіратську спрямованість світового законодавства, як результат маємо те, що численні масиви вузькопрофільної літератури впливових видань, наукових праць сучасних дослідників зберігаються виключно в базах даних підприємств, де вони були опубліковані, що означає неможливість їх відкритого доступу у будь-яких пошукових мережах, що означає розташування такого ресурсу знаходиться під приватною системою або мережею підприємства, де система пошуку може бути не такою швидкою або «розумною» через неправильну або застарілу архітектуру.

Актуальність теми роботи полягає у ключовій необхідності таких підприємств створення своєї системи зберігання та пошуку літературних ресурсів для співробітників або учнів закладу, що дозволить на якісно новому рівні організувати процеси пропонування і поширення інформації. Тобто кожен, хто має доступ до ресурсів підприємства, що зберігає електронні ресурси, буде мати можливість обирати пропоновану літературу в залежності від специфічних параметрів щодо користувача, наприклад, відділ праці, спеціалізація та інше.

Відмінною рисою такої підсистеми пошуку спеціалізованої літератури є можливість будування загальної та незалежної архітектури за допомогою провідних технологій хмарних постачальників та мікро-сервісного принципу з використанням різних пошукових механізмів і засобів доступу до різноманітних

колекцій електронних даних. Так як в якості відповіді на запит до електронної бібліотеки користувачеві може бути представлений далеко не єдиний електронний документ (або його фрагмент) і зовсім не обов'язково в текстовій формі, необхідні такі інформаційні системи, які можуть забезпечити ефективний комплексний пошук і аналіз інформації в колекціях різнорідних об'єктів із максимально константним часом виконання та виключно без збоїв. Така підсистема завдяки своїй архітектурі повинна легко масштабуватися, додаючи окремі нові критерії до пошукового алгоритму, інтегруючись з різними системами, приймаючи та реплікуючи будь-яку базу даних підприємства, що включив її як модуль до своєї електронної системи.

У таких спеціалізованих місцях багато матеріалів, тому питання правильної архітектури їх системи постає загостреним, адже від цього залежить швидкість та якість роботи додатка. Для досягнення мети успішної системи, що надає доступ до спеціалізованої літератури, треба обробляти достатньо великий об’єм даних досить швидко, правильно підбирати пропозиції літератури для кожного з користувачів, працювати без збоїв та постійно, при цьому не витрачати на підтримку такої системи багато фінансів. Стандартний сервер десь у тому самому підприємстві, може не дотриматися усіх цих вимог, бо навіть відключення електроенергії або несправність операційної системи просто призупинить роботу усієї системи.

Внаслідок застарілих моделей надання послуг та складності пошуку матеріалів різко знизився рівень популярності навіть провідних бібліотек та інформаційних центрів. Очевидно, що в найближчі роки традиційними методами змінити ситуацію на краще не виявляється можливим, тому стали необхідні нові нетрадиційні підходи для вирішення проблеми підвищення рівня інформаційного забезпечення студентів, вчених і фахівців за допомогою нових алгоритмів знаходження книжкових джерел різноманітного тематичного походження.

Мета виконання дипломної роботи полягає у проведенні досліджень щодо створення підсистеми пошуку, а саме:

– дослідити та навести варіанти побудування мікро-сервісної архітектури системи пошуку спеціалізованої літератури, у тому числі за допомогою хмарних обчислень, в якості підготовки матеріалів для виконання атестаційної роботи магістра.

– проаналізувати та навести інфраструктуру, як буде інтегруватися система пошуку спеціалізованої літератури;

– проаналізувати та навести варіанти архітектури системи та дослідити переваги мікросервісів у даній роботі;

– проаналізувати та навести варіанти використання хмарних обчислень задля досягнення мети незалежного сервісу пошуку, що буде швидким та надійним.

1 АНАЛІЗ ПРЕДМЕТНОЇ ОБЛАСТІ

1.1 Опис об’єкта дослідження

Як вже будо зазначено, найчастіше місця, де зберігається досить багато спеціалізованих матеріалів статей, наукових робіт тощо, являються університети або підприємства, де працюють або працювали саме автори цих літературних джерел.

Тобто простіше кажучи, кожен такий веб-ресурс має свою електронну бібліотеку, де можна здійснювати пошук того, що потрібно користувачеві, тому що Інтернет не може замінити бібліотек, оскільки пересичений невпорядкованою інформацією з відсутністю обмеження її складу, в тому числі морального і політичного характеру.

Основна відмінність досліджуваної моделі бібліотеки полягає у застосуванні новітньої технології з метою привернення уваги сучасної медіа-читацькій аудиторії, що можливе лише за відповідності їх вимогам для сучасного розвитку бібліотеки. Цією технологією може бути підсистема пошуку спеціалізованої літератури – складова частина, що має міститись у будь-якій системі електронної бібліотеки задля забезпечення зручного доступу користувачів інформаційних джерел.

Користувачі звертаються до сервісу електронної бібліотеки, коли потрібно знайти літературний матеріал. Система відрізняє кожного конкретного користувача від інших, аналізує його вид діяльності (предмет, який викладає кафедра, на якій навчається студент; проект, на якому працює робітник тощо) та пропонує матеріали відносно отриманих даних.

Сервіс має бути простим для використання та мати усі необхідні для роботи дані щодо підприємства, де він запущений, тобто варіант створення такого виду

системи може використовуватися незалежно від підприємства, адже для пошуку потрібні лише «фільтри» за якими шукати та дані, що надаються підприємством.

У даній роботі пропонується розглянути реалізацію такої підсистеми пошуку або пропонування літератури за допомогою мікро-сервісів та розглянути варіанти створення архітектури такої реалізації.

1.2 Аналіз монолітної архітектури

Монолітна архітектура - це традиційний підхід до розробки програмного забезпечення. Додаток складається з трьох основних рівнів: користувальницький інтерфейс, бізнес-логіка та доступ до даних. Ці компоненти програми регулюються однаковою ієрархією, а не як окремі служби. Значення моноліту зводиться до того, щоб мати єдину базу коду та рівень даних для всієї програми (рис.1.1).

Рисунок 1.1 – монолітна архітектура

Компонент архітектури, який відповідає за інтерфейс програмного забезпечення, його взаємодію з користувачами та видимі функціональні можливості. Існує багато підходів до побудови інтерфейсу, але найпоширенішим є Model-View-Controller.

Модель зберігає інформацію, яку відображатиме користувальницький інтерфейс. Модель відповідає за зберігання персональних даних користувачів, даних облікового запису та налаштувань. Часто ці дані зберігаються в текстовому об'єкті поля для швидшого доступу, а не в базі даних.

View відповідає за відображення даних моделі. Він контролює розташування даних на екрані, їх стиль та взаємодію з користувачем. View надає

елементи керування користувачам, збирає введені користувачем дані, реагує на дії та видає вихідні дані. В основному він управляє даними в моделі.

Контролер відповідає за об'єднання двох компонентів. Він контролює дії представлення даних, надсилання запитів на оновлення та управління зберіганням даних. Якщо частину інформації потрібно передати в модель, це обов’язок контролера - подбати про неї [1].

Компонент бізнес логіки монолітної архітектури відповідає за взаємодію між бізнес-об’єктами та методами, в яких вони обробляються та оновлюються. Бізнес-логіка описує шлях виконання певних корисних завдань. Бізнес-логіка обробляє фактичні об’єкти бізнесу - запаси, позики, рахунки. Бізнес-логіка програми визначає, корисна ця функція чи ні. Це вираження бізнес-цілей

Рівень даних визначає правила та методи, що використовуються для зберігання, отримання, обробки та передачі даних. Він включає базу даних та систему управління базою даних. Цей компонент монолітної архітектури призначений для забезпечення всіх інших компонентів інформацією.

Що таке монолітний рівень даних? Він описує операцію обробки, критерії використання даних, їх стан та безпеку. Цей компонент повинен працювати стабільно та безпечно. Проблеми з архітектурою рівня даних призведуть до зниження продуктивності, витоків даних, неможливості обробити величезну кількість запитів та зберігати необхідні дані.

Зазвичай, монолітна архітектура застосовується при виконанні умов:

– єдина команда працює над проектом протягом тривалого періоду часу;

– додаток невеликий і легко поміщається в три основні шари;

– команда невелика, і немає можливості дозволити розробникам спеціалізуватися, будуючи певний функціонал;

– швидкість релізів та низькі витрати на проект є пріоритетними.

1.3 Аналіз мікро-сервісної архітектури

В останні роки навколо так званих мікросервісів дуже багато галасу. І це справді не дивно. У міру того, як програмні додатки стають дедалі складнішими, вони також стають громіздкими та більш комплексними для управління, оновлення та обслуговування. Архітектура мікросервісів - це відносно новий підхід до розробки програмного забезпечення, який вирішує цю проблему шляхом розділення складного програмного забезпечення на незалежні будівельні блоки або послуги. Ці модульні служби можуть бути розгорнуті та керовані індивідуально, роблячи великі програми більш надійними для змін.

Мікропослуги як архітектурний стиль датуються 2005 роком, коли Пітер Роджерс, засновник ресурсно-орієнтованих обчислень, вперше ввів термін "мікро-веб-послуги". Намагаючись створити більш гнучку та сервісно-орієнтовану архітектуру програмного забезпечення, він успішно окреслив принципи добре розробленої платформи мікросервісів. Через два роки архітектор програмного забезпечення Юваль Леві розширив ідею поглибленого використання послуг і надихнув кілька тематичних досліджень під керівництвом експертів на перспективні результати. Сьогодні архітектура мікропослуг вважається не лише актуальною темою, але й майбутнім ефективної розробки та обслуговування програмного забезпечення [2].

Архітектура мікросервісів описує певний стиль, в якому розробляються програмні додатки. На відміну від традиційних архітектурних стилів, що мають на меті побудувати програмне забезпечення як єдине ціле, архітектура мікропослуг базується на модульному підході. Таким чином, додатки розробляються як набори, що складаються з безлічі мікросервісів, які можуть бути незалежно розгорнуті та керовані. Але що таке мікросервіси? Незважаючи на те, що чіткого визначення мікропослуг як таких не існує, існують певні характеристики, пов'язані з цим революційним підходом до розробки програмного забезпечення.

Однією з основних характеристик є спосіб, яким архітектурний стиль компонент складного застосування. Програмне забезпечення розбито на кілька компонентів, які створені як самостійно замінювані та оновлюванні служби. Ці

служби можна визначити як процеси, які часто обмінюються даними через мережу з використанням технологічно-агностичних протоколів. Мікросервіси також створені з урахуванням можливостей бізнесу. Оскільки вони децентралізовані та мають невеликі розміри, вони пропонують велику гнучкість, коли мова заходить про мови програмування, апаратне забезпечення та програмне середовище (рис. 1.2)

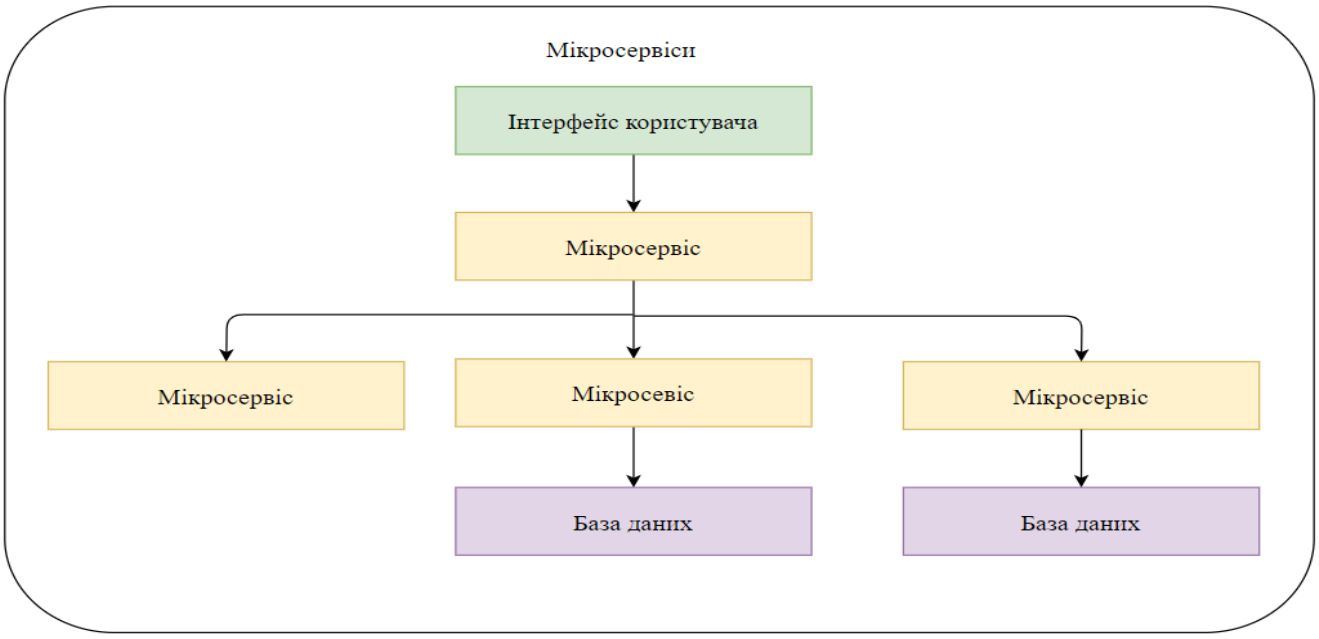

Рисунок 1.2 – Мікросервіси

Ідея вільно пов'язаних між собою сервісів, які можна легко підтримувати та перевірити, звучить інтригуючи для компаній, які стикаються з IT-проблемами через складні програми. Але перш ніж викорінити систему на місці, життєво важливо зрозуміти, як насправді працюють мікросервіси. Для того, щоб створити функціонуючу модульну архітектуру, підприємству потрібно спочатку визначити свої можливості та визначити послуги, які працюватимуть незалежно одна від одної. Схема архітектури мікросервісу потрібна для викриття залежностей, які заважають гнучкості кожної служби. Дуже важливо забезпечити, щоб помилка в окремій частині не впливала на всю систему.

Але навіть незважаючи на те, що мікросервіси розроблені для незалежної роботи, вони іноді все одно мають залежності від інших мікросервісів і потребують зв’язку між собою. Це спілкування відбувається завдяки використанню API (інтерфейсів програмування програм); синхронні або асинхронні протоколи, такі як HTTP / REST або AMQP. В ідеалі кожна мікрослужба має власну базу даних для децентралізації відповідальності та управління оновленнями в індивідуальному порядку. При використанні шаблону "База даних на сервіс" для забезпечення узгодженості даних встановлюється сага, яка є транзакцією, що охоплює кілька служб.

Однак нерідкі випадки, коли кожна мікросервіс з часом розвиває все більше і більше залежностей від послуг. Це нормальне явище, коли послуги починають зростати і починають впливати на кількість екземплярів, розташування та протоколи. Шлюзи API допомагають уникнути проблем, які виникають внаслідок цих змін. І ось як це працює: Шлюз API приймає всі запити API, а потім скеровує на кожен окремий мікросервіс. Він також викликає серверні сервіси та узагальнює результати, щоб визначити найкращий шлях. Він служить точкою входу для кожного запиту, організовує весь вхід і створює спрощений інтерфейс для користувачів.

1.4 Аналіз вимог до системи

Підсистема пошуку спеціалізованої літератури – це комплексна система функціональних ланок, робота яких повинна бути злагодженою та коректою задля забезпечення успішного сценарію існування на ринку електронної бібліотеки, складовою частиною якої являється цей додаток. В залежності від сучасних підходів та тенденцій у сфері науково-літературних джерел та конкуренцію, необхідно забезпечити оперативність та точність роботи як окремої підсистеми

пошуку, так і інших підсистем бібліотеки в цілому, таких як управління персоналом, змістом джерел тощо. Задля запровадження такої правильного життєвого циклу системи потрібно реалізувати автоматизовану роботу бібліотеки, а саме впровадити автоматизовані системи управління(АСУ).

Інформаційна система для бібліотечних підприємств являється узагальненим комплексом інтегрованих підсистем, серед яких пошук спеціалізованої літератури, які створюють середу взаємодії високої ефективності між користувачами, бібліотекарами, постачальниками, адміністраторами, авторами джерел та підприємством, де розташована система. І хоча ціна таких систем може оказатися високою, згідно з дослідженнями різноманітних компаній, потрібно встановлювати нову систему періодично для надання їй актуальності і швидкості роботи.

Практично усі постачальники програмного забезпечення різних напрямів мають свої версії АСУ, розрахованих для віддаленого використання, що працюють за допомогою застосуванням Інтернет-технологій: ASP (Application Server Provider) і клієнт-сервер на основі SQL (Standart Query Language).

Існують основні критерії оцінки доцільності інвестицій в автоматизовані системи управління літературними ресурсами підприємства, наприклад, пошуку та пропонування спеціалізованих видань для робітників чи учнів закладу:

– підвищення продуктивності роботи;
– максимальне використання наявних ресурсів бібліотеки.

Основними вимогами таких підприємств до систем пошуку інформаційних джерел є:

– швидкість та точність пропонування;
– надійність і відмовостійкість;
– супровід і цілодобова підтримка;
– постійний розвиток систем відповідно до вимог ринку.

Усі завдання реалізуються за допомогою АСУ в залежності від розміру фонду книг, підприємства, де буде встановлена система, кількості користувачів.

Побудуємо діаграму послідовності для визначення вимог кроків роботи алгоритму пошуку спеціалізованої літератури. Вони використовуються для уточнення діаграм прецедентів, більш детального опису логіки сценаріїв використання. Якщо потрібно документувати проект з точки зору сценаріїв використання, то sequence діаграми - це найкращий спосіб зробити це.

У таких діаграм зазвичай наявні об'єкти, які працюють та взаємодіють між собою в рамках сценарію, повідомлення, якими вони обмінюються, і які повертаються результати, пов'язані з повідомленнями [3].

Об'єкти позначаються прямокутниками з підкресленими іменами (щоб відрізнити їх від класів), повідомлення (виклики методів) - лініями зі стрілками, а повертаються результати - пунктирними лініями зі стрілками.

Прямокутники, що відображаються на лініях вертикального, виходячи з кожного об'єкту показують фокус об'єктів або їх "час життя".

За допомогою таких схематичних відображень у вигляді діаграм можна побачити, як система підприємства, яка інтегрована із сервісом пошуку буде взаємодіяти із цією службою. В якості прикладу наведено декілька фільтрів через які буде проходити алгоритм пошуку, кожен з яких буде являтися окремою функцією, незалежно від того, як саме буде представлена архітектура системи пошуку. Результат можна побачити на рисунку 1.3.

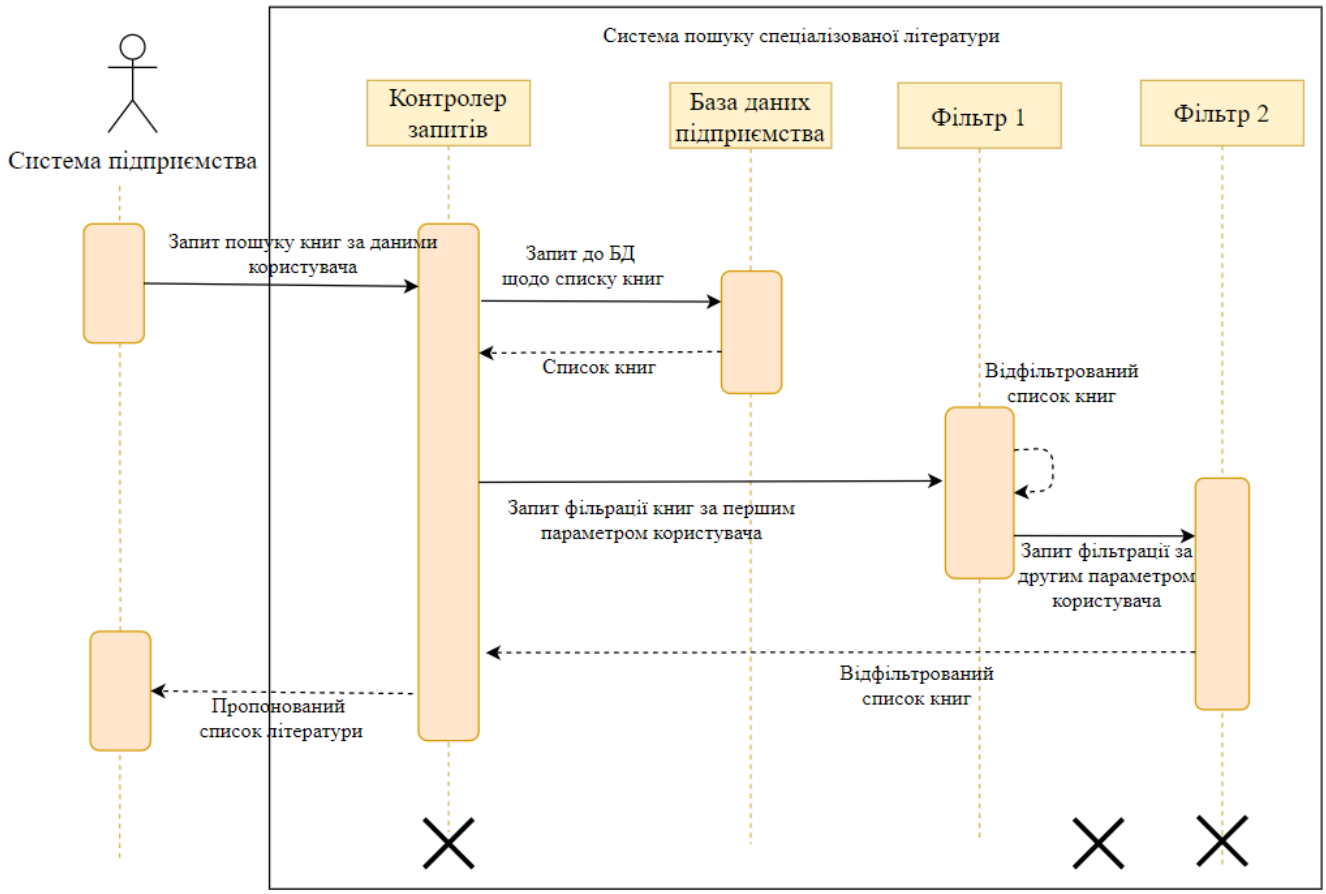

Рисунок 1.3 – Діаграма послідовності системи пошуку літератури

На цій діаграмі зображено, що система закладу взаємодіє з окремою системою пошуку (це може бути взаємодія через протокол HTTP), а вже система пошуку власноруч йде до бази підприємства, забирає необхідні дані для фільтрації та, базуючись на параметрах користувача, формує пропонований список літератури, проходячись по фільтрах в залежності від цих самих параметрів користувача. Як приклад, було приведено два фільтри, але звісно їх може бути більше.

