МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ

Харківський національний університет радіоелектроніки

Факультет Комп’ютерних наук
(повна назва)

Кафедра Програмної інженерії
(повна назва)

КВАЛІФІКАЦІЙНА РОБОТА

Пояснювальна записка

рівень вищої освіти другий (магістерський)

Дослідження ефективності методів та засобів розробки бессерверних додатків для обробки зображень
(тема)

Виконав:
Студент 3 курсу, групи ІПЗм-19-3
Шумілін В.О.
(прізвище, ініціали)

Спеціальність 121-Інженерія програмного забезпечення
(код і повна назва спеціальності)

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

Керівник проф. Лєсна Н.С
(посада, прізвище)

Допускається до захисту

Зав. кафедри ______________ З.В. Дудар
(підпис) (прізвище, ініціали)

2021р.

Харківський національний університет радіоелектроніки

Факультет Комп'ютерних наук
(повна назва)

Кафедра Програмної інженерії
(повна назва)

Рівень вищої освіти другий (магістерський)

Спеціальність 121- Інженерія програмного забезпечення
(код і повна назва спеціальності)

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

Освітня програма Інженерія програмного забезпечення
(повна назва)

ЗАТВЕРДЖУЮ:
Зав. кафедри ________________
(підпис)
« ____ » ____________ 2021 р.

ЗАВДАННЯ
НА КВАЛІФІКАЦІЙНУ РОБОТУ

студента Шумілін Владислав Олександрович
(прізвище, ім'я, по батькові)

1. Тема роботи Дослідження ефективності методів та засобів розробки бессерверних додатків для обробки зображень
затверджена наказом університету від ______________ № ________
2. Термін подання роботи до екзаменаційної комісії 5 травня 2021р.
3. Вихідні дані до роботи *електронні ресурси за обраною тематикою, рекомендацій щодо вибору провайдера бессерверного обчислення та удосконалення окремих функцій провайдера, прототип платформи бессерверних обчислень, Visual Studio 2019, мови JS, C#.*
4. Перелік питань, що потрібно опрацювати в роботі *аналіз проблемної області і постановка задачі, аналіз інфраструктури бессерверної архітектури, розробка прототипу обчислювальної платформи, критерії та методика оцінювання ефективності методів та засобів розробки, аналіз результатів та розробка рекомендацій.*
5. Перелік графічного матеріалу із зазначенням креслеників, схем, слайдів, ілюстрацій *мета роботи, актуальність дослідження, постановка задачі, методика оцінювання ефективності, процедура моделювання функціональних параметрів, критерії оцінювання ефективності, аналіз результатів, рекомендації, висновки.*

6. Консультанти розділів роботи

Найменування розділу	Консультант (посада, прізвище, ім’я, по батькові)	Позначка консультанта про виконання розділу	
		підпис	дата
Спецчастина	проф. Лєсна Н.С.		03.05.2021

КАЛЕНДАРНИЙ ПЛАН

№	Назва етапів роботи	Терміни виконання етапів роботи	Примітка
1	Аналіз проблемної області	26.01.21 – 12.02.21	виконано
2	Постановки задачі	13.02.21 – 15.02.21	виконано
3	Аналіз інфраструктури бессерверної архітектури	16.02.21 – 20.02.21	виконано
5	Розробка методики оцінювання ефективності	21.02.21 – 24.02.21	виконано
6	Визначення критеріїв ефективності	24.02.21 – 10.03.21	виконано
7	Розробка процедури моделювання функціональних параметрів	10.03.21 – 20.03.21	виконано
8	Розробка прототипу бессерверної обчислювальної платформи	20.03.21 – 04.04.21	виконано
9	Проведення оцінювання ефективності методів та засобів	04.04.21 – 16.04.21	виконано
	Розробка рекомендацій щодо вибору провайдера бессерверного обчислювання	17.04.21 – 22.04.21	
10	Оформлення статті	24.04.21 – 01.05.21	виконано
11	Підготовка пояснювальної записки	01.05.21 – 03.05.21	виконано
12	Підготовка презентації та доповіді	01.05.21 – 08.05.21	виконано
13	Нормоконтроль	08.05.21 – 13.05.21	виконано
14	Рецензування	08.05.21 – 13.05.21	виконано
15	Занесення диплома в електронний архів	14.05.2021	виконано
16	Попередній захист	15.05.2021	виконано
17	Допуск до захисту у зав. кафедри	15.05.2021	виконано

Дата видачі завдання 25 січня 2021р.

Студент ______________________________

(підпис)

Керівник роботи ______________ проф. Лєсна Н.С.