1.5 Дослідження інфраструктури системи пошуку

Щоб розібратися у цьому питанні треба розуміти концепти розгортування сервісу пошуку літератури на локальній інфраструктурі та варіанти використання хмарних підходів.

Локальна інфраструктура є домінуючою моделлю обчислень підприємств протягом більше 50 років. Організації підтримують власне обладнання та програмне забезпечення в центрі обробки даних з повним контролем над усіма аспектами обробки, планування, адміністрування та обслуговування. Багато організацій в регульованих галузях промисловості не мали іншого вибору, крім як використовувати локальну модель, через необхідність жорсткого контролю над процесами даних та документування. Однак значні капітальні витрати, пов'язані з побудовою та підтримкою локальної інфраструктури, змушують багато організацій перекласти деякі або всі свої робочі навантаження на більш гнучкі хмарні варіанти. Однак локальні обчислення не зникнуть найближчим часом. Спадкове обладнання та додатки можуть бути несумісними із хмарним середовищем, і організації, які хочуть захистити інвестиції в апаратне та програмне забезпечення, можуть вирішити підтримувати інвестиції в приміщення роками, поки цикли виробництва не закінчиться, і програми не зможуть бути перероблені.

Наступним варіантом пропонується розглянути публічні хмарні обчислення.

Загальнодоступна хмара надає ресурси, такі як процесори, пам'ять, операційні системи, програми та сховище, доступними через загальнодоступний Інтернет на основі оплати за використання. Це можна сприймати як «комп'ютер у небі». Публічна хмара схожа на використання локального сервера, але сервер віртуалізується та управляється в іншому місці хмарним постачальником із високим ступенем автоматизації.

Організації використовують публічну хмару з різних причин, але найпопулярнішими є гнучкість, масштабованість та простота адміністрування. Загальнодоступні хмарні екземпляри можна запускати кількома клацаннями миші і так само легко знімати їх, коли вони більше не потрібні. Розробники та кінцеві користувачі в багатьох випадках можуть розгортати власні хмарні екземпляри без узгодження з IT та супутніми затримками. Виставлення рахунків, як правило, базується на використанні, що надає організаціям підзвітності та гнучкості оплачувати лише ті ресурси, якими вони користуються. Загальнодоступні хмарні

екземпляри можна масштабувати або зменшувати з відносною легкістю, і багато постачальників хмарних послуг пропонують найкращі в своєму роді засоби автоматизації для полегшення адміністрування. Громадська хмара також є чудовою платформою для розробки додатків, які будуть "жити" у хмарі, таких як програми, призначені для використання на мобільних пристроях або з послугами, які доступні через API [4].

Але в даній роботі йдеться про спеціалізовані літературні джерела, що можуть бути захищені авторським правом авторів-працівників підприємств чи університетів. В такому випадку публічна хмара з відкритим доступом в Інтернет не підходить. На допомогу приходить приватна хмара.

Для організацій, які бажають гнучких переваг автоматизації публічної хмари, але їм потрібно тримати ресурси на місцях з метою контролю чи дотримання вимог, приватна хмара є популярною альтернативою. Ця модель забезпечує ті самі переваги щодо масштабованості, автоматизації та гнучкості, що і в публічних хмарних та локальних середовищах, які можуть бути фізично захищені та жорстко керовані. Приватні хмари можуть бути побудовані з використанням існуючої архітектури обладнання центрів обробки даних або ліцензовані у постачальників загальнодоступних хмарних служб, які можуть забезпечити, по суті, версію своїх існуючих служб, у безпечному середовищі. Справжня приватна хмара - це більше, ніж просто віртуалізація.

Отже, для створення підсистеми пошуку спеціалізованої літератури за мікросервісною архітектурою може бути використані хмарні обчислення з приватним підходом.

1.6 Дослідження щодо роботи з базою даних підприємства

Оскільки розглядається підприємство, що займається будь-якою своєю діяльністю, має якусь кількість користувачів, котрі являються авторами спеціалізованої літератури або просто належать до якихось розділів чи підрозділів закладу, для зберігання усієї цієї інформації потрібна база даних.

Але існує два найпопулярніших типи баз – реляційні та нереляційні. В чому їх різниця?

Реляційна база даних структурована, тобто дані впорядковані в таблиці. Багато разів дані в цих таблицях мають взаємозв'язок або залежності. Нереляційна база даних орієнтована на документи, тобто вся інформація зберігається в більшій кількості в списку JSON. У межах однієї конструкції або документа у вас будуть перераховані всі ваші дані.

Реляційні бази ще також називають SQL-базами. SQL - це скорочена назва мови структурованих запитів, що в основному означає дуже надійний спосіб сортування даних у вигляді таблиць, стовпців та рядків. Як структуровані дані в базі даних SQL? Сама таблиця була б справді складена з однієї змінної чи об'єкта, який ми б переглядали. Стовпець буде представляти саму точку даних, яку потрібно зберегти, а рядок - це запис точок даних на стовпець.

На відміну від реляційної бази даних, база даних NoSQL є менш структурованою / обмеженою у форматі, а отже, забезпечує більшу гнучкість та адаптованість. Якщо ви збираєтеся мати справу з набором даних, який не є чітко визначеним, тобто не організованим чи структурованим, ви, ймовірно, не матимете розкоші встановлювати визначені таблиці та взаємозв'язки між набором даних [5].

Розберемо основні відмінності між ними (табл. 1.1).

Таблиця 1.1 – порівняння реляційних та нереляційних баз даних

Аспект	Реляційні бази	Нереляційні бази
Зберігання даних	– зберігаються в рядках і стовпцях – рядки містять всю інформацію про один конкретний запис / сутність, а стовпці містять різні точки даних	– охоплює різноманітні масиви баз даних – кожна база даних має різну модель зберігання даних

Продовження таблиці 1.1 – порівняння реляційних та нереляційних баз даних

Аспект	Реляційні бази	Нереляційні бази
Схема бази даних та її гнучкість	– схема жорстка – кожний стовпець повинен бути визначений перед введенням даних, і кожен рядок повинен містити дані для кожного стовпця – всю базу даних потрібно змінити на випадок, якщо щось потрібно додати	– динамічний характер – інформацію можна додати будь-коли - рядки та стовпці можуть бути порожніми
Масштабування схеми бази даних та ціноутворення	– вертикальне масштабування – для більшої кількості даних потрібні більші сервери, що збільшує вартість – важко та трудомістко масштабувати базу даних	– горизонтальне масштабування – кілька серверів для простоти управління та економічної ефективності
ACID	– підтримується	– не підтримується
зв’язки сутностей	– підтримується	– не підтримується

Як було сказано вище, заклад, що має авторів видань та користувачів системи, що пов’язані з групою, яка мож бути пов’язана за направленністю такою, що можна знайти літературу, це означає наявність зв’язків у системі та вже визначену схему, що за критеріями відповідає реляційній базі.

Тому у даній роботі за основу бази даних закладу береться реляційна система бази даних, хоча варіант нереляційної також буде розглядатися пізніше в експериментальній частині.

1.7 Дослідження оптимізації запитів до бази за допомогою кешування

Кешування даних у пам’яті може бути однією з найефективніших стратегій для покращення загальної продуктивності програми та зменшення витрат на базу даних. Кешування може бути застосовано до будь-якого типу бази даних, включаючи реляційні бази даних, і найкраща частина кешування полягає в тому,

що її реалізація є мінімально інвазивною, і цим ви значно покращуєте продуктивність програми як за масштабом, так і за швидкістю.

Виклики бази даних. при створенні розподілених додатків, що вимагають низької затримки та масштабованості, існує низка проблем, які бази даних на базі дисків можуть поставити перед вашими програмами. Кілька загальних:

Повільна обробка запитів. хоча існує ряд методів оптимізації запитів і конструкцій схем, які можуть допомогти підвищити продуктивність запитів, швидкість отримання даних з диска плюс доданий час обробки запитів, як правило, призводить до часу відповіді вашого запиту на двозначну швидкість мілісекунд. найкраще. Це передбачає постійне навантаження та базу даних, яка працює оптимально.

Вартість масштабування: розподіляються дані в дисковій базі даних NoSQL або вертикально масштабуються в реляційній базі даних, масштабування надзвичайно великих зчитувань може коштувати дорого і може вимагати певної кількості реплік зчитування бази даних, щоб відповідати одному вузол кеш-пам’яті може доставляти з точки зору запитів в секунду.

Необхідність спрощення доступу до даних: Хоча реляційні бази даних забезпечують чудові засоби для взаємозв'язку моделі даних, вони не є оптимальними для доступу до даних. Існують випадки, коли ваші програми можуть захотіти отримати доступ до даних у певній структурі або поданні, щоб спростити пошук даних та підвищити продуктивність програми.

Перш ніж впровадити кешування бази даних, багато архітекторів та інженерів витрачають великі зусилля, щоб витягнути з бази даних якомога більше продуктивності. І хоча це робити з розумними сподіваннями - це здорово, намагатися вирішити проблему за допомогою неправильних інструментів непродуктивно. Наприклад, скажімо, що ви намагаєтеся знизити латентність запиту бази даних, робити це з розумними очікуваннями є найкращою практикою, але спроба кинути виклик законам фізики, пов’язаним із отриманням даних з диска, - це марна трата часу.

Як же допомагає кешування? Кеш бази даних доповнює вашу первинну базу даних, усуваючи на неї непотрібний тиск, як правило, у формі часто доступних прочитаних даних. Сам кеш може жити в багатьох областях, включаючи вашу базу даних, додаток або як самостійний шар [6].

Три найпоширеніші типи кеш-пам’яті баз даних:

Інтегровані кеші бази даних: деякі бази даних, пропонують інтегрований кеш, який управляється в рамках механізму баз даних і має вбудовані можливості запису. Коли базові дані змінюються в таблиці бази даних, база даних автоматично оновлює свій кеш, що чудово. На рівні програми немає нічого необхідного для використання цього кешу. Там, де інтегрованих кешів не вистачає - це їх розмір та можливості. Вбудовані кеші, як правило, обмежуються доступною пам’яттю, виділеною кешем екземпляром бази даних, і не можуть використовуватися для інших цілей, наприклад, для обміну даними з іншими екземплярами.

Локальні кеші: локальний кеш зберігає ваші часто використовувані дані у вашій програмі. Це не тільки пришвидшує пошук даних, але й видаляє мережевий трафік, пов’язаний із отриманням даних, роблячи пошук даних швидшим за інші архітектури кешування. Основним недоліком є те, що серед ваших програм кожен вузол має свій власний кеш-пам'ять, що працює у відключеному режимі. Інформація, що зберігається в окремому вузлі кешу, незалежно від того, чи кешовані дані його бази даних, веб-сеанси чи кошики для покупок користувачів, не може бути передана іншим локальним кешам. Це створює проблеми в розподіленому середовищі, де обмін інформацією є критично важливим для підтримки масштабованого динамічного середовища. Оскільки в більшості додатків використовується декілька серверів додатків, якщо кожен сервер має власний кеш, узгодження значень між ними стає основною проблемою.

Крім того, у разі відключення дані в локальному кеші втрачаються, і їх потрібно буде доставати заново, ефективно заперечуючи кеш. Більшість з цих мінусів пом'якшуються віддаленими кешами. Віддалений кеш (або “бічний кеш”)

- це окремий екземпляр (або кілька екземплярів), призначений для зберігання кешованих даних у пам’яті.

Коли затримка мережі викликає занепокоєння, можна застосувати дворівневу стратегію кешування, яка використовує локальний та віддалений кеш разом. Ми не будемо детально обговорювати цю стратегію, але вона використовується, як правило, використовується лише за крайньої необхідності, оскільки додає складності. Для більшості додатків додаткові накладні витрати на мережу, пов'язані з віддаленим кешем, мало викликають занепокоєння, враховуючи те, що запит на нього, як правило, виконується під час роботи в мілісекундах.

Віддалені кеші: віддалені кеші зберігаються на виділених серверах і зазвичай будуються на сховищах ключ / значення NoSQL. Вони забезпечують від сотень тисяч до мільйона запитів в секунду на вузол кеш-пам’яті.

Крім того, середня затримка запиту до віддаленого кешу виконується в затримці до мілісекунд, на порядок швидше, ніж на базі даних на диску. На таких швидкостях рідко потрібні локальні кеші. Оскільки віддалений кеш працює як підключений кластер, який може бути використаний усіма різними системами, вони ідеально підходять для розподілених середовищ.

1.8 Постановка завдань будування архітектури системи

Узявши за увагу усі дослідження, що були приведені у даній роботі, до завдань будування архітектури можна віднести наступні пункти:

– створити варіант монолітної системи, що зберігає літературні джерела та матиме систему пошуку;

– дослідити переваги та недоліки локальної інфраструктури;

– обґрунтувати переваги хмарних обчислень та інфраструктури порівняно з локальною;

– обґрунтувати обраних хмарний провайдер;

– дослідити сервіси, що надає обраний хмарний провайдер;

– обрати необхідні хмарні сервіси для реалізації підсистеми пошуку спеціалізованої літератури;

– дослідити варіант інтеграції нового модулю пошуку із існуючою системою електронних літературних ресурсів;

– порівняти можливі архітектурні рішення за характеристиками швидкості, надійності, ціноутворення;

– обрати варіант архітектури та підходящий для нього варіант використання.

2 ОПИС ПРИЙНЯТИХ ПРОЕКТНИХ РІШЕНЬ

2.1 Обґрунтування вибору мікросервісної архітектури

Пропонується розглянути, чому для підсистеми пошуку спеціалізованої літератури потрібно обрати саме мікро-сервісну архітектуру. Хоча і монолітна архітектура дуже популярна і на даний момент це найтрадиційніший підхід у розробці програмного забезпечення, це також означає, що одна помилка може вплинути на всю систему, і командам розробників потрібно розгортати нову версію додатка на стороні сервера щоразу, коли вносяться зміни. Тобто існують деякі проблеми, що приховує традиційний підхід.

Монолітні програми можуть перерости у «велику кулю грязі» - ситуацію, коли жоден розробник (або група розробників) не розуміє всю програму [7].

В монолітних додатках повторне використання модулю логіки додатка дуже обмежене через міцний зв’язок усіх модулей в одному додатку; масштабування монолітних додатків часто може бути проблемою; важко досягти оперативної спритності при багаторазовому розгортанні монолітних артефактів додатків.

Пропонується розглянути, як може виглядати монолітна архітектура системи підприємства із базою даних співробітників та їх спеціалізованої літератури із підсистемою пошуку та пропонування інформаційних джерел (рис. 2.1).

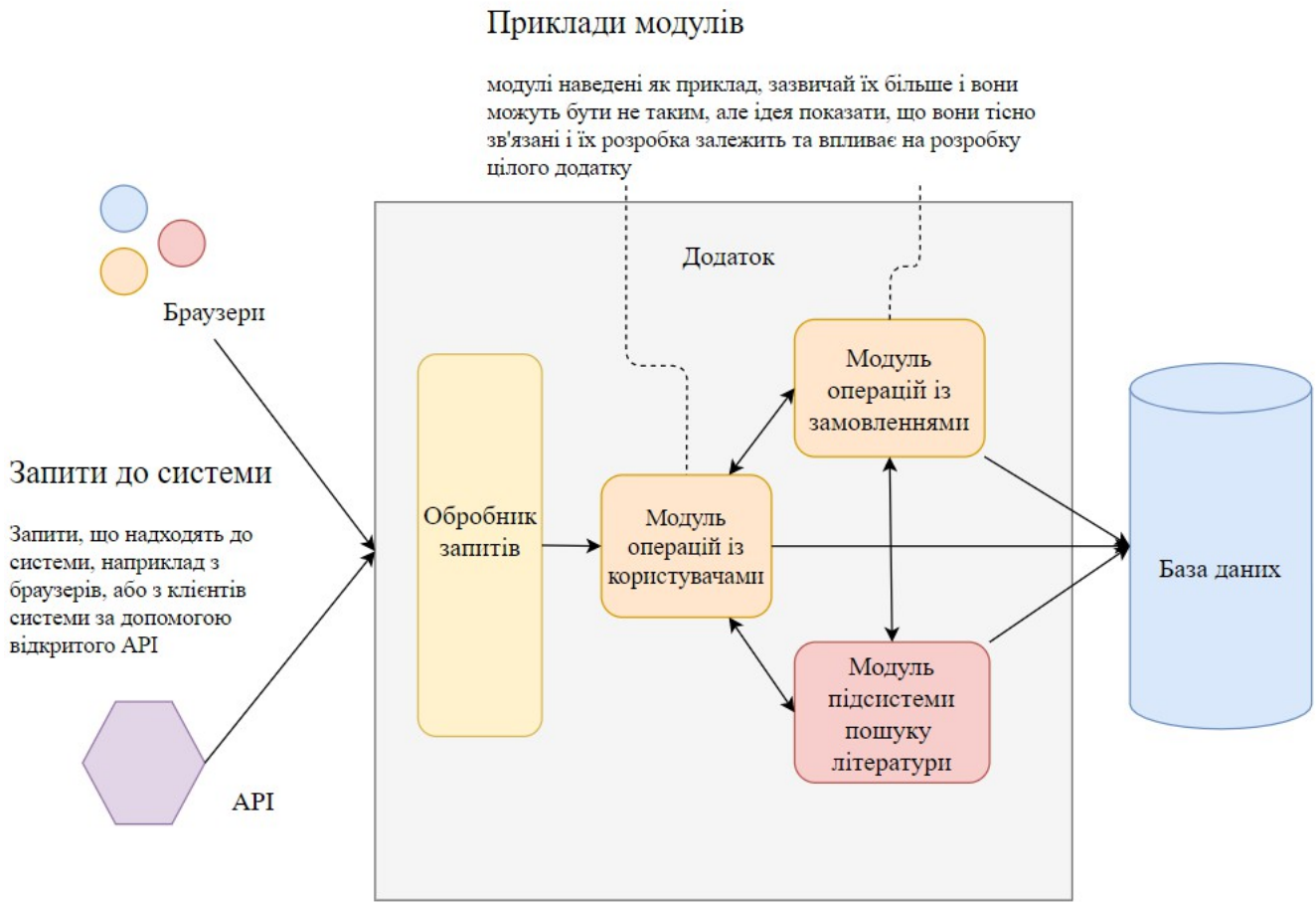

Рисунок 2.1 – Монолітна архітектура системи підприємства із підсистемою пошуку спеціалізованої літератури

З цього рисунку можна побачити, що у монолітній системі зазвичай логіка додатку розбита між модулями, які взаємодіють між собою та увесь додаток взагалі взаємодіє з базою даних. Які проблеми можуть спіткати при розробці системи пошуку у даній ситуації:

– неможливість незалежної реалізації: через те, що сервіс пропонування літератури являється тільки модулем усієї складної системи, він не може розроблятися окремо від неї. Потрібно мати код усього продукту, щоб розробляти певну частину, а з архітектурної точки зору, потрібно приймати в увагу усі тонкощі та аспекти вже побудованої загальної системи та приймати усі обмеження (мова програмування, технології та інше).

– відсутність версіонування змін конкретно підсистеми пошуку: кожен раз, коли потрібно буде змінити пошуковий алгоритм або ще щось, треба ініціювати оновлення цілої системи;

– неефективне використання бази даних: незалежно від того, реляційна чи ні база даних, якщо не передбачено систему кешування даних, пошук буде працювати значно повільніше, а використати готове рішення (наприклад одне з рішень провідних вендорів хмарних обчислень) може бути неможливим через обмеження у використанні технологій;

– неефективна робота самого пошуку: через те, що монолітна система вже написана якоюсь мовою програмування, з архітектурно точки зору, пошук може бути сповільненим (наприклад, мова Java повільніше працюватиме з фільтрацією великого об’єму даних, аніж Python, або у випадку нереляційних баз даних, мова TypeScript працювала б краще, адже усі її об’єкти являються JSON, як і об’єкти, що зберігаються у NoSQL базі.

Бізнес-переваги мікропослуг проти монолітної архітектури є значними. При правильному розгортанні архітектура на основі мікросервісів може принести значну цінність бізнесу. Цю величину можна виразити як уникненням технічного боргу, так і суттєвим підвищенням ефективності. Розберемо низку переваг одну за одною.

Доступ: оскільки мікросервіси розгортаються окремо і обмінюються даними через API, код і програми стають більш доступними для працівників поза розробкою. Модульний підхід до проектування також допомагає зрозуміти різні сервіси, які пропонує компанія, включаючи їх конкретні функції, не турбуючись про інші деталі.