(підпис) (посада, прізвище, ініціали)

РЕФЕРАТ/ABSTRACT

Звіт з науково-дослідної практики: 95 стор., 19 рис., 6 таблиць, 20 джерел.

АРХІТЕКТУРА ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ, БЕССЕРВЕРНІ ОБЧИСЛЕННЯ, ОБРОБКА ЗОБРАЖЕНЬ, AZURE FUNCTIONS, AWS LAMBDA, GOOGLE CLOUD FUNCTIONS, SERVERLESS.

Метою роботи є визначення найбільш ефективних методів та засобів розробки бессерверних додатків для обробки зображень, виявлення переваг і недоліків цих методів та порівняння між собою різних провайдерів бессерверних обчислень з урахуванням потреб бізнесу.

Запропонована методика оцінювання ефективності методів та засобів розробки бессерверних додатків для обробки зображень, визначені критерії оцінювання ефективності та розроблена процедура моделювання функціональних параметрів провайдерів бессерверних обчислень. Розроблений прототип бессерверної обчислювальної платформи. Отримані результати оцінювання ефективності.

AZURE FUNCTIONS, AWS LAMBDA, GOOGLE CLOUD FUNCTIONS, SERVERLESS, IMAGE PROCESSING, SERVERLESS COMPUTING, SOFTWARE ARCHITECTURE.

The purpose of the work is to determine the most effective methods and tools for developing server-free applications for image processing, identify the advantages and disadvantages of these methods and compare different providers of server-free computing, taking into account the needs of the business.

The methodology for evaluating the effectiveness of methods and for creating serverless data for processing images, for evaluating the criteria for evaluating efficiency, and for modeling the functional parameters of providers for serverless computations, has been promoted. The fragmentation is a prototype of serverless computing platforms. Otrimani results and evaluation of efficiency.

Я, Шумілін Владислав Олександрович, студент гр. ІПЗм-19-3, здобувач вищої освіти на другому (магістерському) рівні кафедри «Програмна інженерія», заявляю: моя кваліфікаційна робота на тему «Дослідження ефективності методів та засобів розробки бессерверних додатків для обробки зображень», що буде представлена в екзаменаційну комісію для публічного захисту, виконана самостійно, в ній не містяться елементи плагіату і вона може бути опублікована в електронному архіві відкритого доступу ElAr KhNURE. Всі запозичення з друкованих та електронних джерел мають відповідні посилання.

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

ЗМІСТ

Вступ 9
1 Аналіз проблемної області та постановка задачі 12
1.1 Клієнт-серверна та бессереверна архітектури програмного забезпечення 12
1.2 Аналіз переваг та недоліків бессерверної архітектури програмного забезпечення 18
1.3 Порівняння політики ціноутворення та функціональних параметрів послуг основних провайдерів бессерверних обчислень 21
1.4 Serverless бази даних 24
1.5 Особливості бессерверної обробки зображень 27
1.6 Постановка задачі 29
2 Аналіз інфраструктури бессерверної архітетктури 30
2.1 Принцип роботи бессерверних функцій 30
2.2 Метадані бессерверних функцій 32
2.3 Виконання бессерверної функції 33
2.4 Незмінність бессерверних функцій 35
2.5 Генерація контейнерів 35
2.6 Видалення контейнера 36
2.7 Образ контейнера 37
2.8 Асинхронне виконання 38
2.9 Контейнери для Windows 38
2.10 Безпека 39
2.11 Логування і налагодження бессерверних додатків 40

3 Критерії, методика та результати оцінювання ефективності методів та засобів розробки бессерверних додатків для обробки зображень 42
3.1 Методика та критерії ефективності методів та засобів розробки бессерверних додатків для обробки зображень 42
3.2 Результати оцінюванння ефективності методів та засобів розробки бессерверних додатків 44
3.2.1 Результати оцінювання показників масштабування бессерверних функцій 45
3.2.2 Результати оцінювання показників затримки холодного старту бессерверних функцій 47
3.2.3 Результати оцінювання показників швидкості обробки запитів бессерверних функцій 48
3.2.4 Результати оцінювання показників масштабованості бессерверної архітектури в порівнянні з монолітною архітектурою 51
3.3 Рекомендацій щодо вибору провайдера бессерверного обчислення та удосконалення окремих функцій провайдера 54
4 Опис програмної системи 57
4.1 Компоненти прототипу бессерверної обчислювальної платформи 57
4.2 Функціональні особливості прототипу бессерверної обчислювальної платформи та обгрунтування обраних технологій 58
4.3 Тестування прототипу бессерверної обчислювальної платформи 59
Висновки 62
Перелік джерел посилань 64
Додаток А Перелік джерел посилання за науковими напрямами керівника та науковців кафедри програмної інженерії 66
Додаток Б Звіт результатів перевірки кваліфікаційної роботи на унікальність тексту 67

Додаток В Слайди презентації 68

Додаток Г Лістинг коду програми 79

Додаток Д Наукові публікації 86

Додаток Е Експертний висновок результатів перевірки кваліфікаційної роботи на відповідність оформлення вимогам ДСТУ 3008:2015.. 95

ВСТУП

Ми живемо у епоху діджиталізації, світ швидко змінюється, важко уявити бізнес-проект, який би не використував переваги інтернету. Все це приводить до збільшення кількості ІТ фахівців, веб-сайтів, мобільних та настільних додатків. Як наслідок, ми маємо величезну кількість серверів, які обробляють незліченну кількість даних, але щоб використовувати такий сервер потрібно оплатити послуги адміністратора, ліцензію на програмне забезпечення, аренду серверу чи придбати свій. У випадку, коли сервер має значне недовикористання ресурсів або простій, власник витрачає зайві гроші.

У свою чергу розробники програмного забезпечення витрачають свій час на вирішення бізнес-проблем за допомогою коду. Після чого операційна команда витрачає величезну кількість годин, щоб запустити цей код на доступних серверах та слідкують за тим, щоб сервер працював безперебійно, що є нескінченим завданням. У звязку з цим виникло питання чому б не делегувати ці задачі комусь-іншому?

Багато інновацій в ІТ за останні два десятиліття — віртуальні машини, хмарні обчислення, контейнери — були зосереджені на тому, щоб не потрібно було думати про основну фізичну машину, на якій працює код. Бессерверні обчислення - це все більш популярна парадигма, яка доводить це бажання до свого логічного завершення: з бессерверними обчисленнями не потрібно нічого знати про апаратне забезпечення або ОС, на якій працює код, оскільки про все це дбає постачальник послуг.

Бессерверні обчислення - це модель виконання хмарних обчислень, в якій хмарний провайдер виступає в ролі сервера, динамічно керуючи розподілом машинних ресурсів. Ціноутворення базується на фактичній кількості ресурсів, споживаних додатком, а не на попередньо придбаних одиницях потужності.

Таким чином, існують дві найбільші переваги бессерверних обчислень: розробники можуть зосередитись на бізнес-цілях коду, який вони пишуть, а не на

інфраструктурних питаннях; а організації платять лише за обчислювальні ресурси, які вони насправді використовують, а не купують фізичне обладнання або орендують хмарні екземпляри, які в основному простоюють без діла.

Наприклад, у користувача може бути програма, яка більшу частину часу простоює, але за певних умов повинно одночасно обробляти багато запитів на події. Або може бути програма, яка обробляє дані, надіслані з пристроїв IoT, з обмеженим або періодичним підключенням до Інтернету. В обох випадках традиційний підхід вимагав би надання міцного сервера, який міг би справлятися з піковими робочими навантаженями, але цей сервер був би недостатньо використаний більшу частину часу. Завдяки бессерверній архітектурі користувач платить лише за ті серверні ресурси, які фактично використовуються. Бессерверні обчислення також добре підходять для певних видів пакетної обробки. Одним з канонічних прикладів використання бессерверної архітектури є служба, яка завантажує та обробляє ряд окремих файлів зображень та надсилає їх до іншої частини програми.

На даний момент існує кілька конкурентів на ринку бессерверних обчислень, кожен з них має свої переваги і недоліки і має сервіс з обробки зображень в хмарі. Здебільшого, обробка зображень здійснюється як фонова операція, і існує декілька підходів щодо побудови бессерверної архітектури для обробки зображень, які потребують детального розгляду.

Актуальність теми полягає у необхідності систематизації переваг та недоліків методів та засобів розробки бессерверних додатків для обробки зображень за для вибору найбільш ефективних методів та засобів обробки зображень в залежності від вимог певної бізнес задачі.

Тема роботи безпосередньо пов’язана з програмами наукових досліджень кафедри ПІ. Використані набуті знання з навчальних дисциплін і матеріали наукових публікацій з enterprice-програмування, основ баз даних, архітектури програмного забезпечення, хмарних обчислень.

Мета роботи полягає у визначенні найбільш ефективних методів та засобів розробки бессерверних додатків для обробки зображень та порівняні між собою різних провайдерів бессерверних обчислень з урахуванням потреб бізнесу.

Об’єктом дослідження є процес обробки зображень за допомогою бессерверних додатків.