Управління помилками: якщо архітектуру мікросервісів застосовано правильно, пошук та ізоляція помилок стає набагато простішою. Щоразу, коли виникає помилка, розробники точно знають, де шукати, що економить дорогоцінний час та сили. А виправлення лише однієї незалежно запущеної служби означає, що це не вплине на інший код, і немає необхідності розгортати цілу програму.

Технічне обслуговування: ця перевага йде паралельно з попередніми двома. Архітектура мікросервісів полегшує обслуговування та управління послугами. Насправді, кожною службою може керувати інша команда, оскільки це унікальний процес із власною базою даних. Крім того, можна виконувати окремі оновлення, не зупиняючи всю систему програмного забезпечення [8].

Адаптованість: мікросервіси нещільно пов'язані між собою та мають стандартизований спосіб спілкування між собою. Але оскільки вони побудовані окремо, їм не обов’язково використовувати одну мову або технологію програмування. Це означає, що кожна послуга може бути розроблена таким чином, щоб вона найкраще відповідала функціям, які вона повинна виконувати.

Еволюційні: вимоги бізнесу можуть змінюватися, що створює необхідність додавати нові функції до пропонованих послуг. У традиційних архітектурних підходах, що розглядають програмне забезпечення в цілому, це означає втручання у всю бізнес-логіку та потенційне порушення існуючих функцій. Завдяки своєму модульному підходу архітектура мікропослуг дозволяє уникнути цієї проблеми.

Масштабованість: однією з головних переваг, яку можуть запропонувати мікросервіси, їх здатність масштабуватися горизонтально. Це означає, що будь-яку розгорнуту службу можна продублювати, щоб уникнути вузьких місць через повільне виконання. Іншим варіантом підвищення продуктивності запуск служби на більш потужному обладнанні або декількох машинах, які паралельно обробляють дані.

Тестування та моніторинг: одиничні програмні компоненти набагато легше тестувати, ніж складні програми. Якщо кожну мікросервіс перевірено належним чином, програмне забезпечення в цілому має тенденцію бути більш надійним та надійним. А можливість ізолювати лише частини програми дозволяє покращити моніторинг та управління безпекою.

Постачання: протягом останніх кількох років технології автоматизації різко вдосконалились, що також спричинило еволюцію хмари та AWS. Завдяки технологіям автоматизації інфраструктури та принципу безперервної доставки

операційна складність створення, впровадження та тестування мікропослуг значно зменшилась.

Інтеграція: програмне забезпечення повинно мати можливість інтегрувати стільки нових функцій та послуг, скільки потрібно. Незважаючи на те, що це звучить логічно, інтеграція нових компонентів програмного забезпечення в існуючу програму може бути проблемою при традиційній розробці програмного забезпечення. Завдяки використанню інструментів безперервної інтеграції мікросервіси повністю революціонізували цей процес.

Багаторазове використання: часто забувається, але великою перевагою мікросервісів той факт, що незалежні компоненти, що складають ціле програмне забезпечення, можуть бути використані повторно для майбутніх проектів. Повторно використовуючи успішні процеси та лише незначно їх модифікуючи, команді розробників не потрібно винаходити колесо та споживає менше ресурсів.

Тобто використання архітектури, заснованої на мікропослугах, може призвести до набагато більш ефективного використання коду та базової інфраструктури. Нерідко спостерігається значна економія витрат на цілих 50% за рахунок зменшення обсягу інфраструктури, необхідної для запуску певної програми.

Дохід: швидші ітерації та зменшення простоїв можуть збільшити дохід (або за допомогою ефективності, створеної за допомогою ідеології зворотного стягнення плати, або за рахунок покращення залучення користувачів). Утримання та залучення користувачів зростає з постійними вдосконаленнями, пропонованими мікросервісами.

Розглядаючи конкретно тему даної роботи, нам потрібно розробити архітектуру окремого сервісу, котрий буде інтегруватися в систему електронної бібліотеки. Нам непотрібно «вбудовуватись» в неї, тому що помилка в роботі системи нашого пошукового алгоритму не повинна спричиняти помилки у самої бібліотеки. Це означає, що концепту мікросервісів не уникнути. Також алгоритм, що буде шукати та пропонувати літературу не повинен мати збій при помилці одного з фільтрів або при перенавантаженні серверу, де він розташований або ще

щось, тому що його робота – це єдине, що може пропонувати цей сервіс для системи бібліотеки.

У будь-якому випадку для інтегрування підсистеми пошуку спеціалізованої літератури із системою закладу, де зберігаються інформаційні ресурси, був обраний мікросервісний підхід (рис 2.2).

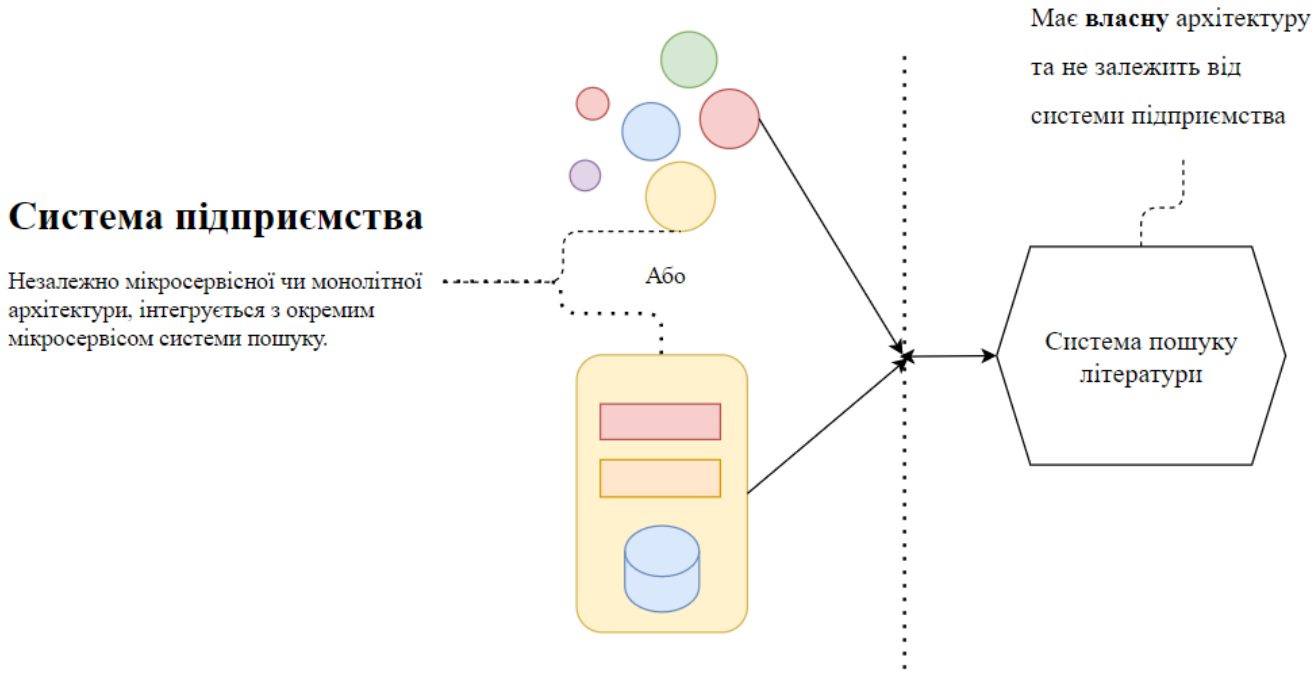

Рисунок 2.2 – Окремий мікросервіс системи пошуку літератури

На рисунку відображено, що система пошуку та пропонування літератури користувачеві повинна бути незалежною від системи підприємства та бути інтегрованою за допомогою якогось API (наприклад HTTP). Це надаватиме такі переваги:

– будування та розгортування тільки сервісу пошуку без загальної системи;

– може бути використана інша мова програмування та інші технології;

– навіть, якщо загальна система не використовує хмарні обчислення, система пошуку може їх мати в своїй реалізації;

– помилка у системі пошуку не впливатиме на роботу загальної системи.

2.2 Обґрунтування використання хмарних обчислень

Хмарні обчислення - чи не найяскравіша технологічна інновація 21 століття. Це пов’язано з тим, що він бачив найшвидше впровадження у загальнодоступний потік, ніж будь-яка інша технологія в домені. Це прийняття було спричинене головним чином постійно зростаючою кількістю смартфонів та мобільних пристроїв, які можуть отримати доступ до Інтернету. Хмарні обчислення призначені не лише для організацій та бізнесу; це також корисно і для пересічної людини. Це дозволяє нам запускати програми, не встановлюючи їх на наших комп’ютерах; це дозволяє нам зберігати та отримувати доступ до нашого мультимедійного вмісту через Інтернет, це дозволяє нам розробляти та тестувати програми без необхідності мати сервери тощо. Хмарні обчислення - це диво 21 століття, яке зберігає своє значення майже в усіх сферах, про які ви можете подумати [9].

Чому хмарні обчислення важливі для бізнесу?

Доведено, що об’єднані обчислювальні ресурси, доступні за допомогою хмарних обчислень, пропонують величезні переваги для бізнес-організацій. Розкриття цих переваг широко класифікується на три категорії, які включають ефективність, гнучкість та стратегічну цінність.

Ефективність: ефективність ділових операцій досягається наступними способами завдяки використанню хмарних обчислень:

Доступність: хмарні обчислення полегшують доступ до програм та даних з будь-якого місця в усьому світі та з будь-якого пристрою, що має Інтернет-з’єднання.

Економія витрат: хмарні обчислення пропонують підприємствам масштабовані обчислювальні ресурси, отже, економить їх на витратах на їх придбання та обслуговування. Ці ресурси оплачуються на основі принципу оплати праці, що означає, що підприємства платять лише за ресурси, якими вони користуються. Це виявилось набагато дешевшим, ніж самостійне придбання ресурсів.

Безпека: хмарні провайдери, особливо ті, що пропонують приватні хмарні послуги, прагнули запровадити найкращі стандарти та процедури безпеки, щоб захистити дані клієнта, збережені в хмарі.

Аварійне відновлення: хмарні обчислення пропонують найефективніші засоби для малих, середніх і навіть великих підприємств для швидкого та надійного резервного копіювання та відновлення своїх даних та програм.

Гнучкість: гнучкість досягається наступними способами при використанні хмарних обчислень:

Масштабованість: хмарні обчислення - найкращий варіант для підприємств з коливаннями робочих навантажень, оскільки хмарна інфраструктура масштабується залежно від потреб бізнесу.

Вибір інструментів: хмарні обчислення дозволяють компаніям вибрати конкретні попередньо побудовані інструменти та функції для пошуку рішень, адаптованих до їхніх конкретних потреб.

Хмарні параметри: хмарні обчислення пропонують приватні хмарні, загальнодоступні та гібридні хмарні рішення, кожна з яких має різні функції. Організації можуть вибрати ці варіанти залежно від того, що найкраще відповідає їх потребам.

Вибір контролю: підприємства можуть визначити рівень контролю за допомогою опцій як послуги, пропонованих хмарним провайдером.

Стратегічна перевага: хмарні обчислення дозволяють компаніям отримати стратегічну перевагу у своїй ніші наступними способами:

Підвищена продуктивність праці: постачальники хмарних послуг купують та керують базовою хмарною інфраструктурою, що дозволяє бізнесу зосередити свої сили на своїх основних бізнес-операціях.

Автоматичне оновлення програмного забезпечення: усі програмні програми, доступ до яких доступні через хмару, як правило, оновлюються. Це дозволяє компаніям отримувати доступ до найновіших функцій без необхідності самостійно підтримувати систему.

Конкурентоспроможність: підприємства, які використовують хмарні обчислення, здатні маневрувати більш спритно у порівнянні з конкурентами, які віддають свої сили придбанню та підтримці IT-ресурсів.

Посилення співпраці: завдяки можливостям хмарних обчислень люди з різних місць можуть співпрацювати у бізнес-проектах, не обов'язково знайомлячись.

Чому хмарні обчислення - це майбутнє.

Беручи до уваги численні переваги, які хмарні обчислення пропонують організаціям, можна обґрунтувати, що хмарні обчислення все частіше стають новою нормою. Хмарні обчислення допомагають суспільству впоратися з майбутніми проблемами, такими як управління великими даними, кібербезпека та контроль якості. На додаток до цього, нові технології, такі як штучний інтелект, технологія розподіленого реєстру та багато інших можливостей, стають доступними як послуги за допомогою хмарних обчислень.

Отже, ці технології мають бути пристосованими до різних платформ, таких як мобільні пристрої, отже, збільшується їх використання. Інновації, засновані на хмарних обчисленнях, такі як хмарна автоматизація та хмара промисловості, також розробляються для інтеграції хмарних обчислень у більш конкретні промислові заходи, що зробить різні операції ще більш впорядкованими. Остаточний вердикт хмарних обчислень полягає в тому, що це трансформаційна технологія, яка допомогла організаціям з різних юрисдикцій доставляти свої товари та послуги краще, ніж раніше.

Беручи в увагу те, що система пошуку спеціалізованої літератури – не основний компонент системи підприємства, адже воно може займатися основними своїми бізнес-напрямами, сервіс, що буде розроблюватися не повинен впливати на роботу загальної системи. Тобто підприємству непотрібно постійно підтримувати систему, оновлювати її, завершувати роботу у випадку, якщо пошук літератури не працює, не уповільнюватися та інше.

Також за версією журналу Forbes, хмарні вирішення багатьох технічних проектів набирають популярності та за прогнозами з кожним роком будуть все більш популярними (рис. 2.3).

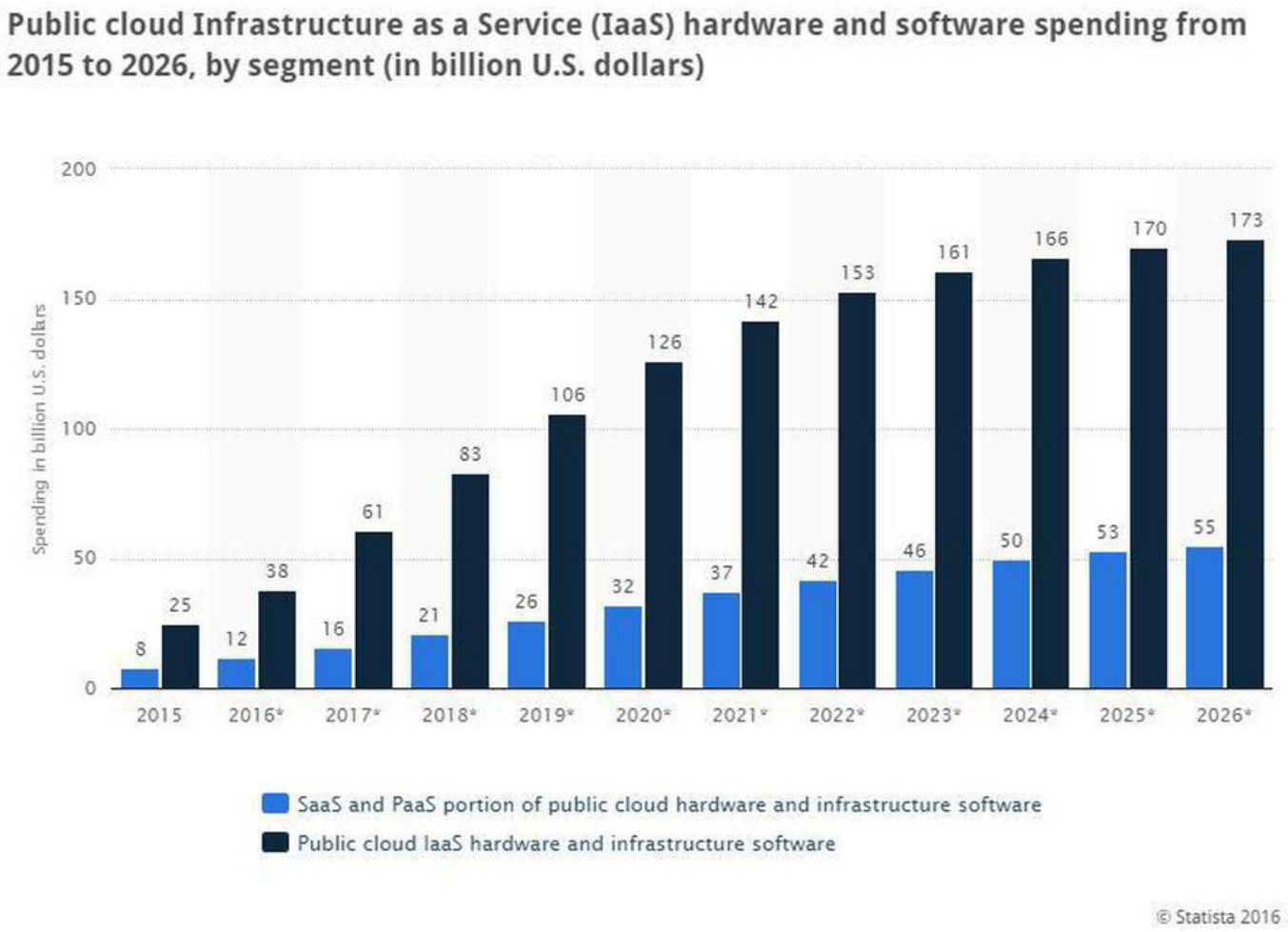

Рисунок 2.3 – зростаюча популярність використання хмарних обчислень

Сама принцип системи пошуку у тому, що це незалежна система. Тому підприємству не варто знати, якою мовою вона написана, за допомогою яких технологій, сервіс пошуку повинен бути масштабований, швидкий, безвідмовний та легко змінюємий. Це і дають нам хмарні обчислення, тому у даній роботі для розробки окремого модулю пошуку було вирішено використовувати хмари.

2.3 Обґрунтування вибору провайдера 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 - a

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 [10].

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 варантів.

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.

Потім Google здійснив свій гібрид у 2021 році, випустивши Anthos, який фактично є ребрендингом платформи Google Cloud Services і поєднує в собі комбінацію існуючого Google Kubernetes Engine (GKE), GKE On-Prem та Anthos Config Консоль управління. Це обіцяє уніфіковане адміністрування, політику та безпеку гібридних розгортань Kubernetes.

Тепер потрібно торкнутися досить важливого питання – ціноутворення.

Ціноутворення може бути великою привабливістю для тих, хто розглядає можливість переходу до хмари, і з поважною причиною: вже деякий час спостерігається тенденція до зниження цін, оскільки великі постачальники конкурують.

Загалом, ціни приблизно порівнянні, тим більше, що AWS у 2017 році змінила ціноутворення на подингові ціни на свої послуги EC2 та EBS, приводячи його у відповідність із Azure та Google.

Однак чітке порівняння може бути важким, оскільки всі три пропонують дещо різні моделі ціноутворення, знижки та часті зниження цін. Звичайно, не всі замовники також заплатять ціну за наклейку, особливо на рівні підприємства, де про знижки за обсяги можна домовитись з торговими представниками.

Усі постачальники пропонують безкоштовні ознайомчі рівні, що дозволяє клієнтам спробувати свої послуги перед покупкою, і, як правило, пропонують кредити для залучення інноваційних стартапів на свої платформи, а також «завжди безкоштовні» рівні з суворими обмеженнями використання.

Розглянемо плюси і мінуси AWS.

Як згадувалося раніше, причини вибору одного постачальника за іншого відрізнятимуться для кожного клієнта. Але є аспекти конкуруючих хмар, які за певних обставин дадуть переваги.

Широта та глибина пропозиції AWS розглядається як плюс для AWS.

Вони стали провідним постачальником, будуючи свій набір хмарних сервісів з 2006 року. Всі вони створені для зручності для підприємств, щоб вони сподобалися IT-директорам, а також його основній аудиторії розробників.

Постачальник високо оцінює варіанти конфігурації платформи, функції моніторингу та політики, безпеку та надійність. Партнерська екосистема та загальна стратегія продукту також розглядаються як провідні на ринку, а його AWS Marketplace має велику кількість сторонніх програмних послуг.

Але масштаб пропозиції від AWS іноді також розглядається як недолік. Незважаючи на те, що вибір цих опцій є привабливим, також може бути важко орієнтуватися у великій кількості пропонованих функцій, а деякі вважають AWS складним постачальником для управління.

Розглянемо плюси та мінуси Microsoft Azure.

Великою привабливістю для Azure є те, що Microsoft вже має міцні опори в організації і може легко зіграти роль у допомозі цим компаніям перейти на хмару. Azure, природно, добре взаємодіє з ключовими локальними системами Microsoft, такими як Windows Server, System Center та Active Directory.

Згідно з опитуванням 100 старших керівників IT, проведеним компанією Goldman Sachs у січні 2020 року, 56 заявили, що віддають перевагу Azure проти 48, які вибрали AWS.

Однак одним із мінусів є низка відключень протягом багатьох років, включаючи велике глобальне відключення в травні 2021 року.

Ще однм недоліком можна вважати «залежність від платформи». У той час, коли інші постачальники хмарних ресурсів можуть взаємодіяти з будь-якими системами, незалежно від виробника цієї системи, то Microsoft відрізняється високим рівнем надійності роботи тільки зі своїми локальними системами.

Розглянемо плюси та мінуси Google Cloud Platform

Google має хороший досвід роботи з інноваційними компаніями, що працюють у хмарі, і має міцну позицію в спільноті з відкритим кодом, але традиційно намагається вийти на ринок підприємств.

Її стратегія виходу на ринок була зосереджена на тому, щоб показати себе на менших, інноваційних проектах у великих організаціях, а не стати стратегічним хмарним партнером. Збільшення ширини своїх партнерських стосунків та підтримка дохмарного бізнесу та IT-процесів повинні стати

головними напрямками, якщо GCP хоче залучити більше традиційних підприємств.

Але недоліки у Google Cloud Platform досить складно визначити, адже це відносно новий постачальник послуг, тому його популярність тільки набирає обертів, тому в даному випадку складно визначити, чи буде надійно ця система у реалізації системи пошуку спеціалізованої літератури.