Предмет дослідження: методи та засоби розробки бессерверних додатків для обробки зображень.

У ході даної роботи задля дослідження ефективності використаних методів та засобів обробки зображень застосовувались теоретичні (наукове моделювання) та емпіричні (експеримент, дослідження, вимірювання) наукові методи.

Наукова новизна. Вперше запропонована методика оцінювання ефективності методів та засобів розробки бессерверних додатків, розроблена процедура моделювання функціональних параметрів провайдерів бессерверних обчислень.

Практична значимість. Отримані результати оцінювання ефективності методів та засобів розробки бессерверних додатків для обробки зображень та розроблені рекомендації щодо їх використання дозволять здійснити обгрунтований вибір найбільш ефективних методів та засобів бессерверних обчислень в залежності від висунутих вимог для певної бізнес задачі, а також збільшити швидкість обробки запитів.

Публікації. Результати дослідження, проведеного в атестаційній роботі, були опубліковані тези доповіді “Дослідження ефективності методів та засобів розробки бессерверних додатків для обробки зображень” в матеріалах XXIV міжнародного молодіжного форума «Радіоелектроніка та молодь у XXI столітті» та опублікована в журналі «Наука онлайн» стаття «Результати дослідження ефективності методів та засобів розробки бессерверних додатків для обробки зображень».

1 АНАЛІЗ ПРЕДМЕТНОЇ ОБЛАСТІ ТА ПОСТАНОВКА ЗАДАЧІ

1.1 Клієнт-серверна та бессереверна архітектури програмного забезпечення

Архітектура програмного забезпечення системи відображає організацію чи структуру системи. Система являє собою набір компонентів, що виконують певну функцію або набір функцій. Іншими словами, архітектура програмного забезпечення забезпечує міцну основу, на якій може бути побудовано програмне забезпечення.

Проектування архітектури програмного забезпечення виконується після етапу аналізу і формулювання вимог. Побудова архітектури системи здійснюється шляхом визначення цілей системи, її вхідних і вихідних даних, декомпозиції системи на підсистеми, компоненти або модулі та розроблення її загальної структури.

Існує безліч архітектурних схем і принципів високого рівня, зазвичай використовуваних в сучасних системах. Їх часто називають архітектурними стилями. Архітектура програмної системи рідко обмежується одним архітектурним стилем. Замість цього, часто використовується комбінація стилів, наприклад мікросервісна та трирівнева архітектури, де мікросервіс реалізований за принципами трирівневої архітектури, а саме має рівень доступу до бази даних, рівень бізнес логіки та рівень користувацього інтерфейсу.

Розглянемо найпопулярнішу архітектуру програмного забезпечення - "клієнт-сервер" [1]. Це архітектура, в якій мережеве навантаження розподілено між постачальниками послуг, званими серверами, і замовниками послуг, званими клієнтами.

Зазвичай програми – це сервери та клієнтські програми розташовані на різних обчислювальних машинах і взаємодіють між собою через обчислювальну мережу за допомогою мережевих протоколів, але вони можуть бути розташовані також і на одній машині. Схематичне зображення взаємодії клієнтів та серверу наведено на рисунку 1.1.