Який тоді провайдер потрібно обирати?

У дуже широких термінах AWS продовжує лідирувати з точки зору пропонування найширшого спектру функціональних можливостей та зрілості. Він і надалі залишається явним лідером ринку, але розрив скорочується.

Широкий перелік інструментів та послуг, а також зручні для підприємств особливості роблять його потужним пропозицією для великих організацій. Тим часом його величезна та постійно зростаюча інфраструктура забезпечує економію на масштабі, що дозволяє агресивне зниження цін.

Але у даній роботі, це не питання, що краще у загальному стані. Потрібно визначитися з провайдером, для цього пропонується звернутися до графіку популярності використання того чи іншого хмарного провайдеру за допомогою сервісу https://trends.google.com/, де оберемо 3 теми пошукових запитів: Amazon Web Services, Google Cloud Platform, Microsoft Azure. Виставимо фільтр, щоб шукало по всьому світу та за останні 5 років. Результат графіку на рис.2.4.

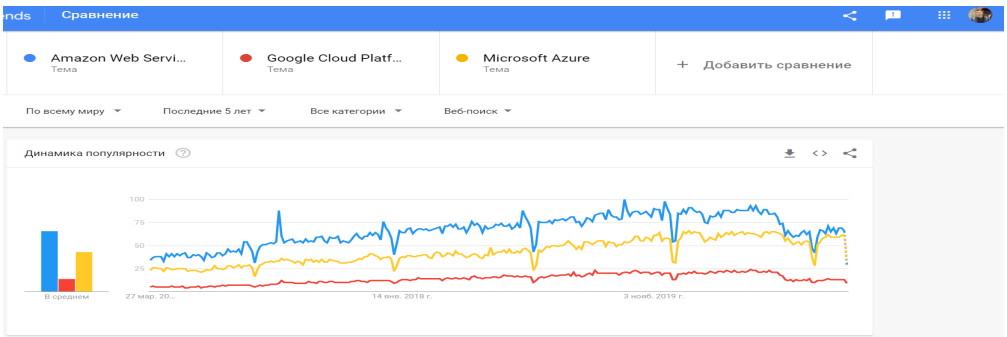

Рисунок 2.4 – порівняння AWS, GCP, Microsoft Azure

З досліджень та графіку популярності стає очевидно, що усі хмарні провайдери досить популярні та мають усі необхідні сервіси для вирішення проблеми будування архітектури системи пошуку спеціалізованої літератури.

Слід зазначити, що сервіси, котрі будуть використовуватись у системі пошуку, наявні і в Amazon, і в Azure, і в Google Cloud Platform. Це означає, що обравши будь-яку з платформ, можна буде мігрувати на іншу без втрат коду, без проблем написання та використання інших сервісів та без конфігурування усього заново.

Тому можна обирати будь-який з хмарних провайдерів. У даній роботі був обраний Amazon Web Services, оскільки він являється найпопулярнішим та має наглядну документацію із діаграмами та прикладами використання того чи іншого сервісу.

Слід також зазначити, що за версією авторитетного веб-журналу, AWS досі являється найбільш популярним вибором серед технічних рішень різноманітних проектів (рис. 2.5).

Рисунок 2.5 – AWS як найпопулярніший хмарний провайдер

2.4 Вибір необхідного стеку хмарних сервісів

Для того, щоб ефективно використовувати хмарні обчислення задля будування архітектури системи пошуку спеціалізованої літератури треба обрати усі необхідні для цього сервіси, що можуть знадобитись у даній реалізації.

Почнемо з варіанта, коли в нас підсистема пошуку – це окремий мікросервіс, котрий включає у себе роботу усього пошуку, тобто фактично моноліт модулю пошуку. Це означає, що наша підсистема буде розгортуватися та збиратися як один файл виконання, котрий буде відповідати за інтегрування з загальною системою підприємства, прийматиме запит із усіма необхідними даними про користувача, потім система буде з'єднуватись з базою даною підприємства, витягувати собі усі необхідні дані про літературу, фільтрувати за обраними критеріями, наприклад, за усіма характеристиками користувача та повертатиме відфільтрований список назад до підприємства.

Фактично, це буде клієнт-серверний додаток, що буде доступний для підприємства за посиланням. Такий додаток повинен відповідати усім принципам SOLID та бути побудованим за REST – архітектурою, що означає:

– мати шарову структуру додатку, де перший шар контролерів приймає усі запити до системи та повертає результат

– другий шар сервісів буде приміняти бізнес-логіку, у даному випадку фільтрацію

– третій шар доступу даних ходитиме до бази підприємства та витягуватиме усі необхідні дані

Для реалізації такого додатку, очевидно потрібна серверна мова програмування, що має можливості з'єднання з БД та швидкої обробки даних, бажано, паралельно. До таких мов відноситься Java та її основний фреймворк – Spring. За допомогою цієї мови та її технологій можна за допомого збірника проектів, наприклад, Maven, можна побудувати такий додаток.

Збірник проектів включатиме усі бібліотеки, від котрих залежить робота додатка, та за допомогою його команд буде збирати увесь додаток в один виконуваний файл для розгортки та запуску системи пошуку.

Усе, що у даному випадку будування архітектури залишається потрібним – сервер, на котрому буде виконуватись цей додаток. Для цього обраний провайдер AWS має окремий сервіс, що зветься EC2.

Amazon Elastic Compute Cloud (Amazon EC2) забезпечує масштабовані обчислювальні можливості в хмарі Amazon Web Services (AWS). Використання Amazon EC2 позбавляє вас необхідності вкладати гроші в апаратне забезпечення, щоб ви могли швидше розробляти та розгортати програми. Ви можете використовувати Amazon EC2 для запуску стільки або стільки віртуальних серверів, скільки вам потрібно, налаштування безпеки та мережі та управління сховищем. Amazon EC2 дозволяє збільшувати або зменшувати масштаби, щоб впоратися зі змінами у вимогах чи стрибками популярності, зменшуючи потребу прогнозувати трафік [11].

Amazon EC2 надає такі функції:

– віртуальні обчислювальні середовища, відомі як інстанси

– попередньо налаштовані шаблони для ваших примірників, відомі як Amazon Machine Images (AMI), які упаковують біти, необхідні для вашого сервера (включаючи операційну систему та додаткове програмне забезпечення)

– різні конфігурації процесора, пам’яті, сховища та можливостей мережі для ваших примірників, відомі як типи примірників

– зберігання тимчасових даних, які видаляються при зупинці, сплячому режимі або завершенні роботи вашого екземпляра, відомі як томи зберігання екземплярів

– кілька фізичних розташувань для ваших ресурсів, таких як екземпляри та томи Amazon EBS, відомі як Регіони та Зони доступності

– статичні адреси IPv4 для динамічних хмарних обчислень, відомі як еластичні IP-адреси

Ви можете створити віртуальні мережі, які логічно ізольовані від решти хмари AWS і які за бажанням можна підключити до власної мережі, відомої як віртуальні приватні хмари (VPC).

Тобто фактично AWS надає нам обмежений безкоштовний сервер, де може бути запущений додаток системи пошуку спеціалізованої літератури.

Коли вже є запущений сервер, неважливо, якої саме він архітектури, залишається питання доступу до бази даних. База підприємства, звісно, буде розташована не прямо на тому ж сервері, що і підсистема пошуку і, незалежно від того, реляційна вона чи ні, швидкість запитів буде низькою.

Тобто кожен раз, коли потрібно дістати якусь інформацію з бази, система пошуку буде витрачати час на з'єднання з сервером БД, відсиланням запиту та отриманням запиту, але сервер тієї ж бази може бути в цей час навантажений запитами підприємства. Що робити у цей час?

Як вже розглядалося, вирішенням цієї проблеми може бути налаштування кешу бази даних. Для цього AWS надає можливості сервісу ElastiCache або DAX.

ElastiCache / DAX - це сховище даних у пам'яті, одним із основних випадків використання якого є використання як кеш-пам'яті.

Він ніколи не буде безпосередньо підключатися до вашої бази даних, натомість значення, отримані з бази даних, будуть додані до кластеру ElastiCache / DAX.

Вашій програмі спочатку потрібно буде прийняти рішення перевірити кеш-пам'ять, а якщо вона була порожньою, то запит вашої бази даних для отримання даних. Потім успішний результат буде записаний до магазину ElastiCache / DAX, щоб наступного разу, коли ваша програма спробує це зробити, вона просто отримала результат із кешу.

Також потрібно розглянути варіант архітектури системи пошуку спеціалізованої літератури не як окремого фактичного моноліту, а також як повноцінного мікросервісного додатку. У даному варіанті, кожна функція фільтру пошуку може розглядатися як окрема сервіс-функція, тоді фізично навіть не потрібен постійно працюючий сервіс.

Serverless стає в нагоді, це термін, який зазвичай стосується безсерверних додатків. Безсерверні програми - це ті, які не потребують жодного серверного забезпечення та не вимагають управління серверами.

Усі послідовні функції заміняються окремими AWS Lambda, безсерверними функціями. Тепер система підприємства за посиланням запускає легкий додаток, що тільки чекає результату виконання AWS Lambda. Оплата трохи зросла, адже за кожен виклик Lambda потрібно платити, але тільки за пам’ять, що була використана під час виконання, тому правильно оптимізувавши, кошти будуть мінімальними.

AWS Lambda - це безсерверна обчислювальна послуга, що надається Amazon Web Services (AWS). Користувачі AWS Lambda створюють функції, автономні програми, написані однією з підтримуваних мов та середовищ виконання, та завантажують їх до AWS Lambda, який виконує ці функції ефективно та гнучко.

Функції Lambda можуть виконувати будь-які обчислювальні завдання, починаючи від обслуговування веб-сторінок та обробки потоків даних до виклику API та інтеграції з іншими службами AWS [12].

Поняття "безсерверних" обчислень означає відсутність необхідності підтримувати власні сервери для запуску цих функцій. AWS Lambda - це повністю керована служба, яка піклується про всю інфраструктуру для вас. І тому «безсерверність» не означає, що в них немає задіяних серверів: це просто означає, що про сервери, операційні системи, мережевий рівень та решту інфраструктури вже подбали, так що ви можете зосередитися на написанні код програми.

Загальне порівняння AWS Lambda та AWS EC2 можна побачити на рис. 2.6

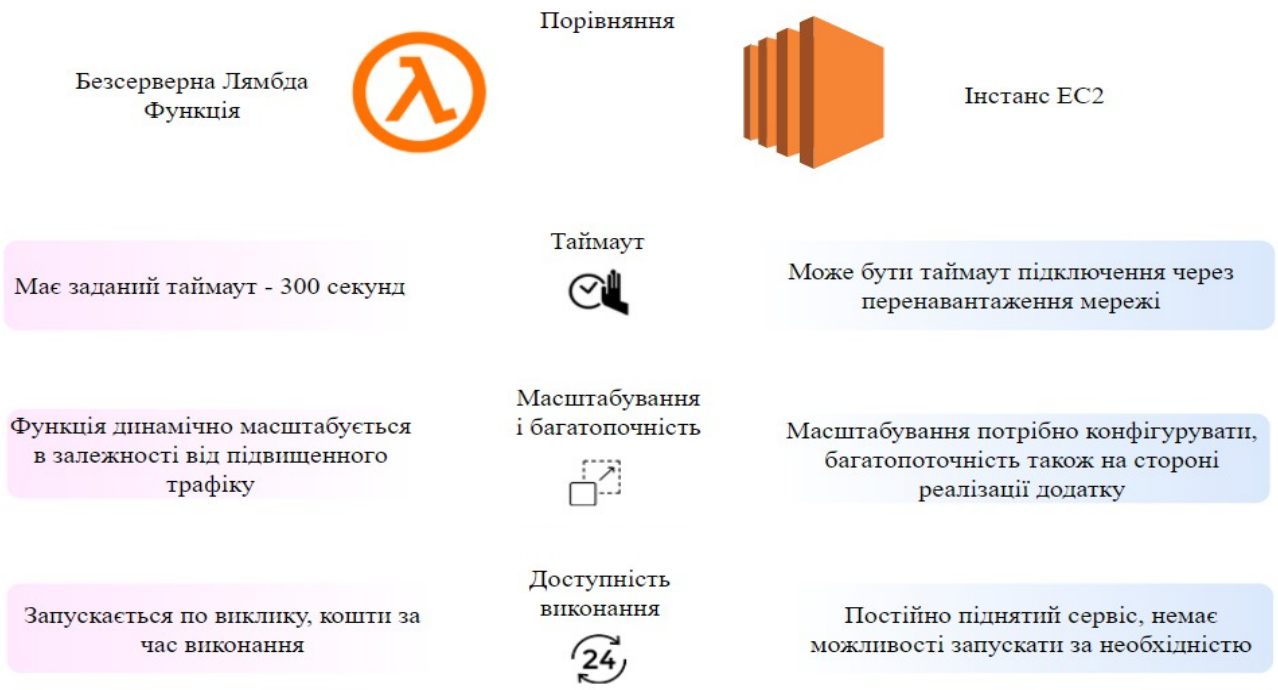

Рисунок 2.6 – порівняння інстансу фізичного сервера з безсерверною функцією

Тобто вся архітектура системи пошуку тепер може складатися з ланки послідовно або паралельно працюючих безсерверних функцій, що можуть також бут написані незалежно, розгортуватися незалежно та не впливати на роботу системи пошуку в цілому.

Проблема цього підходу у тому, що AWS Lambda працює після її виклику, потім вона «виключається», тобто стан її виконання неможливо подивитися та відстежити. AWS Lambda не має контексту, тому при неправильній роботі будь якої з них, буде дуже важко зрозуміти, який був вихід функції та чи були помилки. Чи означає це, що потрібно повернутися до попереднього підходу, коли фізичний сервер працював постійно та мав історію запитів тощо? AWS пропонує вирішення – новітня технологія: Step Functions.

2.5 Використання Amazon Step Functions

Досліджуючи необхідність мікросервісних додатків запускати деякі частини логіки системи послідовно, розбиваючи їх на окремі сервіси, можна дійти до висновку, що найпоширенішим рішенням являється використання синхронного виклику хмарних функцій як сервісів, у випадку AWS – Lambdas. Проходячись по вже існуючим дослідженням було знайдено таку характеристику:

«Багато організацій, які використовують веб-служби Amazon (AWS), спочатку використовували AWS Lambda, продукт Amazon Function-as-a-Service, для автоматизації робочих процесів навколо запущеного та масштабованого коду. Хоча ця послуга заощаджує час для розробників, багато компаній знайшли проблеми, використовуючи AWS Lambda із більш складними програмами».

Незважаючи на все, що цей сервіс робить добре, багато користувачів мають проблеми з AWS Lambda у двох основних сферах: виконання довших робочих місць та управління станом. На фронті виконання AWS Lambda може виконувати функції лише до 15 хвилин. Після цього сервіс закінчується, залишаючи незавершені завдання – незавершеними.

Очевидно, це не працює для організацій зі складними програмами, які покладаються на тривалі роботи. Крім того, користувачі не можуть повернутися назад і повторно виконати код, який не вдається, адже, як було указано раніше, функція, що виконувалася закінчується після своєї роботи і заново викликатися не може, кожен новий запуск – це окремій новий запуск із своїм особистим станом.

Можливо додати контейнери за допомогою такого інструменту, як Amazon Elastic Container Service (ECS), щоб виконати ці довші завдання, але що відбувається, коли контейнери зайняті іншими завданнями? Ви повинні додати стан.

Проблема в тому, що AWS Lambda не створена для ефективного управління станом. Розробники повинні писати код у свої програми для зберігання цього самого стану, що ускладнює управління та збільшує час обробки. Отже,

користувачі змушені вибирати між запуском вимогливих програм та забезпеченням достатнього стану для максимального використання [13].

Що таке AWS Step Functions? AWS Step Functions - це послуга, що надається Amazon Web Services, що спрощує організацію декількох служб AWS для виконання завдань. Step Functions дозволяють створювати кроки в процесі, де результат одного кроку стає входом для іншого кроку, все за допомогою візуального редактора робочого процесу.

Step Functions забезпечують досить зручну функціональність: автоматичну обробку повторних спроб, активацію та відстеження для кожного кроку робочого процесу та забезпечення виконання кроків у правильному порядку. Хоча спочатку цей список може здатися не вражаючим, виявляється, зовсім не тривіально забезпечувати, щоб усі ці речі відбувалися правильно в робочих процесах, що містять десятки кроків і сотні паралельних виконань. Step Functions виконує багато роботи, яку раніше потрібно було виконати у вашій програмі [14].

Під капотом Step Functions є автоматом стану, і його основні абстракції називаються станами. Конфігурація функцій кроку являє собою карту всіх можливих кроків та переходів між ними. Стани та їх переходи визначаються з використанням мови штатів Амазонки. Ця мова базується на JSON і є власністю Amazon.

Чому крокові функції AWS є невід’ємною частиною безсерверної екосистеми? Створюючи безсерверні додатки, розробники, як правило, хочуть масштабувати програму відповідно до зростаючого навантаження, зберігаючи при цьому низькі витрати і дозволяючи декільком командам одночасно працювати над різними частинами програми. У безсерверній моделі однією з найкращих практик для досягнення цих цілей є розділення бізнес-логіки програми на набір нерозв’язаних служб. Велика безсерверна програма може складатися з десятків або сотень безсерверних сервісів.

Проблема виникає, коли така велика кількість служб потребує доступу до різних частин спільного стану. Для ефективної роботи цих служб команди також повинні мати можливість організувати потік даних через усі сервіси додатків в

одному місці, і саме цим займаються AWS Step Functions. Вони стали найважливішим елементом безсерверної екосистеми впродовж усього стану системи і управління даними, необхідними для ефективної роботи безсерверних систем у масштабі.

Step Functions полегшують життя безсерверного розробника, дозволяючи швидко створювати складні послідовності завдань в AWS, одночасно беручи на себе функцію обробки помилок і повторної спроби виконання логіки та дозволяючи відокремити бізнес-логіку вашого додатка від логіки оркестрації. Нижче ми розглянемо особливості того, як ця технологія може допомогти у дизайні та розробці системи [15].

Надається можливість швидкого створювання складних послідовностей завдань. Організація послідовності з десяти окремих безсерверних додатків, управління їх повторними спробами та налагодження будь-яких помилок може бути надзвичайно складним завданням. Коли ви додаєте ще більше функцій, складність управління ними зростає в геометричній прогресії.

Завдяки графічному інтерфейсу та вбудованому оперативному контролю, Step Functions управляє послідовністю завдань для вас і знімає великий оперативний тягар з вашої команди.

Керування станом між виконанням різних безстанових функцій. Для багатьох безсерверних робочих процесів налаштування черг та баз даних для зв'язку між усіма безсерверними службами може зайняти багато часу та спричинити помилки, не кажучи вже про те, що не всі випадки використання вимагають реальної бази даних або реальна система черг.

Step Functions дозволяють легко налаштувати управління станом на ранніх етапах, і це продовжує працювати добре з приростом масштабу завдань, коли використовуються багато сервісів разом.

Із використанням цієї технології логіка робочого процесу додатків стає від’ємною від бізнес-логіки. Тут ми маємо ще одну гарну практику безсерверної розробки. Додавання логіки робочого процесу та складних серверних конфігурацій до додатків, які повинні обробляти лише бізнес-логіку, збільшує

складність програм і може легко створювати проблеми. Крім того, управління станом окремо від бізнес-логіки дозволяє розробникам зберігати ясність при роботі в безсерверній системі.

Також до плюсів можна віднести ефективніші робочі процеси з паралельним виконанням. Коли ви самостійно керуєте станом різних функцій, отримати високу продуктивність від системи може бути складно. Деякі оркестровані фрагменти можуть обробляти лише одне завдання за раз, що може сповільнити роботу всієї програми.

Step Functions дозволяють одночасно виконувати багато паралельних робочих процесів, переконуючись, що продуктивність масштабується залежно від завантаження програми.

Усі вищезазначені переваги можуть забезпечити високу цінність для багатьох команд та випадків використання, але перед тим, як взяти участь у запуску функцій Step у виробництві, слід також врахувати потенційні недоліки використання цієї програми у вашому додатку. Хоча AWS Step Functions вирішують багато проблем для безсерверної спільноти, перехід усіх ваших шарів оркестрації на Step Functions не обов'язково є правильним рішенням для вас. Ось декілька речей, про які слід пам’ятати перед тим, як почати використовувати покрокові функції у виробництві.

Конфігурація з мовою станів Amazon є досить складною; його синтаксис заснований на JSON і тому оптимізований для машинної читабельності замість читабельності людьми. Вивчення мови може бути складним завданням, і, оскільки воно є власністю Amazon, навички, які ви набуваєте, застосовуватимуться не де-небудь, крім контексту AWS Step Functions [16].

Ця мова є власністю Amazon і може використовуватися лише на AWS. Тому, якщо ви вирішите перейти до іншого постачальника хмарних послуг, вам доведеться повторно впровадити рівень оркестровки або замінити його альтернативою іншого постачальника.

Ось кілька випадків використання, коли функції кроку AWS можуть підвищити продуктивність вашої команди.

Обробка даних. Інші види обробки даних також добре підходять для функцій кроку AWS, особливо якщо джерело та адреса знаходяться всередині AWS. Функції кроку спрощують взаємозв’язок декількох кроків обробки даних, включаючи ті, що зроблені за допомогою функцій AWS Lambda, простий служби черги та простий служби сповіщень.

Організація робочого процесу без серверів. Основна цінність крокових функцій AWS полягає в простому організуванні програм, які вимагають взаємозв'язку безлічі безсерверних функцій. Якщо у вас є бізнес-процеси, для отримання кінцевого результату яких потрібна комбінація декількох роз’єднаних безсерверних програм, функція Step може бути правильним вибором для зручності організації [17].