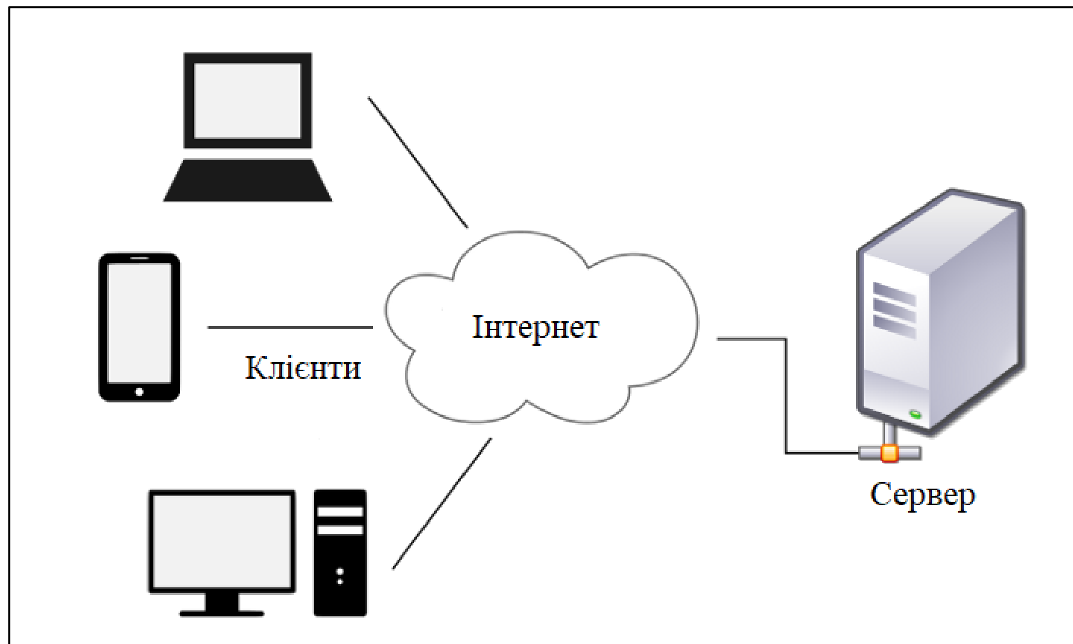

Рисунок 1.1 - Схематичне зображення взаємодії клієнтів та серверу

Програми-сервери очікують від клієнтських програм запити і надають їм свої ресурси у вигляді даних (наприклад, завантаження файлів за допомогою HTTP, FTP, BitTorrent, потокова мультимедія або робота з базами даних) або у вигляді сервісних функцій (наприклад, робота з електронною поштою, спілкування за допомогою систем миттєвого обміну повідомленнями або перегляд web-сторінок).

Традиційна архітектура клієнт-сервер має дві основні проблеми [2]:

- єдина точка відмови. Якщо сервер з будь-якої причини виходить з ладу, весь веб-сайт вийде з ладу і стане недоступним для користувачів та клієнтів. Багато підприємств, як правило, мають свою виробничу базу даних, що зберігається на тому ж сервері, що і їх веб-сайт, а це означає, що якщо сервер вийде з ладу, вони також не зможуть підключитися до їх бази даних;
- завжди включений. Незалежно від кількості користувачів на сайті, завжди необхідно платити за час сервера.

Також розробники веб-серверу повині вирішити наступні операційні задачі:

- забезпечення обчислювальної потужності;

- географічне розповсюдження додаткових копій для збереження послуги у разі несправності;
- балансування навантаження і маршрутизація для ефективного використання ресурсів;
- масштабування вгору або вниз у відповідь на зміну навантаження системи;
- моніторинг стану серверу;
- логування журналу системних повідомлень, необхідних для відладки або налаштування продуктивності;
- оновлення системи, включаючи виправлення безпеки;

Бессерверна архітектура спрямована на вдосконалення традиційної структури клієнт-сервера. Ця архітектура (також відома як бессерверні обчислення або функція як послуга, FaaS) - це шаблон дизайну програмного забезпечення, при якому додатки розміщуються сторонніми службами, усуваючи втручання розробника серверного програмного та апаратного забезпечення. Додатки розбиті на окремі функції, які можна викликати та масштабувати окремо. У будь-якому випадку таке програмне забезпечення виконується на сервері, проте все керування цим сервером виконується не розробником, а сторонім сервісом – провайдером [3].

Провайдер бессерверних обчислень є платформою що дозволяє побудувати розподілений веб-сервер. Розподілені обчислення або розподілена обробка даних – спосіб розв’язання обчислювальних завдань з використанням декількох комп’ютерів, об’єднаних в одну мережу. Розподілені обчислення є окремим випадком паралельних обчислень, тобто одночасного розв'язання різних частин одного обчислювального завдання декількома процесорами одного або кількох комп'ютерів. Тому необхідно, щоб завдання, що розв'язується, було сегментоване – розділене на менші задачі, що слабо пов’язані між собою та можуть обчислюватися паралельно. Особливістю розподілених обчислювальних систем, на відміну від локальних суперкомп’ютерів, є можливість майже необмеженого приросту продуктивності за рахунок горизонтального масштабування.

Бессерверна архітектура будуються шляхом видалення бази даних із сервера та запуску окремих служб та функцій за вимогою користувача. Ці функції розміщує платформа «Функція як послуга (FaaS)», така як Amazon Web Services (AWS) Lambda, Google Cloud Functions, IBM OpenWhisk або Microsoft Azure Functions. Вказані платформи потребують плату лише з урахуванням того, скільки разів виконувались функції, як довго вони працюють та виділеної пам'яті. Схематичне зображення Serverless архітектури наведено на рисунку 1.2.