Безкоштовний рівень AWS включає 4000 переходів стану функцій кроку AWS на місяць. Ця частина безкоштовного рівня не закінчується, тому ви можете скористатися нею, навіть якщо ваш обліковий запис AWS не є абсолютно новим.

Окрім безкоштовного рівня, Step Functions коштує 0,025 дол. США за 1000 переходів стану.

Можливо, важко уявити, що саме це буде означати для вашого щомісячного рахунку AWS, тому пропонується розібрати кілька прикладів сценаріїв ціноутворення нижче.

Припустимо, що у вас є робочий процес, який перетворює зображення для вашого веб-сайту у чотири різних розміри, причому робочий процес складається із десяти переходів стану в середньому для кожного зображення і близько 10% виконання робочого циклу передбачає одну повторну спробу. Припустимо, що ви завантажуєте 100 000 зображень на місяць.

У цьому випадку вартість функцій кроку AWS буде такою:

(10 переходів x 100 000 виконання + 1 повторна спроба x (100 000 x 10% виконань)) x 0,025 дол. США за 1000 переходів стану = 25,25 дол. США.

Майте на увазі, що плата за AWS Step Functions доповнює будь-які збори за передачу даних та вартість будь-яких послуг AWS, якими ви користуєтесь у робочих процесах. Наприклад, якщо ці 100 000 виконання робочого процесу

призводять до 600 доларів на місяць плати за AWS Lambda плюс 100 доларів на місяць до плати за передачу даних, загальна вартість складе:

600 доларів США за плату AWS Lambda + 100 доларів США за плату за передачу даних + 25,25 доларів США за плату за крокові функції = 725,25 доларів США на місяць для всієї системи.

Пропонується розглянути знову основну проблему, але тепер із усіма перевагами Step Functions та визначити, чи потрібно все ж таки використовувати цю технологію.

По-перше, технологія мікросервісів передбачає собою, що кожен сервіс – це незалежна частина системи, яка може бути навіть написана іншою мовою програмування, тому коли потрібне послідовне виконання декількох функцій як сервісів, у випадку AWS – Lambdas, то впровадження Step Functions – необхідний крок для завдання порядку виконання, залишаючи сервіси незалежними. Без цієї технології доведеться на кожному кроку, у кожній функції задавати ім’я наступної, тобто сервіси будуть знати один про одний, що означає – тісне зв’язування і порушення концепту мікросервісної архітектури.

По-друге, як один з недоліків Step Functions було визначено, що їх доволі складно конфігурувати, тому що потрібно знати нову мову AWS – Amazon State Language. Але, уявімо конфігурацію функцій, що послідовно викликаються та знають одна про одну, як вже було сказано. Для того, щоб виключити один з кроків або просто змінити, щоб після 3го кроку, наприклад, викликався 6й, а потім 4й? Як вже було описано, функція, викликаючи наступну у своєму ж коді, тісно зв’язана із цією самою наступною. Тому зміна порядку викликів або виключення одного з етапів виконання призведе до зміни коду майже кожної функції окремо у той час, коли один раз зрозумівши ASL та конфігурувавши Step Functions, можна з легкістю маніпулювати порядком и так далі, не торкаючись окремих функцій.

По-третє, AWS Step Functions вирішують питання контексту між усіма послідовними функціями, що безперечно є їхнім плюсом, адже концепт Lambda як раз говорить, що це функція безконтекстна та швидка, тому вигадування логіки

передачі його без Step Functions – це суперечить самому принципу беззсерверних функцій та робить логіку виконання та підтримки більш складнішою та повільнішою. Як говорить один з принципів розробки SOLID, літера S – Single Responsibility, що означає «єдиний обов'язок». Тобто якщо лямбди не вміють працювати з контекстом і за своїм принципом не повинні – не треба їх змушувати, нехай про це подбає той сервіс, що був створений для цього – AWS Step Functions.

І нарешті, ціноутворення. Як вже було сказано вище, Step Functions не дуже дорогий сервіс. Якщо компанія дозволяє собі систему побудовану на мікросервісній архітектурі і що використовує AWS, то у випадку послідовного виконання функцій, необхідно замислитись над використанням технології автомату стану Амазона. Адже, навіть якщо там сто або більше функцій виконується послідовно, коли вони сконфігуровані у Step Functions – це все одно лише одне виконання автомату, а знову, як було сказано вище, за тисячу таких виконань Amazon Web Services вимагатимуть лише 0,025 дол. США [18].

3 БУДУВАННЯ АРХІТЕКТУРИ СИСТЕМИ ПОШУКУ

3.1 Приклад бази даних підприємства із спеціалізованою літературою

Як приклад системи, що має електронні ресурси інформації, такі як наукові статті, методичні вказівки та інше, що не потребують публічного розповсюдження, був взятий університет.

База даних для системи керування електронної бібліотеки для, наприклад, університету, розроблена за допомогою MySQL Workbench під керуванням системи управління бази даних MySQL. Схема наведена на рисунках 3.1 та 3.2.

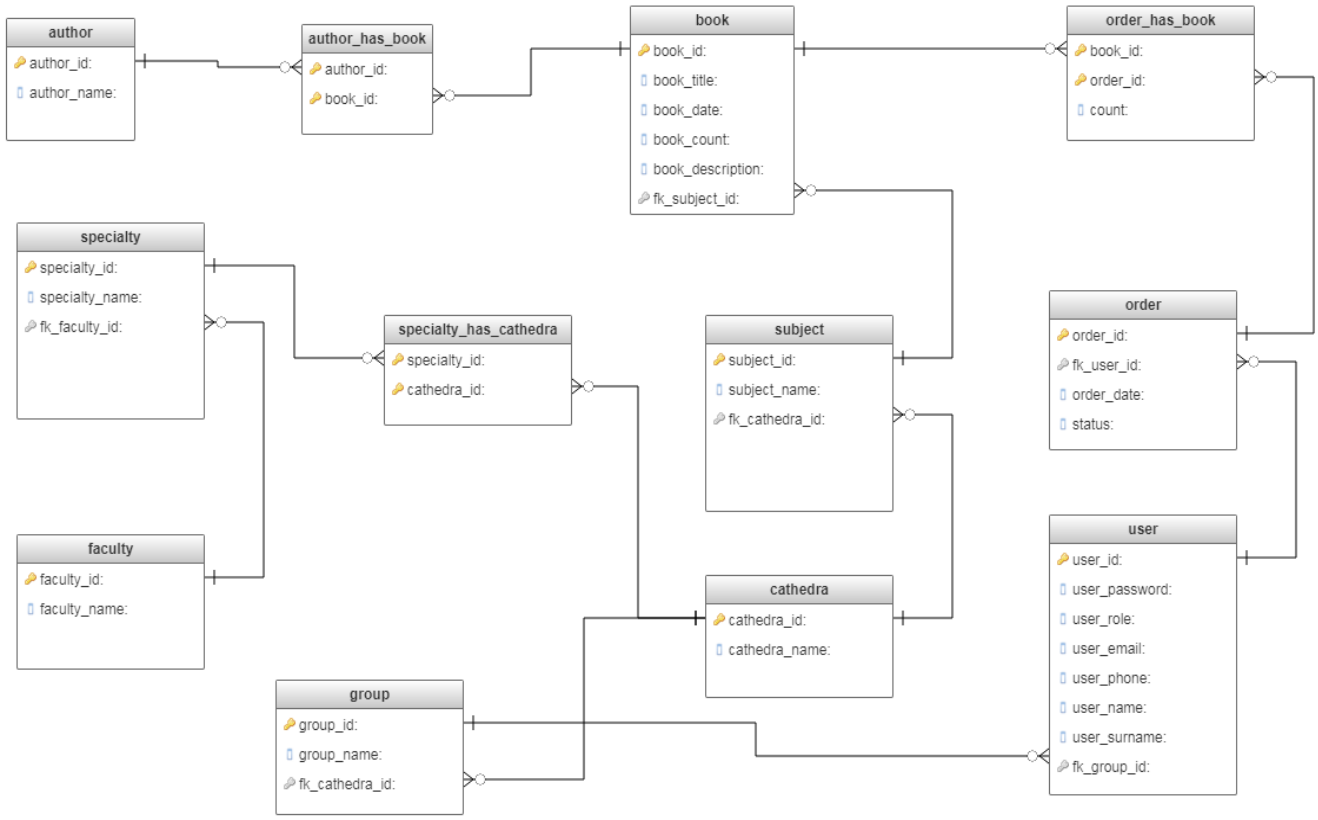

Рисунок 3.1 – Логічна схема бази даних розроблюваної системи

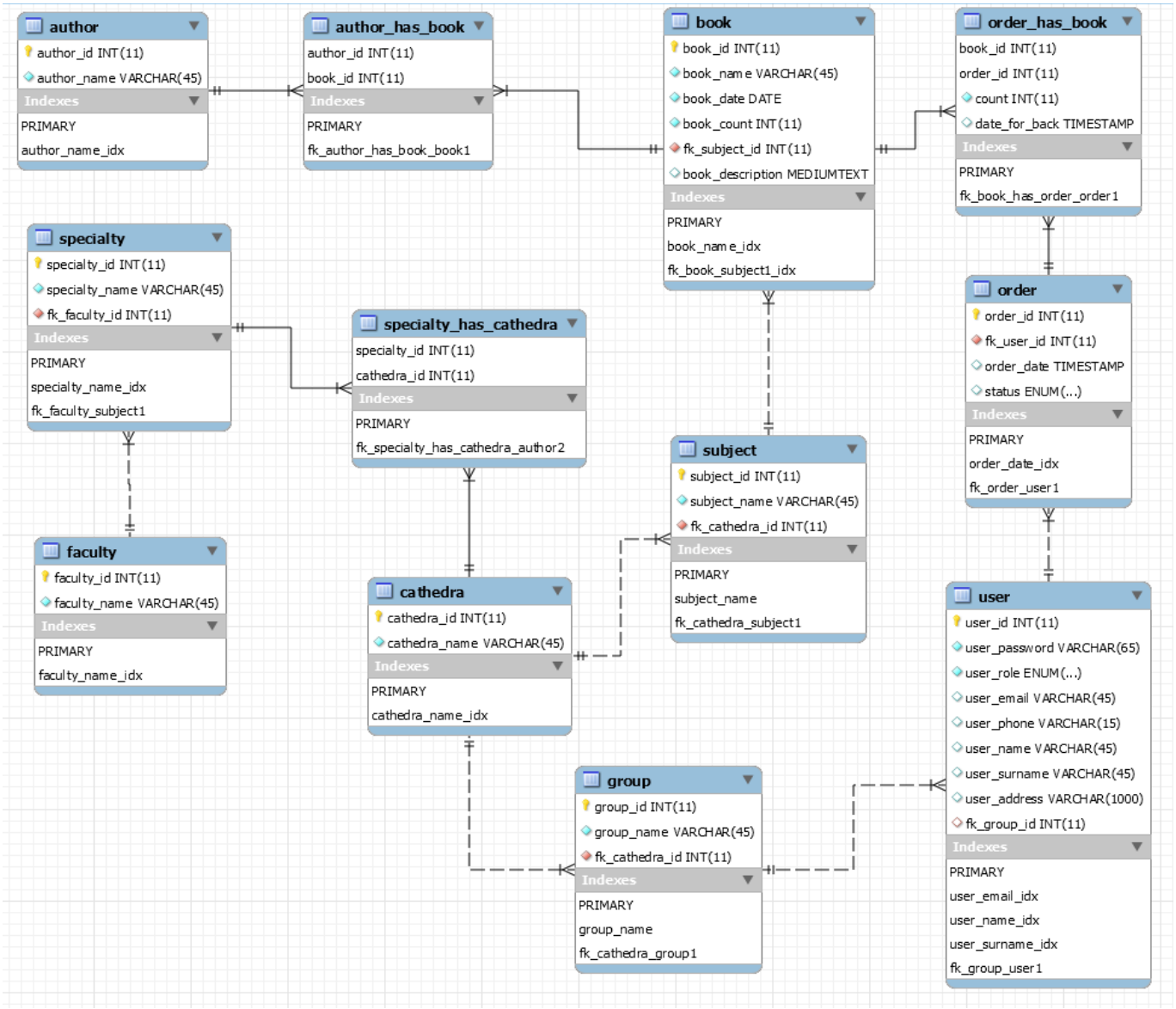

Рисунок 3.2 – Фізична схема бази даних MySQL розроблюваної системи

Але для алгоритму пошуку літератури декілька таблиць з їхніми відношеннями не потрібні зовсім, наприклад, формування замовлення та інше. Якщо виділити потрібні таблиці, за якими дійсно може і буде фільтруватися будь-яке літературне джерело, то ми отримаємо список, що складається з факультету, спеціалізації, кафедри та предмету (рис. 3.3).

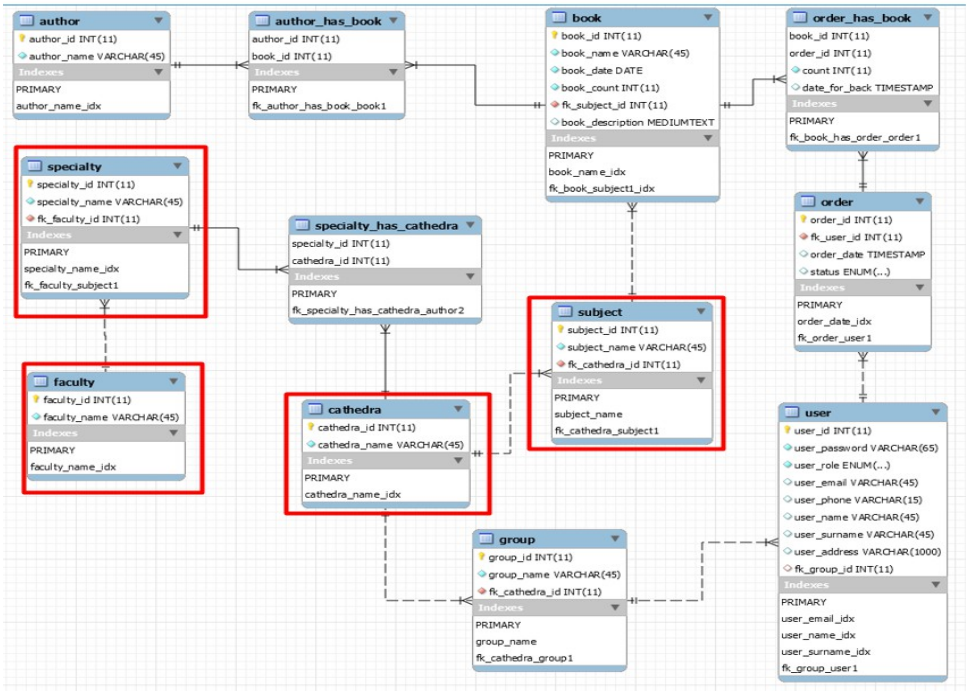

Рисунок 3.3 – необхідні дані для пошуку літератури користувачеві

У цьому випадку може стати в нагоді механізм реплікації, тобто створюватиметься окрема база даних, що включатиме тільки необхідні таблиці та поля для фільтрації книжок. Це може бути як SQL так і NoSQL база даних.

Для зображення будування архітектури системи пошуку літератури пропонується створити просту NoSQL базу даних, що включатиме таблицю з книжками, що буде складатися з необхідних для фільтрації характеристик (рис. 3.4)

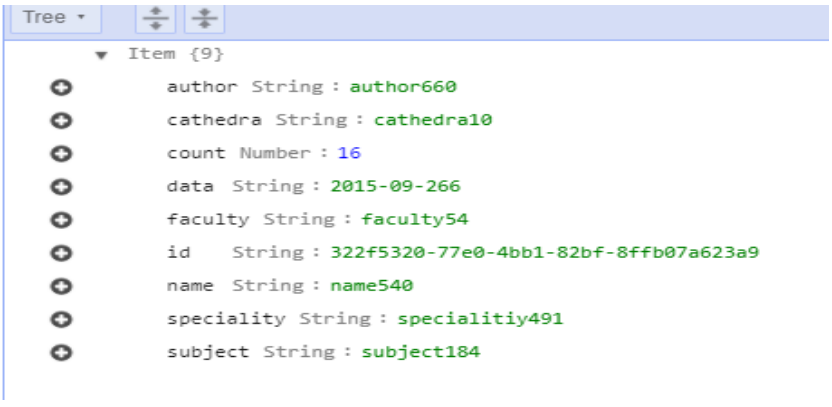

Рисунок 3.4 – структура таблиці книжок для фільтрації

3.2 Будування архітектури системи пошуку як окремого цілого сервісу

Почнемо будувати архітектуру. Насамперед, для того, щоб побудувати окремий сервіс пошуку спеціалізованої літератури, треба знати та розуміти мову програмування та технології, що будуть використовуватися.

Для даної задачі була обрана серверна мова програмування Java з версією 1.8 та вище та фреймоврк Spring, включаючи Spring Boot, Spring MVC та Spring Data. Середовище розробки – IntelliJ IDEA Community Edition (безкоштовна).

Перш за все, за принципами архітектури MVC нам потрібно створити такий додаток, щоб в ньому був шар контролера, завдяки якому в нас буде доступ до додатку, шар логіки, що буде фільтрувати книги за пошуком та шар доступу даних, котрий звертатиметься до бази даних підприємства, щоб взяти ці книги.

Слід також пам'ятати, що нам потрібен «збірник» проекту, тому що механізм розгортки нашого додатку на сторонньому сервісі повинен бути з одного файлу (наприклад, .jar), щоб уникнути ручного встановлення усіх залежностей та необхідних бібліотек, щоразу, коли оновлятиметься версія додатка.

Загальна архітектура взаємодії системи підприємства, вище зазначеного, як приклад, системи університету ХНУРЕ, буде виглядати наступнім чином, тобто система пошуку спеціалізованої літератури виділена як повністю окремий додаток, що має свій життєвий цикл та не має ніяких залежностей від системи ХНУРЕ та навпаки. Даний варіант вже приміняє парадигму мікросервісного підходу, коли кожен з сервісів може бути написаний своєю мовою, мати свою стратегію розготки та підтримуватися різними командами, не створюючи при цьому незручностей для будь-якої з систем. Можливість використання сервісу пошуку бути надаватися за допомогою API, яке виставе система для зовнішнього доступу. Контракт за яким система ХНУРЕ буде звертатися до системи пошуку унікальний, тобто для кожного підприємства оговорюватиметься окремий контракт інтегрування.(рис. 3.5).

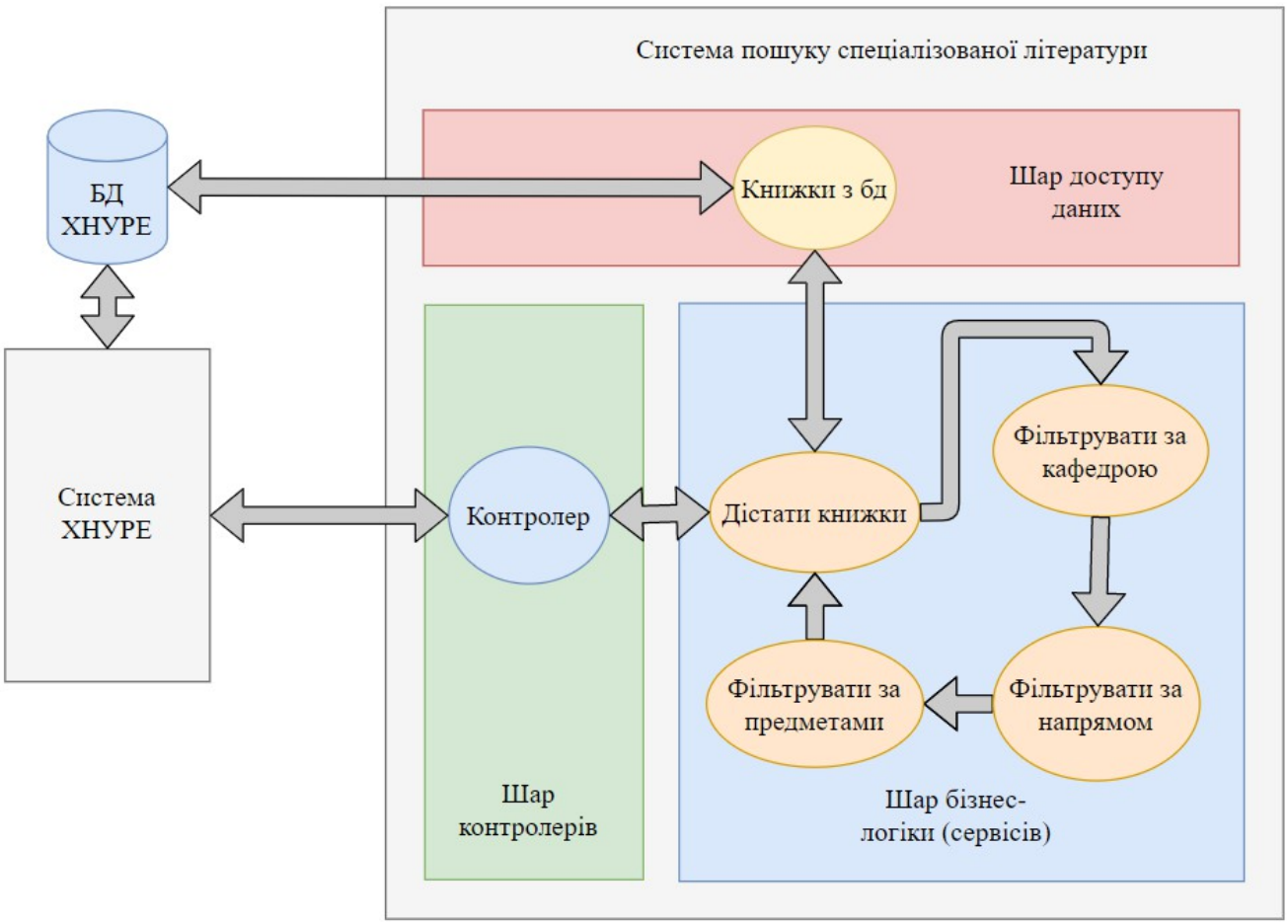

Рисунок 3.5 – Інтеграція системи ХНУРЕ з окремим мікросервісом підсистеми пошуку у вигляді однієї суцільно системи

Створювати таку систему ми будемо за допомогою збірника проектів Maven, найпоширенішого інструменту управління збіркою для Java. Впорядкована модель конфігурації на основі XML від Maven дозволяє розробникам швидко описувати або розуміти обриси будь-якого проекту на основі Java, що робить запуск та спільний доступ до нових проектів швидким. Була обрана конфігурація збирання проекту у .jar формат.

Переходячи від імплементації самого сервісу до його розгортки та інфраструктури, треба нагадати, що обраний хмарний провайдер, Amazon Web Services, надає віртуальні сервери для роботи таких видів додатків, ці сервери – EC2.

Щоб використовувати будь-який сервіс Амазону, потрібен його акаунт, тому у якості передумови на початку цієї роботи було створено обліковий запис AWS.

Після успішного входу до Амазону, треба обрати потрібний для нашої роботи – EC2. Після чого на екрані з'явиться можливість створити новий EC2 Instance. Оберемо Linux сервер, що матиме можливість у подальшому використовувати безкоштовну конфігурацію. Linux у даному випадку є більш переважним, аніж Windows, тому що сама система важить менше та потребує менших затрат оперативної пам'яті (рис. 3.6).

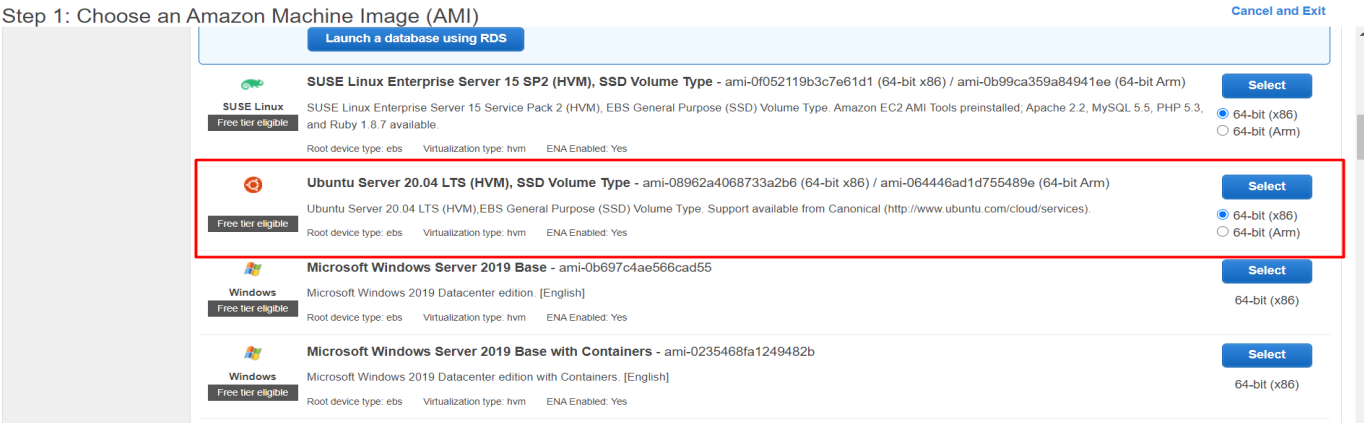

Рисунок 3.6 – обирання віртуального серверу

Під час створення віртуального серверу, треба розуміти, що в залежності від об'єму пам'яті, ОЗУ та інших характеристик, ціна підтримання роботи такого серверу буде збільшуватись, тому у даній роботі був обраний інстанс із повністю безкоштовним утриманням на деякий час (рис. 3.7).

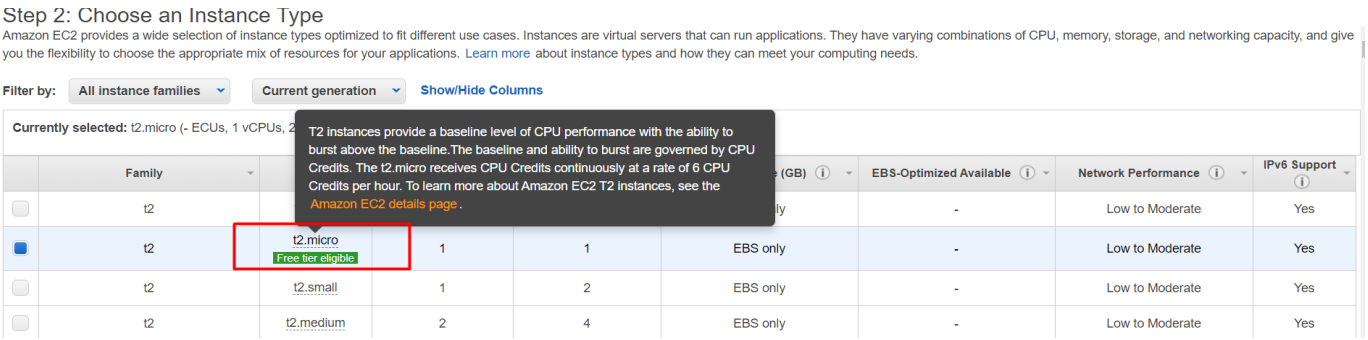

Рисунок 3.7 – обирання конфігурації сервісу

Після обирання конфігурації сервер запускається, побачимо на екрані, що він був запущений без помилок (рис. 3.8)

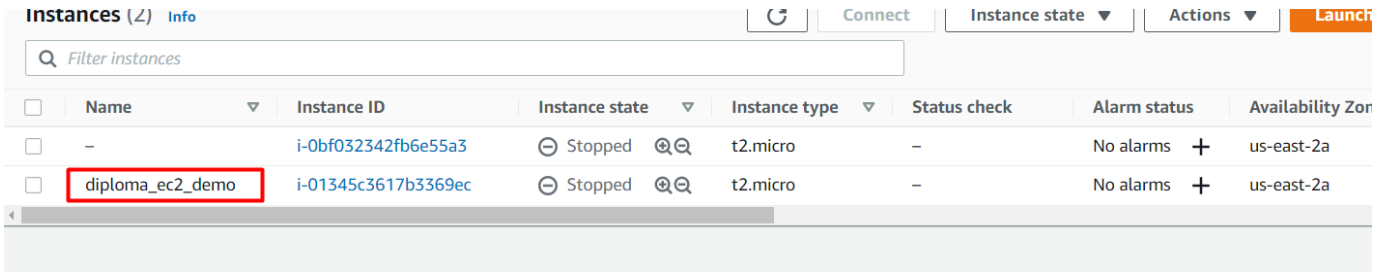

Рисунок 3.8 – запущений віртуальний сервер

Маючи вже віртуальний хмарний сервер та розроблений додаток, потрібно визначитися, якою уде стратегія розгортки системи пошуку спеціалізованої літератури. Наразі мається файл .jar, який може знаходитися на локальній машині будь-якого з розробників додатку, та сервер, котрий запущений у якомусь регіоні Амазона.

Для того, щоб запустити наш файл на сервері, потрібно спочатку підключитися до нього. Для цього потрібно дозволити підключення до серверу за допомогою протоколу SSH, наприклад, з будь-якого IP-адресу. Також, розуміючи, що додаток пошуку спеціалізованої літератури – це, перш за все, веб-додаток, котрий виставляє API для доступу за протоколом HTTP (HTTPS), потрібно також дозволити нашому серверу бути доступним через цей самий HTTP (HTTPS) також з будь-якого місця знаходження. Це все ми можемо зробити через Amazon Security Group, яка діє як віртуальний брандмауер для ваших екземплярів EC2 для контролю вхідного та вихідного трафіку. Вхідні правила контролюють вхідний трафік до вашого екземпляра, а вихідні правила контролюють вихідний трафік з вашого екземпляра. Ви можете будь-коли змінити правила для Security Group. Нові та змінені правила автоматично застосовуються до всіх екземплярів, пов'язаних із групою безпеки (рис. 3.9).

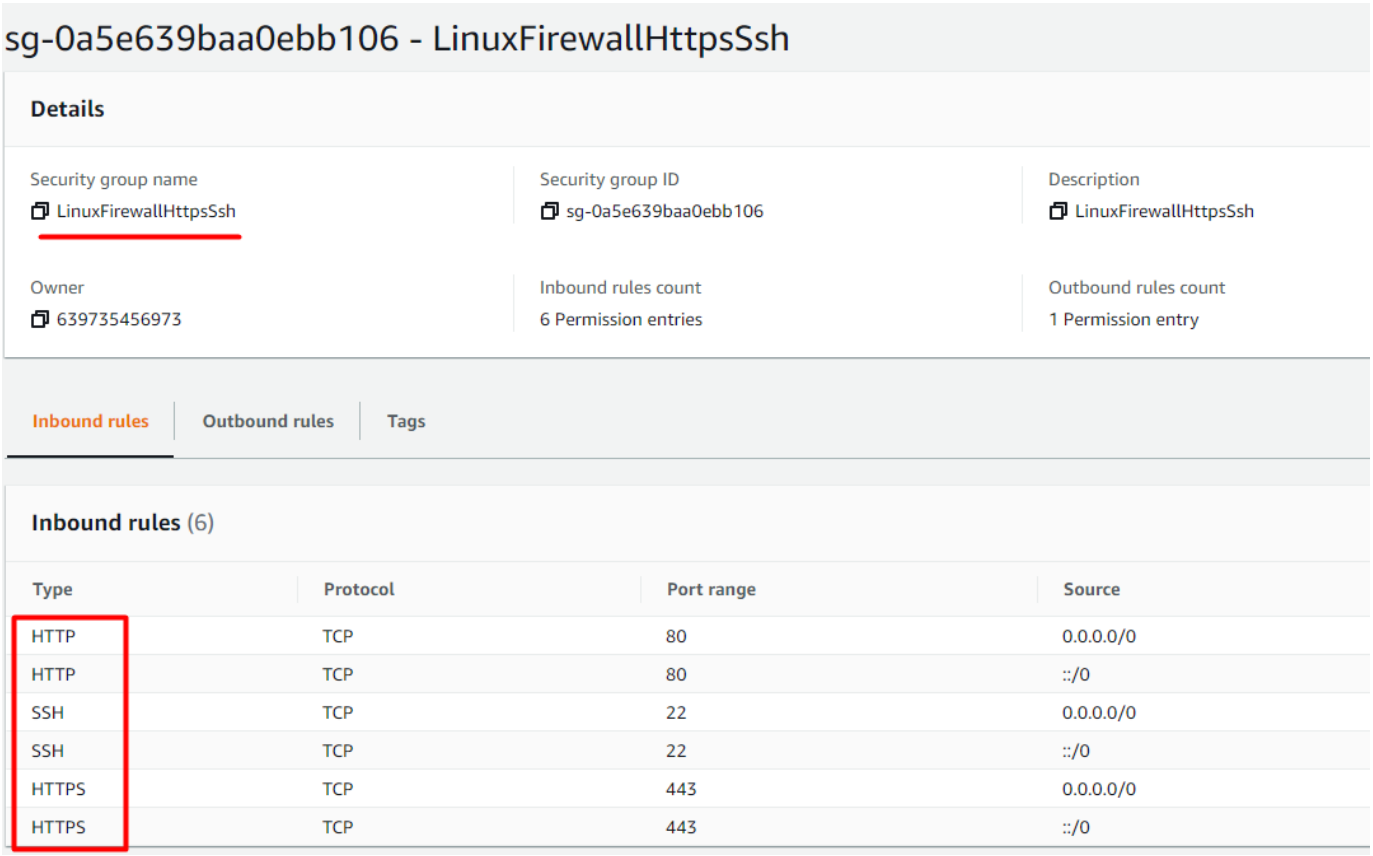

Рисунок 3.9 – Відкриті порти захищеної групи віртуального серверу

Надавши доступ на підключення до сервера, потрібно підключитися за допомогою SSH-клієнта для того, щоб завантажити та запустити додаток пошука спеціалізованої літератури. Для цього використовується такий клієнт як MobaXterm, це набір інструментів для віддалених обчислень. В одній програмі він надає безліч функцій, призначених для програмістів, веб-майстрів, IT-адміністраторів та майже всіх користувачів, яким потрібно більш просто обробляти свої віддалені завдання.

Створимо нову сесію та оберемо підключення через SSH за допомогою .pem ключів, що згенерував Амазон на етапі створення серверу, впишемо віддалений хост, взявши його зі вкладки із запущеним ec2-інстансом та звернувшись до позначення Public IP та впишемо ім’я користувача (за замовчуванням у віртуальних серверів Амазона на платформі Linux, ім’я користувача виглядає як «ec2-user», а у системі Ubuntu – «ubuntu») (рис. 3.10).

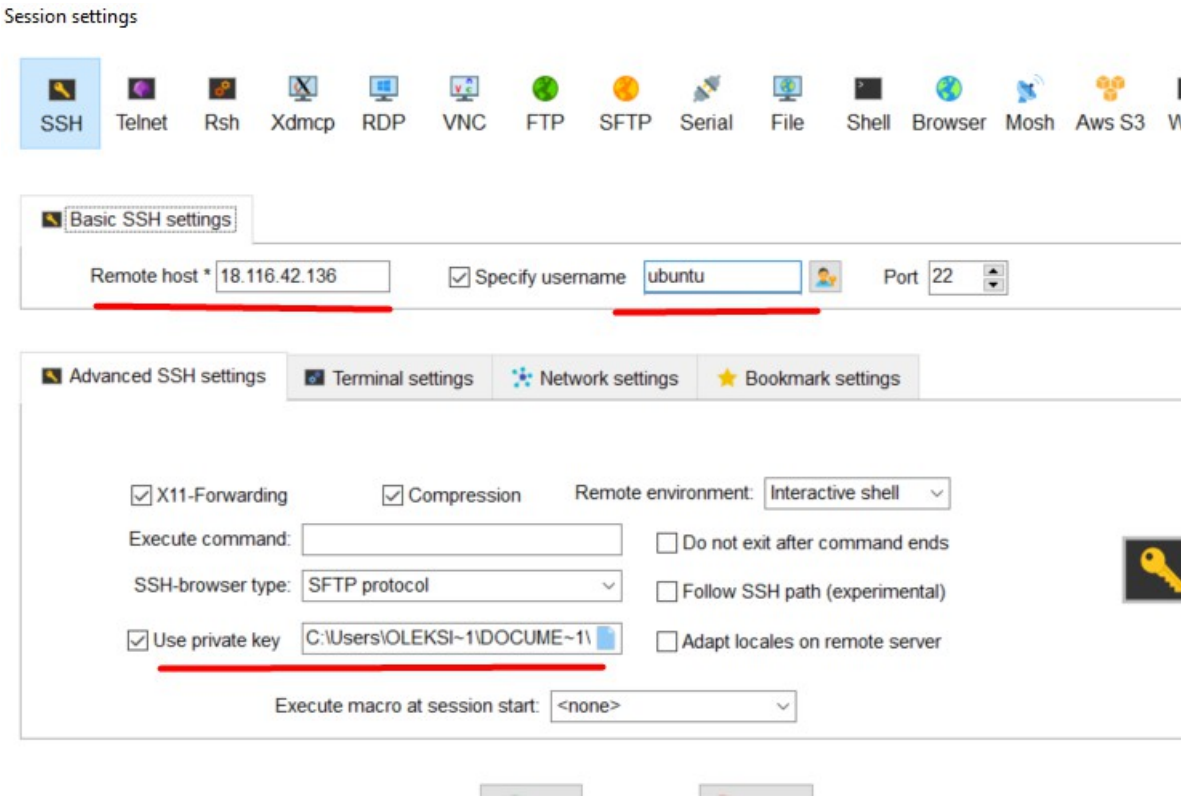

Рисунок 3.10 – підключення до віддаленого віртуального серверу

Після успішного підключення потрібно визначити процес, за яким розробник може завантажити .jar файл додатку пошуку спеціалізованої літератури до віртуального серверу. Пропонується варіант, що користувач завантажитиме файл до сервісу Амазона – Simple Storage Service (S3). Це сервіс, в якому необмежена кількість пам’яті може використовуватися також безкоштовно, потрібно лише створити базову директорію, де будуть зберігатися файли. Такі директорії зазвичай звуться «бакетами».

Отже, створивши такий бакет та завантаживши туди файл розгортки системи пошуку, можна знову звернутися до віддаленого серверу та скачати цей файл за допомогою команди терміналу AWS. Після цього запустити наш додаток і він повинен бути доступний. Відобразимо цей процес на діаграмі послідовності (рис. 3.11).

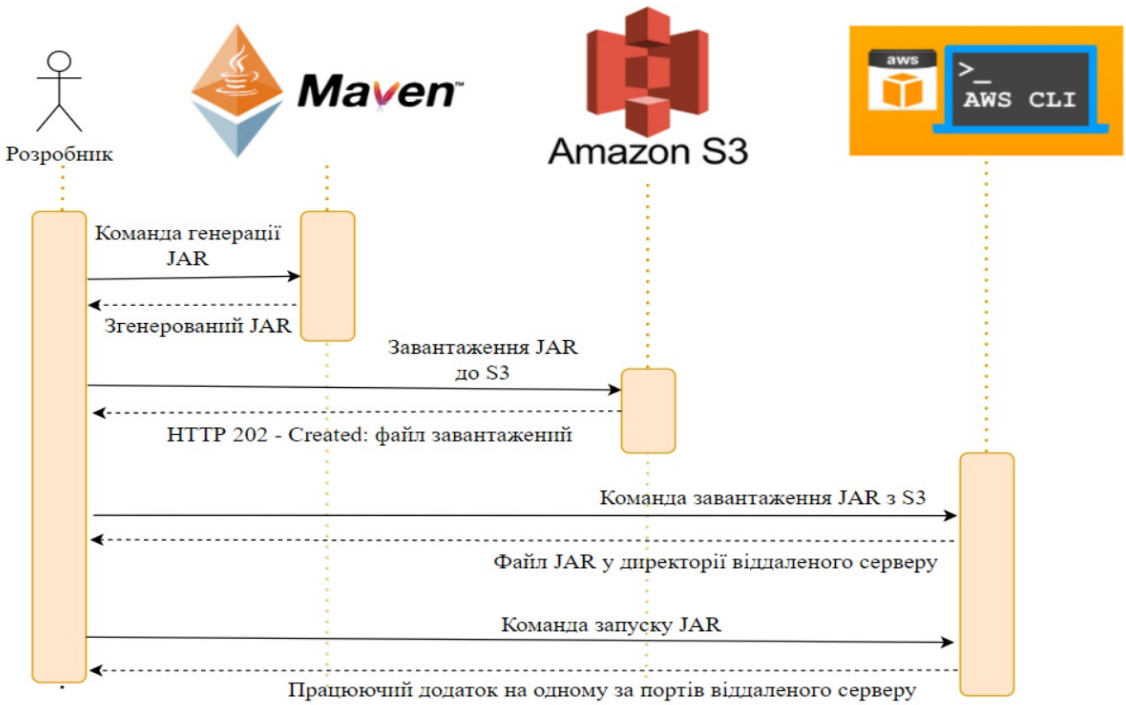

Рисунок 3.11 – діаграма послідовності розгортки та запуску системи пошуку на віртуальному сервері

Далі потрібно перевірити на працездатність. У першу чергу завантажимо файл до S3-бакету (рис. 3.12)

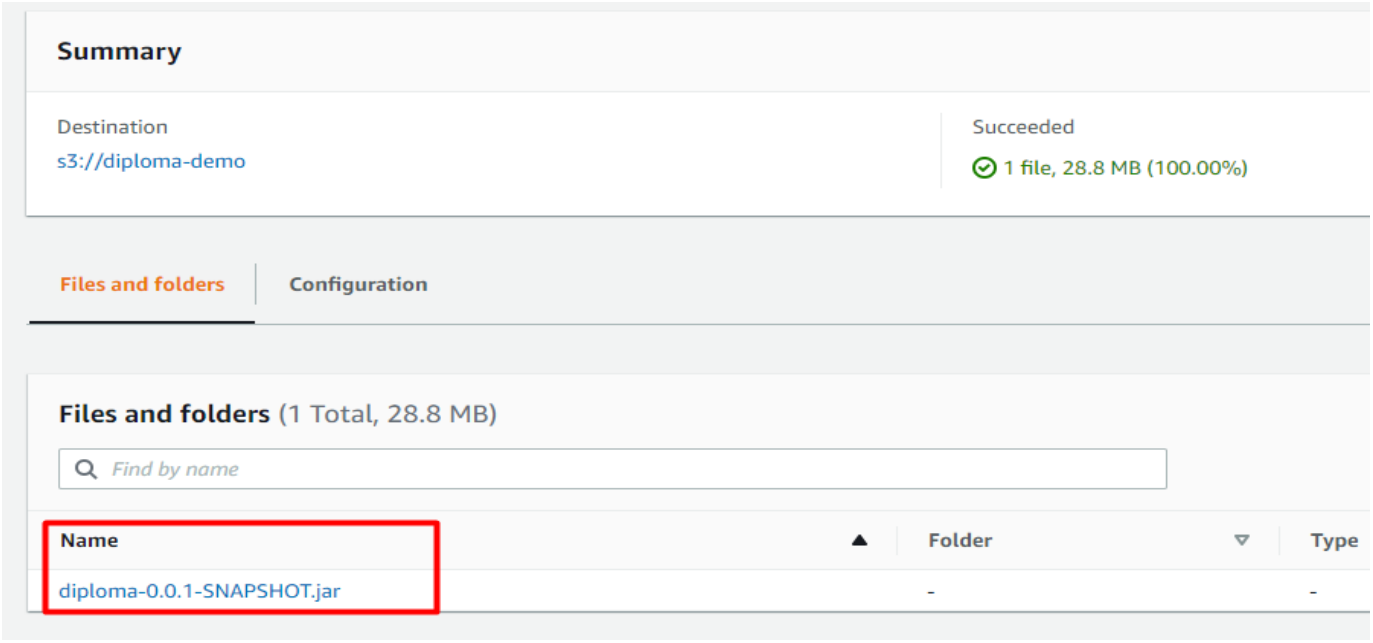

Рисунок 3.12 – файл для розгортування у S3-бакеті

Завантажимо цей файл на сервер та виконаємо запуск, переконаємося, що запустився сервер розгортування на потрібному порті (рис 3.13).