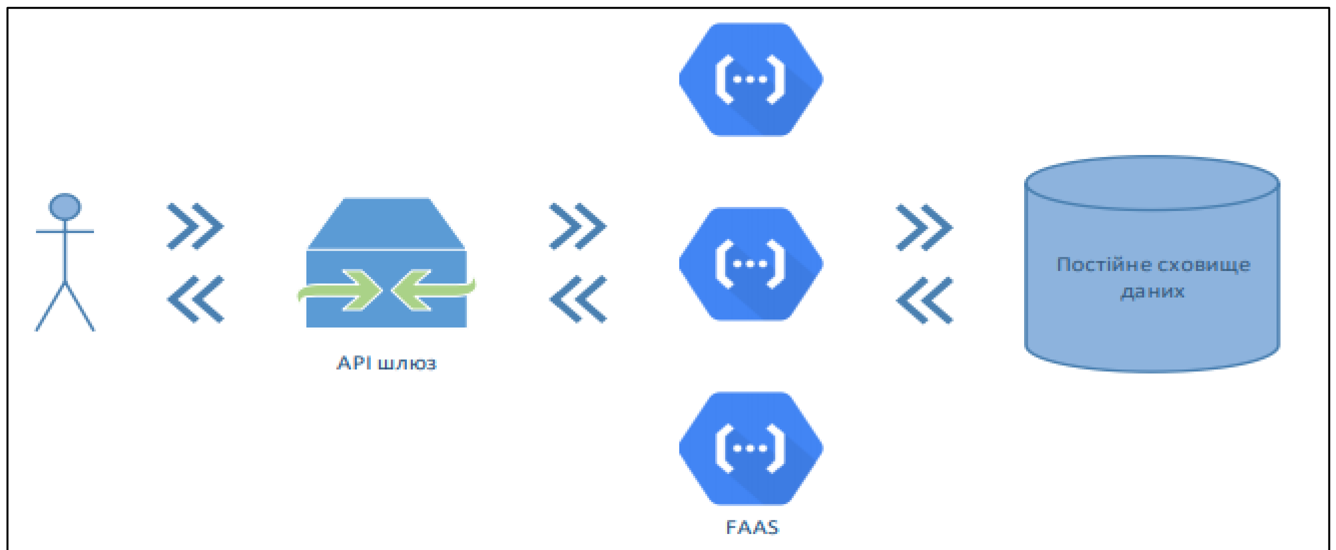

Рисунок 1.2 - Схематичне зображення Serverless архітектури

Деякі експерти вважають, що бессерверні обчислення - це лише ребрендинг попередніх рішень, а саме узагальнення платформи як сервісу хмарних продуктів, таких як Parse, Heroku чи Firebase. Інша частина фахівців вважають, що спільне використання веб-хостингу, яке було популярне в 90-х роках, надавало багато того, що можуть запропонувати бессерверні обчислення. Наприклад, існувала модель програмування, що дозволяла забезпечити високий рівень багатокористувальної роботи, гнучку реакцію на змінне навантаження та API стандартизованого виклику функцій, загальний інтерфейс шлюзу (CGI), який навіть дозволяв прямо розгорнути вихідний код, написаний мовами високого рівня, такими як Perl або PHP [4]. Оригінальне рішення App Engine від Google, яке значною мірою було не оцінено ринком лише за кілька років до того, як серверні обчислення набули

популярності, також дозволяли розробникам розгорнути код, залишаючи більшість аспектів операцій хмарному провайдеру.

Однак, інші дослідники вважають, що Serverless є значним нововведенням порівняно з PaaS та іншими попередніми моделями. Сьогодні бессерверні обчислення з хмарними функціями відрізняються від своїх попередників кількома важливими характеристиками: автоматичним масштабуванням, ізоляцією, гнучкістю платформи та екосистемою підтримки обслуговування. Особливо видно відміну у автоматичному масштабуванні, яке вперше були запропоновано AWS Lambda. Даний сервіс відслідковував навантаження з набагато більшою достовірністю, ніж серверні методи автоматичного масштабування, швидко реагуючи на зміни навантаження, коли це потрібно, і зменшуючи ресурси до нульового обсягу та нульової вартості за відсутності запитів. Вартість обчислюється набагато простішим способом і забезпечує мінімальну одиницю виміру, що дорівнює 100 мілісекундам, на відміну від інших послуг автомасштабування, де ціна рахується за годину. Отже, ця послуга вимагала від клієнтів оплати часу фактичного виконання коду, а не часу, відведеного на ресурси, зарезервовані для програми. Ця різниця послужила стимулом для хмарних провайдерів ефективно розподіляти ресурси.