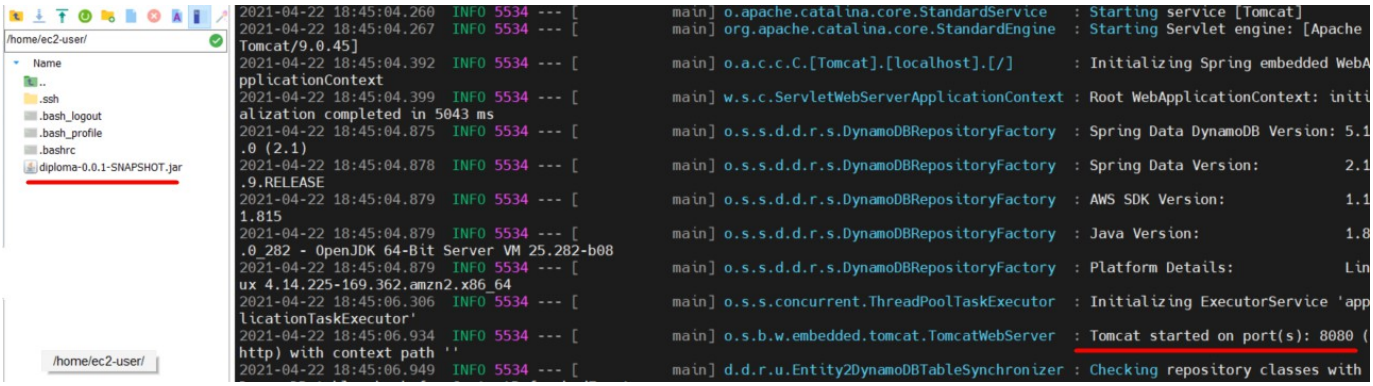

Рисунок 3.13 – запущений додаток на порті 8080

Наостанок перевіримо, що результат виконання серверу доступний за HTTP запитом, що означатиме, що система підприємства, наприклад університету ХНУРЕ може інтегруватися з системою пошуку спеціалізованої літератури за HTTP протоколом. Для цього використаємо програму відкритого доступу – Postman. Postman - це платформа для співпраці для розробки API. Функції листоноші спрощують кожен крок побудови API та впорядковують співпрацю, щоб ви могли створювати кращі API - швидше.

Виконаємо простий запит, що дістає усі книги доступні у підприємстві (для прикладу поки що була додана лише одна книга) (рис 3.14).

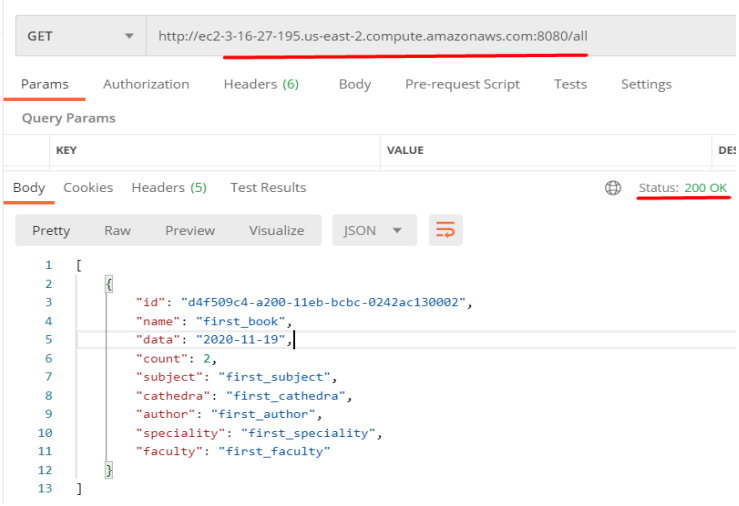

Рисунок 3.14 – працюючий віртуальний сервер

3.3 Будування архітектури системи пошуку як ланки окремих сервісів

Як вже було визначено раніше, не обов’язково тримати всю систему пошуку спеціалізованої літератури як цілу систему з усіма функціями-фільтрами, що триматиметься як 1 файл для розготки та буде змінюватись та оновлюватися кожного разу, як зміниться якийсь один конкретний фільтр.

Вже було зазначено, що така архітектура має незручність через те, що неправильна робота однієї з функцій фільтрування книжок за пошуком може впливати на роботу усієї системи пошуку спеціалізованої літератури загалом. Як вирішення – можна розбити кожен фільтр на окремий сервіс, але непотрібно для кожного створювати віртуальний сервер.

У випадку створення віртуального серверу для кожної з функцій в першу чергу зменшить продуктивність та швидкість виконання запиту, адже наразі система підприємства інтегруватиметься з системою пошуку за HTTP протоколом, що вже займає час, а далі, якщо в середні буде 10 різних фільтрів, тоді буде ще 10 HTTP з’єднань, не говорячи про те, що кожен сервер потребуватиме розгортування цілого додатку на своєму інстансі.

Тому тут більше підійде парадигма функціонального програмування, де кожна функція не залежить одна від одної та має лише дані на вхід та вихідні дані. Для цього була створена технологія Function as a Service, що у Амазоні реалізується за допомогою AWS Lambda функцій.

Побудуємо архітектуру системи пошуку спеціалізованої літератури, як вона виглядала би для вже описаного випадку інтегрування з прикладом системи ХНУРЕ (рис. 3.15)

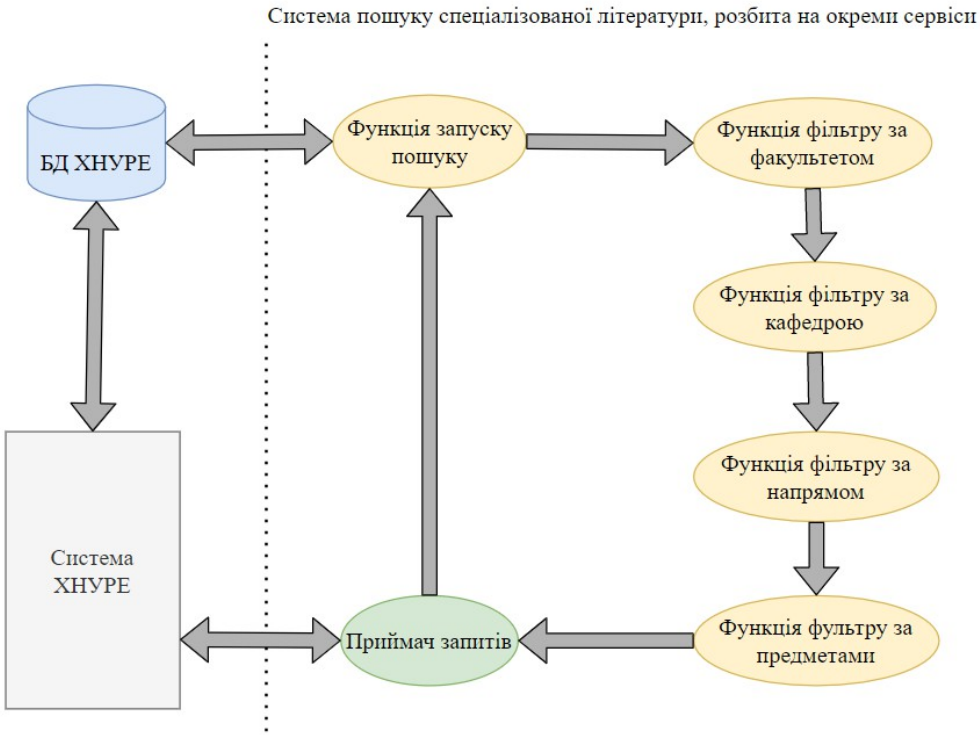

Рисунок 3.15 – мікросервісна архітектура системи пошуку за окремими функціями

Побудуємо стільки лямбда-функцій, скільки зараз в наявності фільтрів.

Для того, щоб розпочати розробку, потрібно визначитись, якою мовою програмування будуть реалізовані лямбди. Амазон підтримує декілька мов, включаючи Java, Python та NodeJS. Розглядаючи кожну з них, потрібно пам’ятати, що лямбда-функції повністю ініціалізуються з початком роботи та повністю виключаються після її завершення. А ціна за виконання залежить від швидкості та витраченої пам’яті цього сервісу. Тому використання Java, котра тягне за собою кожного разу встановлення віртуальної машини, усіх залежностей, роботи компілятора, а вже потім інтерпритатора, може уповільнити функцію, а значить зробити її ще й дорожчою. Це зовсім непотрібно, тому для наглядності будуть використовуватись мови Python та NodeJS.

Побудуємо діаграму послідовності для цієї реалізації системи (рис. 3.16)

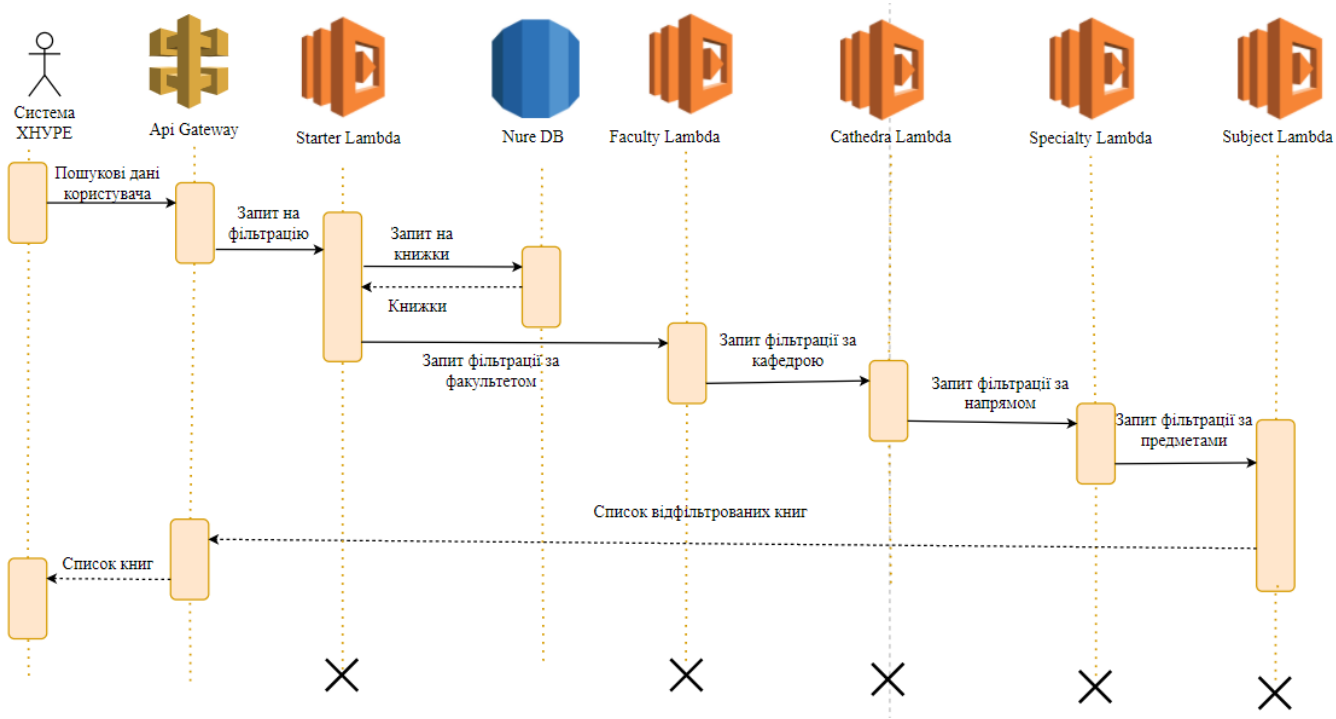

Рисунок 3.16 – діаграма послідовності мікросервісної архітектури системи пошуку, розроблена окремими функціями

За своєю реалізацією, лямбда-функція недоступна ззовні за будь-яким відкритим протоколом, наприклад HTTP. Тому, щоб мати можливість відправляти запити, треба розробити шлюз, котрий прийматиме запити та транслюватиме їх до функції запуску ланцюга функцій (рис. 3.17)

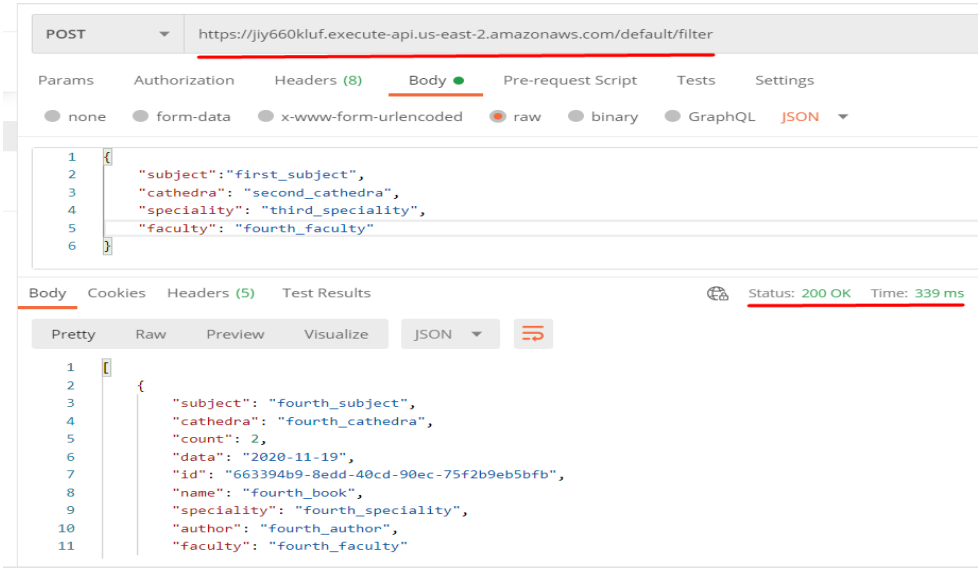

Рисунок 3.17 – працюючий ланцюжок безсерверних функцій

3.4 Будування архітектури системи пошуку як автомату стану

Створивши ланцюг з послідовно працюючих функцій, стає ясно, що такий варіант архітектури може бути масштабований, але також кожна функція тісно зв’язана з наступною. З цього випливає, що при зміні декількох функцій або додаванні декількох фільтрів до архітектури системи пошуку спеціалізованої літератури, треба буде змінювати логіку додатку в декількох місцях, а саме в кожній функції.

Це означає, що немає централізованої точки, де можна було б конфігурувати усю послідовність ланцюга функцій. Це і може вирішити сервіс під назвою Step Functions. Ця технологія відрізняється від інших ще й тим, що має графічне відображення виконання кожного кроку, тому також ці кроки можна називати різними іменами, тобто це означає, що можна створити алгоритм послідовності системи пошуку та задати у коді виклик окремих лямбд [20].

Зараз потрібно створити кроки Step Functions, за допомогою мови Amazon, що створена у вигляді JSON (рис. 3.18).

```
{
  "Comment": "Diploma Step Function Example",
  "StartAt": "Filter faculty",
  "States": {
    "Filter faculty": {
      "Comment": "Filtering faculty.",
      "Type": "Task",
      "Resource": "arn:aws:lambda:us-east-2:639735456973:function:step_faculty_filter",
      "InputPath": "$",
      "ResultPath": "$.lambda_result",
      "OutputPath": "$.lambda_result",
      "Next": "Filter speciality"
    },
    "Filter speciality": {
      "Comment": "Filtering speciality.",
      "Type": "Task",
      "Resource": "arn:aws:lambda:us-east-2:639735456973:function:step_speciality_filter",
      "InputPath": "$",
      "ResultPath": "$.lambda_result",
      "OutputPath": "$.lambda_result",
      "Next": "Filter cathedra"
    },
    "Filter cathedra": {
```

Рисунок 3.18 – створення кроків AWS Step Functions.

Після успішного створення кроків, отримаємо граф виконання прикладу алгоритму системи пошуку спеціалізованої літератури (рис. 3.19)

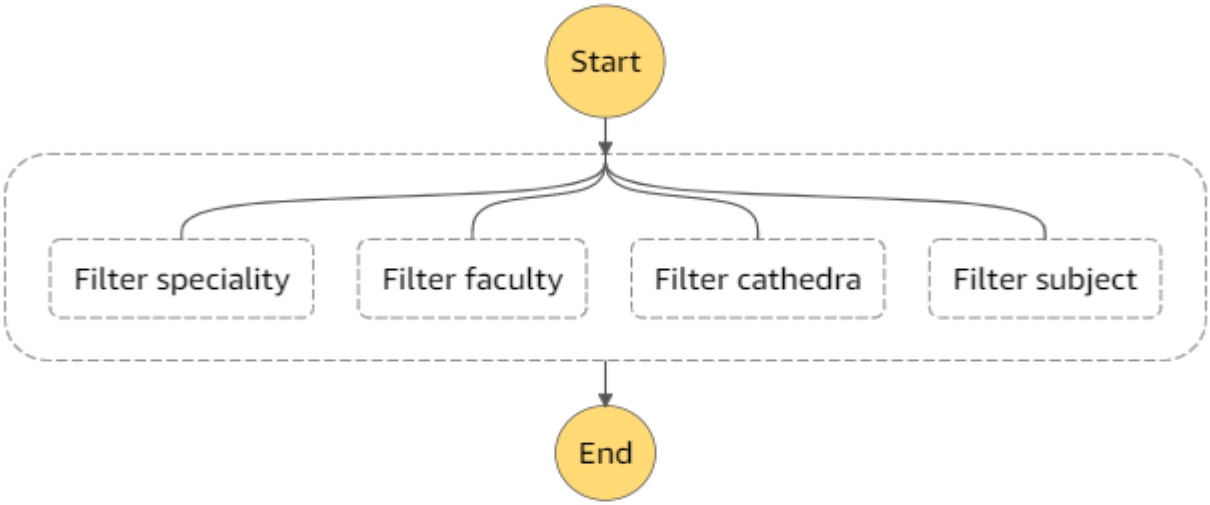

Рисунок 3.19 – приклад графу виконання алгоритму пошуку із Step Functions

Запустимо новий автомат стану, щоб переконатися у працездатності системи: коли система працює, тоді граф Step Functions стає зеленим (рис. 3.20).

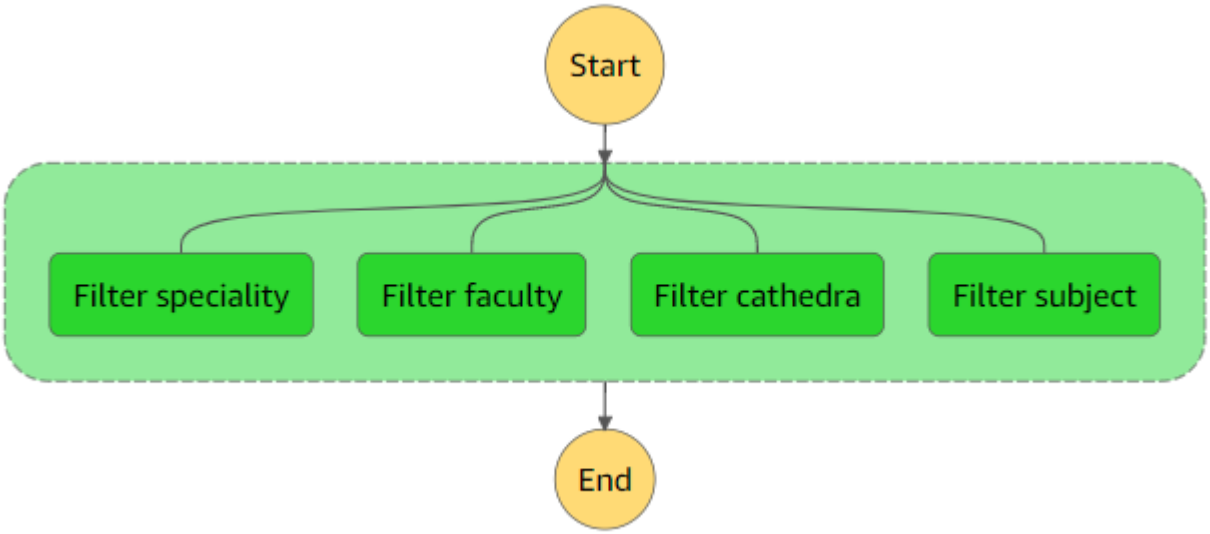

Рисунок 3.20 – працюючий алгоритм пошуку з використанням AWS Step Functions

3.5 Порівняння підходів

Для того, щоб обрати найкращий варіант архітектури пропонується протестувати продуктивність кожного з підходів. Додамо в таблицю книжок спочатку десять записів та перевіримо кожен з підходів. Надалі проведемо таку ж саму перевірку та заміряємо час, якщо в таблиці тисяча записів, сто тисяч записів або мільйон.