Компоненти безсерверної обчислювальної архітектури включають безсерверні обчислювальні платформи, контейнери API, засоби моніторингу системи, бази даних та сховища даних. Ці компоненти - це різноманітні послуги, що надаються та підтримуються провайдерами хмарних послуг. Серед них є обчислювальні платформи, які виконують програмний код для обробки ділової логіки, наприклад, доступу та маніпулювання базами даних. Засоби моніторингу (або системи реєстрації) збирають відповідні дані та інформацію про роботу вашої програми та забезпечують її доступність. І останнє, але не менш важливе: деякі послуги, які відповідають вашим бізнес-вимогам і можуть бути інтегровані у ваші програми, такі як електронна пошта, відстеження місцезнаходження, зберігання даних та аналітика. На рисунку 1.3 зображены ключові компоненти бессерверної обчислювальної архітектури.

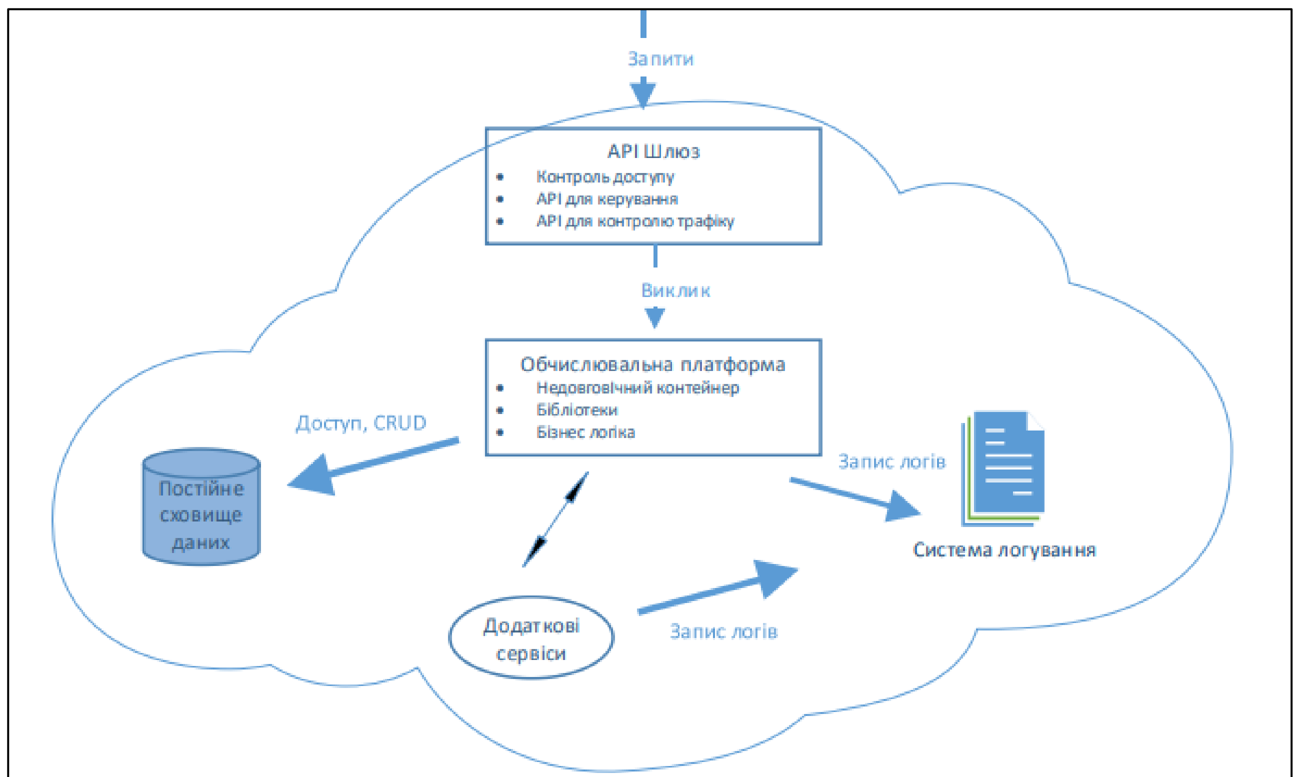

Рисунок 1.3 - Основні компоненти бессерверної архітектури

Запит до сервера здійснюється через виклик прикладного програмного інтерфейсу, після чого виклик перенаправляється до контейнера API [5]. Контейнер перевіряє, чи дозволений такий виклик, і витягує дані, що надсилаються в тілі, або параметру запиту, якщо виклик дозволений. Згодом викликається відповідна бессерверна функція для продовження обробки запиту. Функція опрацьовує вхідні дані на основі бізнес логіки, встановлює, якщо це необхідно, з'єднання з базою даних для отримання або маніпулювання даними, а потім готує відповідь користувачу. Тим часом детальна інформація, що стосується процесу, відстежується та реєструється в інструменті моніторингу роботи. Таким чином, кожний виклик може бути проаналізованим для розслідування певного інциденту. Крім того, інструмент моніторингу повинен мати можливість відслідковувати та зберігати дані про використання будь-яких допоміжних послуг та сервісів, які належать провайдеру бессерверних обчислень, це забезпечує повний контроль над помилками у додатку.