Розглянемо та порівняємо результат виконання різних підходів, коли в базі даних підприємства знаходиться лише 10 книг. Результат наведено у діаграмі з таблицею 3.1.

Таблиця 3.1 – Результат заміру продуктивності підходів з кількістю – 10 книг

Далі приведемо таку ж саму таблицю, але для тисячі записів у базі даних підприємства (таблиця 3.2).

Таблиця 3.2 – Результат заміру продуктивності підходів з тисячою книг

Для правильного порівняння, тепер варто збільшити кількість записів у базі даних до ста тисяч та перевірити за таким самим алгоритмом: запустити десять разів кожен з підходів, винести в табличку результати та побудувати наглядну діаграму результатів виконання кожного з запиту різних підходів. Так отримується висновок, коли потрібно використовувати який підхід (табл. 3.3)

Таблиця 3.3 – Результат заміру продуктивності підходів із ста тисячами книг

Таку ж саму операцію потрібно повторити і для мільйона записів, оскільки можна спостерігати тенденцію збільшення часу виконання першого підходу з віртуальним сервером, тобто при великому об’ємі даних цей сервер почав працювати повільніше, а другий та третій підхід починають працювати на рівні з ним, це, скоріш за все, означає, що ланцюг з безсерверних функцій як варіант архітектури системи пошуку може бути використаний для великого обсягу даних, оскільки можна спостерігати, що продуктивність його виконання із збільшенням даних залишається майже незмінною, а підхід із автоматом станів покращує роботу та прискорює її, роблячи функції незалежними одна від одної. Отже, пропонується провести тестування на мільйоні елементів (табл 3.4).

Таблиця 3.4 – Результат заміру продуктивності підходів з мільйоном книг

Розробляючи систему пошуку спеціалізованої літератури, вже було зазначено, що використання технологій кешування може покращити роботу з БД у декілька разів. У даній роботі базою даних системи виступає AWS Dynamo DB, тому для оптимізації запитів до цієї бази пропонується розглянути варіант використання Amazon DynamoDB Accelerator (DAX) [19].

За офіційною документацією від Амазон, DAX - це повністю керована служба кешування, яка знаходиться перед таблицями DynamoDB. Він працює в режимі запису та сумісний з API DynamoDB. Відповіді повертаються з кеш-пам'яті за мікросекунди, що робить DAX чудовим рішенням для послідовних з часом трудомістких навантажень. DAX зрозумілий та простий у використанні. Як керований сервіс, ви просто створюєте свій кластер DAX і використовуєте його для ваших існуючих читань і записів. Вам не потрібно турбуватися про

виправлення, обслуговування кластера, реплікацію чи управління несправностями.

Кожен кластер DAX може містити від 1 до 10 вузлів, є можливість додавати вузли для збільшення загальної пропускної здатності. Розмір кешу (також відомий як робочий набір) базується на розмірі вузла (dax.r3.large до dax.r3.8xlarge), який ви вибираєте під час створення кластера. Кластери працюють у VPC, а вузли розподілені по зонах доступності.

Вам потрібно буде використовувати DAX SDK для Java для інтегрування з DAX. Цей SDK взаємодіє з вашим кластером за допомогою низькорівневого інтерфейсу TCP, який налаштовано на низьку затримку та високу пропускну здатність.

Також у даній документації було проведено дослідження, як саме покращує продуктивність DAX сервіс. Перший запуск отримав доступ до DynamoDB безпосередньо та продемонстрував некешовану базову ефективність (рис 3.21).

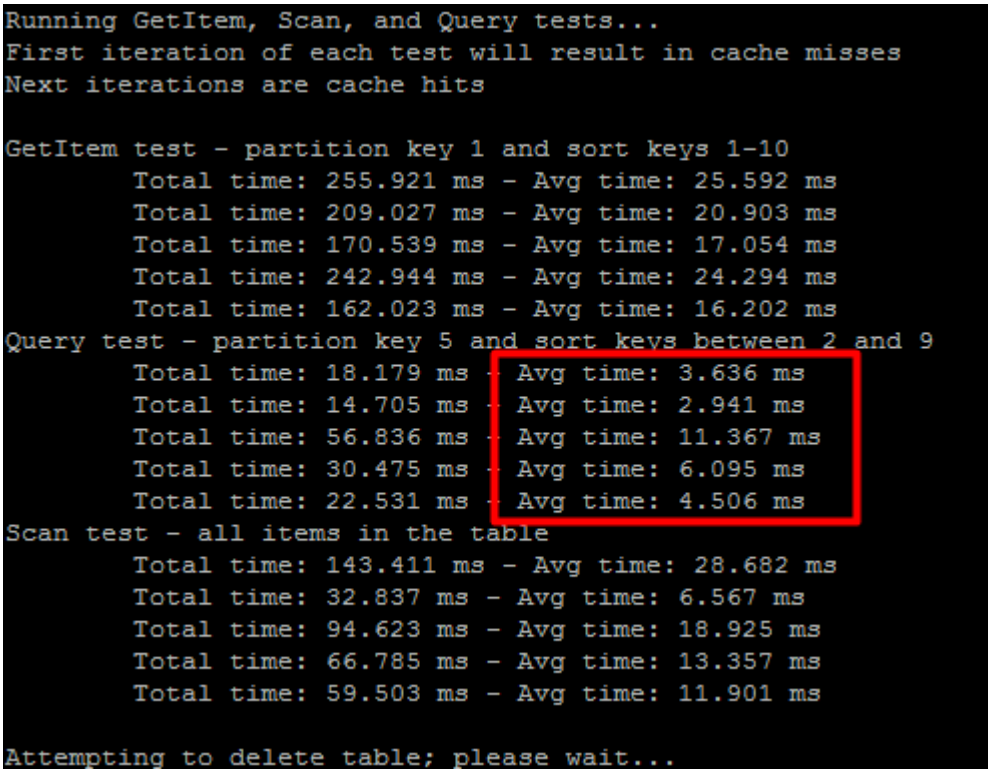

Рисунок 3.21 – запити до БД без використання DAX

Як видно з середньої групи результатів, запити виконувались від 2,9 до 11,3 мілісекунд. Другий запуск використовував DAX і показав вплив кешування на продуктивність (рис. 3.22).

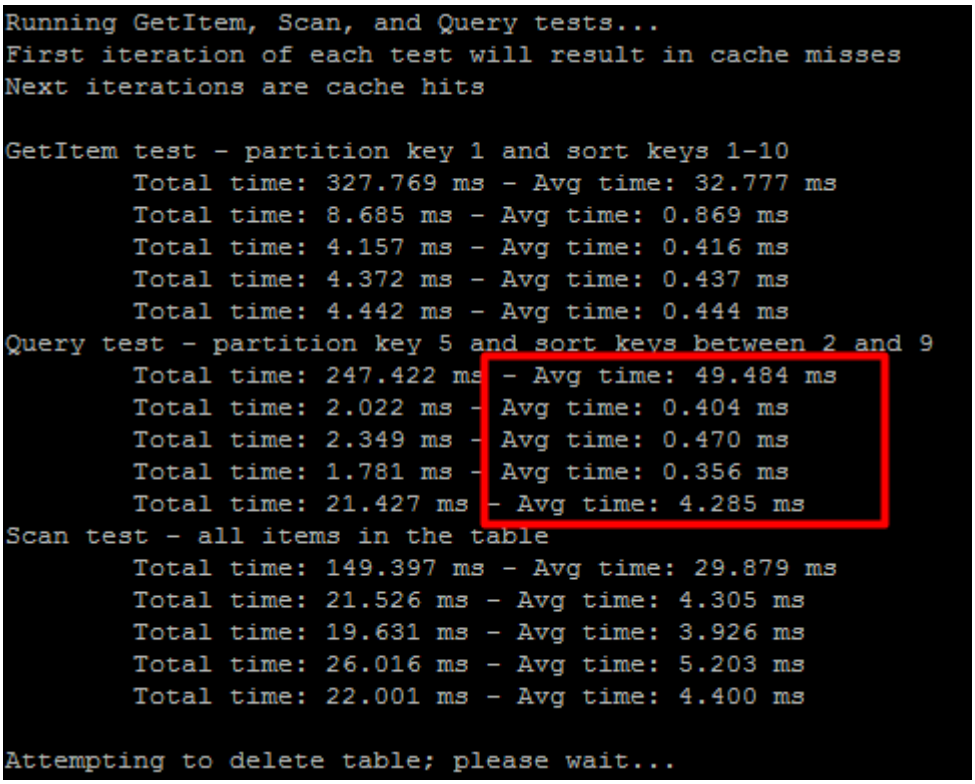

Рисунок 3.22 – запити з використанням DAX

Як можна побачити, перший запит з кешуванням виконувався довше, ніж без його використання, оскільки час був також затрачений на оновлення інформації в самому кеші, але послідуючі виконування показують, як зменшився час виконання від 0.35 до 0.47 мілісекунд.

Коли продуктивність системи вже була заміряна, потрібно також розповісти про ціноутворення кожного з варіантів будування архітектури системи пошуку спеціалізованої літератури.

Почнемо з першого варіанту, коли система пошуку представлена у вигляді окремого сервісу, розгорнутого у якості моноліту на фізичному сервері Амазону. Оскільки цей сервер буде написаний за допомогою однієї з серверних мов програмування, що буде реалізована за принципами RESTful додатків, де буде включатися інтеграція з БД, шар сервісів та контролерів та оскільки даних для

обробки може бути багато, пропонується звертати увагу на кількість оперативної пам’яті та розглянути наступні варіанти інстансів Амазону [21] (табл. 3.5).

Таблиця 3.5 – варіанти використання різних серверів Амазону

Тип серверу	Безкоштовний рівень	RAM (GiB)	Linux USD за годину	Windows USD за годину
t2.nano	FALSE	0.5	0.0058	0.0081
t2.micro	TRUE	1	0.0116	0.0162
t2.small	FALSE	2	0.023	0.032
t2.medium	FALSE	4	0.0464	0.0644
t2.large	FALSE	8	0.0928	0.1208
t2.xlarge	FALSE	16	0.1856	0.2266
t2.2xlarge	FALSE	32	0.3712	0.4332

Тобто в залежності від потреби, ціна може також змінюватись, наприклад, якщо розглядається маленьке підприємство, де не дуже багато книжок, котрі можуть бути використані для пошуку та фільтрації, то це означає, що додаток, котрий буде там працювати, навіть дістаючи усі дані з БД, не буде навантажувати оперативну пам’ять серверу досить сильно, тоді можна обрати варіант серверу t2.nano або t2.micro. Але якщо кожна з книжок матиме досить багато характеристик, що будуть відображатися строками, кожен символ з яких важить 2 байти у кодировці UTF-8, наприклад 10 полей, кожне з яких включає 10 символів у середньому, це вже 200 байтів на книжці. При тисячі книжок, діставання усіх них – це вже 200 000 байтів, тобто 0.2 гіга-байти. Якщо 3 користувача одночасно запитають фільтрацію, то це викличе навантаження у 0.6 гіга-байт оперативної пам’яті, що вже не зможе бути оброблено t2.nano сервером. У такому випадку, використання серверу t2.medium, що буде працювати, наприклад 30 годин на місяць, тобто годину кожного дня в середньому, буде коштувати 13,92 USD на місяць.

Тепер розглянемо другий спосіб реалізації архітектури підсистеми пошуку спеціалізованої літератури, коли сервіс розгортуватиметься як декілька окремих

безсерверних функцій, що будуть виконуватись паралельно або послідовно, в залежності від конфігурації системи. Для цього потрібно розуміти, скільки коштує AWS Lambda. Для цього можна звернутися до електронного ресурсу від Амазона, що дозволяє порахувати кошти. Побачимо загальну таблицю(табл. 3.6):

Таблиця 3.6 – Загальне ціноутворення AWS Lambda

Запити	0.20 дол. США за 1 мільйон запитів
Час виконання	0.0000166667 дол. США за кожну Гб-секунду

Тобто виклики лямбда-функцій від Амазону рахуються окремо від пам’яті, що була затрачена під час використання. Амазон також надає змогу налаштування кожної функції на використання максимальної пам’яті: менше пам’яті – нижче кошти.

Результати наступної таблиці були взяті з офіційного сайту Амазон та являються дійсним відображенням ціноутворення використання функцій в залежності від витраченого об’єму пам’яті під час підіймання та обробки даних запиту користувача (табл. 3.7).

Таблиця 3.7 – ціноутворення AWS Lambda в залежності від пам’яті

Пам’ять (MB)	Кошти за 1 мілісекунду
128	0.0000000021 дол. США
512	0.0000000083 дол. США
1024	0.0000000167 дол. США
1536	0.0000000250 дол. США
2048	0.0000000333 дол. США
3072	0.0000000500 дол. США
4096	0.0000000667 дол. США

Продовження таблиці 3.7 – ціноутворення AWS Lambda в залежності від пам’яті

Пам’ять (MB)	Кошти за 1 мілісекунду
5120	0.0000000833 дол. США
6144	0.0000001000 дол. США
7168	0.0000001167 дол. США
8192	0.0000001333 дол. США
9216	0.0000001500 дол. США
10240	0.0000001667 дол. США

Якщо ви виділили своїй функції 512 МБ пам’яті, виконали її 3 мільйони разів протягом одного місяця, і вона працювала протягом 1 секунди кожного разу, ваші витрати будуть розраховуватися наступним чином:

Щомісячні обчислювальні нарахування рахуються так:

Щомісячна ціна обчислень становить 0,00001667 дол. США за ГБ-с, а безкоштовний рівень забезпечує 400 000 ГБ-с.

Загальне обчислення (секунди) = 3М * (1s) = 3 000 000 секунд

Всього обчислень (ГБ-с) = 3 000 000 * 512 МБ / 1024 = 1 500 000 ГБ-с

Загальне обчислення - Безкоштовне обчислення рівня = Щомісячні платні обчислення ГБ

1 500 000 ГБ-с - 400 000 ГБ вільного рівня = 1 100 000 ГБ-с

Щомісячні обчислювальні платежі = 1,100,000 * 0,00001667 $ = 18,34 $

Щомісячна плата за запит буде розрахована наступним чином:

Щомісячна ціна запиту становить 0,20 дол. США за 1 мільйон запитів, а безкоштовний рівень забезпечує 1 мільйон запитів на місяць.

Загальна кількість запитів - Безкоштовні запити рівня = Щомісячні платні запити

3 мільйони запитів - 1 мільйон безкоштовних запитів рівня = 2 мільйони щомісячних платіжних запитів

Щомісячна плата за запит = 2 млн * 0,2 дол. США / м = 0,40 дол. США

Загальна щомісячна плата буде обчислюватися наступним чином:

Загальна вартість = кошти за обчислення + кошти за запит = 18,34 дол. США + 0,40 дол. США = 18,74 дол. США на місяць

Щоб дослідити економічну складову використання підходу із автоматами стану Амазон (AWS Step Functions), треба тільки дослідити, скільки коштує цей окремий сервіс, оскільки він також буде викликати ті ж самі безсерверні функції (табл 3.8).

Таблиця 3.8 – кошти за використання AWS Step Functions

Безкоштовні виклики на місяць	Кошти за 1 виклик
4000	0.000025 дол. США

Це означає, що за 5 000 (п’ять тисяч) виконувань на місяць будуть коштувати 0.025 дол. США, не включаючи кошти на використання лямбда-функцій. Звертаючись до попереднього обчислення безсерверних функцій, якщо 300 (триста) функцій будуть викликані по 10 000 (десять тисяч) разів в одному виконуванні автомату стану, то матимемо 18,74 дол. США + (10 000 – 4000) * 0.000025 = 18,74 + 0,15 = 18,89 дол. США.

ВИСНОВКИ

У даній роботі розглядалася проблема будування архітектури системи пошуку спеціалізованої літератури, адже наразі існує досить багато спеціалізованих джерел, але усі вони розташовані у базах даних підприємств з обмеженим функціоналом, доступом, швидкістю та зручністю пошуку таких книжкових ресурсів. Було зазначено, що цільовою аудиторією користувачів системи пошуку можуть виступати працівники підприємств, що мають у своїх сховищах даних корисні джерела, необхідні для опрацювання усім співробітникам даних підприємств.

Було проаналізовано та порівняно варіанти мікросервісної та монолітної архітектури та обґрунтовано вибір саме мікросервісної архітектури. Найважливішим фактором рішення стало те, що підприємству необхідно тільки надати характеристики користувачів, за якими можуть фільтруватися книжкові джерела та самі книжкові джерела, надалі система пошуку виступатиме як окремий додаток, що буде доступний за API та не впливатиме на роботу суцільної системи підприємства.

У даній роботі також було досліджено використання хмарних обчислень та розгортування системи пошуку у хмарній інфраструктурі та доведено, чому це вирішення являється кращим за локальну інфраструктуру. У якості постачальника хмарних послуг було обрано Amazon Web Services.

Розглянуто варіанти таких архітектур та запропоновано три вирішення проблеми будування стійкої та працюючої системи: коли сервіс розгорнутий як окремий суцільний монолітний; як ланцюжок з незалежних функцій; як окремий автомат станів, котрий сам викликатиме усі функції, необхідні для фільтрування під час пошуку спеціалізованої літератури.

Як результат роботи, мається три працюючі варіанти архітектури системи пошуку спеціалізованої літератури, кожна з яких підпадає під конкретні потреби різноманітних підприємств.

У роботі досліджено економічну складову кожного з підходів. Починаючи з першого, він може бути як найдешевшим, так і найдорожчим: при використанні найлегшого з серверів Амазону підхід буде найдешевшим, але треба брати в увагу

те, що пам’яті вистачить на дуже обмежений список книг. Більший список книг – більший сервер Амазону, тоді дорожче споживання. Вже на середньому сервері, такий підхід стане найдорожчим. До недоліків також можна віднести відсутність масштабування, даний варіант підходить для підприємств з фіксованим списком фільтрів для пошуку та, бажано, книг.

Другий варіант архітектури з ланцюга окремих безсерверних функцій стає вигіднішим при значній кількості літературних джерел. Як було відображено, на малій кількості книг, підхід працює не швидше за велику кількість. Ціна використання такого підходу залишається незмінною від кількості книг і являється найдешевшим варіантом при великій кількості книг. До переваг можна віднести, що концепт безсерверних функцій не виділяє цілий сервер під обмежений набір функцій, що значно поліпшує процес розгортки, зміни, написання нового функціоналу в алгоритм виконування. Повільність підходу також залежить від щільного зв’язку між функціями за допомогою протоколів Амазону, що вирішується третім підходом.

Третім варіантом у роботі було досліджено архітектуру з використанням автомату стану, що викликає та налаштовую ланцюг з безсерверних функцій. Цей підхід відрізняється тим, що час його роботи практично незмінний для будь-якої кількості книг. Він являться найповільнішим для малої кількості та найшвидшим для великою. До переваг відноситься також те, що це найкращий підхід відносно масштабування системи: додавання нових фільтрів пошуку, зміни стану, контролю вхідних параметрів та стійкості до помилок або їх відстежування. Даний підхід також характеризується незначними коштами, кошти за функції, як у другого підходу, додаючи кошти за виконання автомату стану, що має безкоштовну кількість викликів та при правильному налаштуванні може додати лише близько 20 дол. США.

ПЕРЕЛІК ПОСИЛАНЬ

1. М. Фаулер. Архитектура корпоративных программных приложений / М. Фаулер. – Издательский дом Вильямс, 2006 – 544 с.

2. Ньюмен С. Создание микросервисов / Ньюмен С. – СПб.: Питер, 2016 – 304 с.

3. Фаулер М. Скотт К. UML. Основи / СПб.: Символ, 2006 - 184 стр.

4. The pros and cons of Cloud vs. in-house Servers – Режим доступу: https://sysgen.ca/cloud-vs-in-house-servers/ – Дата доступу: 04.05.2021

5. Кулямін В.В. Технологія програмування. Компонентний підхід. / Мінськ «Біном. Лабораторія знань», 2007 – 463 стр.

6. E. Evans. Domain-Driven Design: Tackling Complexity in the Heart of Software / E. Evans – Addison-Wesley, 2003 – 560 p.

7. Martin Fowler – Microservices – Режим доступу: http://martinfowler.com/articles/microservices.html – Дата доступу: 28.04.2021

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. – Режим доступу: https://nginx.com/blog/introduction-to-microservices/ – Дата доступу: 13.04.2021

10. Офіційний сайт Google: Compare AWS and Azure services to Google Cloud. – Режим доступу: https://cloud.google.com/free/docs/aws-azure-gcp-service-comparison - Дата доступу: 29.05.2021;

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

12. Using an API Gateway. – Режим доступу: https://nginx.com/blog/buildingmicroservices-using-an-api-gateway/ – Дата доступу: 27.05.2017

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

14. AWS Step Functions: Developer Guide – Режим доступу: https://www.amazon.com/AWS-Step-Functions-Developer-Guide-ebook/dp/B078XBSLY5 – Дата доступу: 10.05.2021

15. Офіційний сайт C++ Micro Services. – Режим доступу: http://cppmicroservices.org/ – Дата доступу: 02.05.2021

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

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

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

19. Amazon DynamoDB Accelerator (DAX) – In-Memory Caching for Read-Intensive Workloads – Режим доступу: https://aws.amazon.com/blogs/aws/amazon-dynamodb-accelerator-dax-in-memory-caching-for-read-intensive-workloads/ – Дата доступу: 08.05.2021

20. Ситнікова, П.Е. AWS Step Functions як невід'ємна складова архітектура системи з послідовним виконанням безсерверних функцій / П.Е. Ситнікова, О.П. Говдерчак // АСУ та прилади автоматики. - 2021. – вип. 177. – С. 17-22

21. AWS Pricing Calculator – Режим доступу: https://calculator.aws/#/ – Дата доступу: 11.05.2021