1.2 Аналіз переваг та недоліків бессерверної архітектури програмного забезпечення

Традиційна інфраструктура має недоліки, що зумовлюють інтерес до можливості заміни її на бессерверну архітектуру. Існує проблема необхідності розміщення додаткових ресурсів для обробки пікових навантажень традиційною інфраструктурою. Незважаючи на те, що переважну частину часу навантаження серверу є низьким, для обробки пікових навантажень необхідно тримати, а отже і оплачувати, значні обчислювальні ресурси. Різницю у ціні між традиційною та бессерверною архітектурами при пікових навантяженнях зображено на рисунку 1.4.

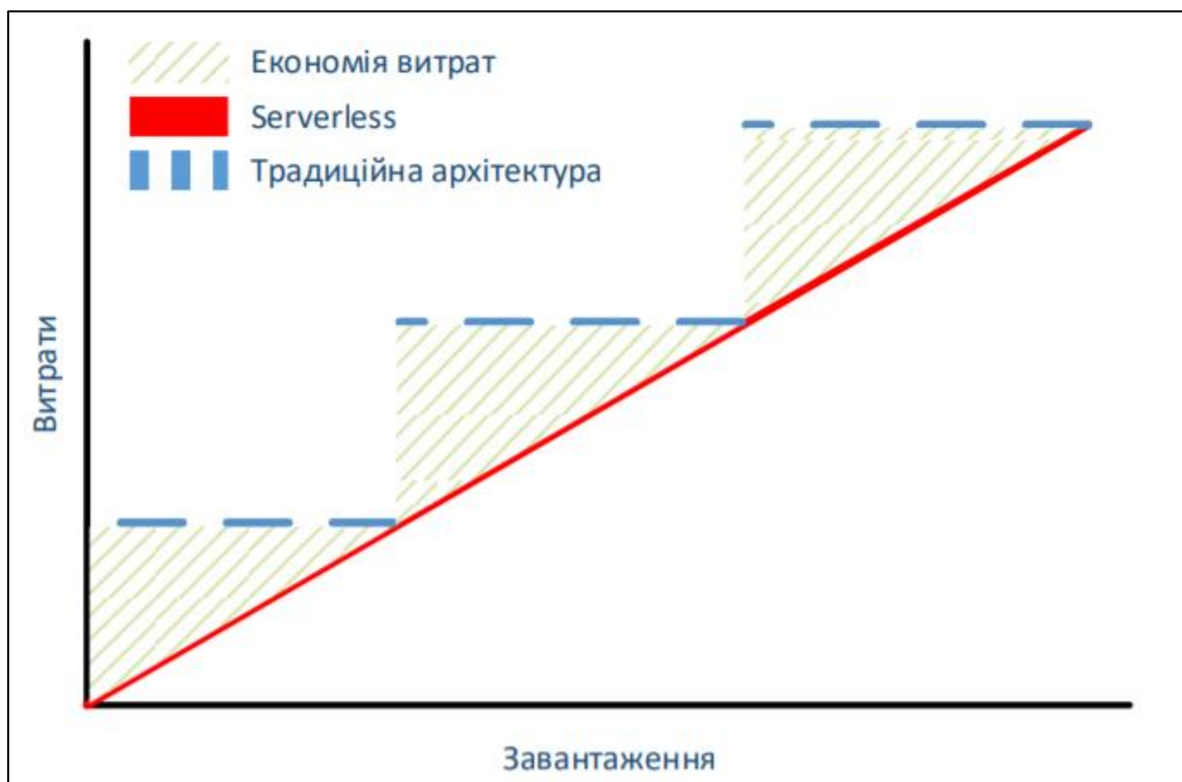

Рисунок 1.4 - Різниця у ціні між традиційною та бессерверною архітектурами при пікових навантяженнях

На будь-якій платформі бессерверних обчилень користувач просто записує хмарну функцію мовою високого рівня, вибирає подію, яка повинна спровокувати

виконання функції, наприклад, завантаження зображення у хмарне сховище або додавання мініатюри зображень у таблицю бази даних, та дозволяє Serverless системі обробляти все інше: вибір екземпляра, масштабування, розгортання, відмовостійкість, моніторинг, ведення журналів, виправлення безпеки та інше [6]. У таблиці 1.1 узагальнено відмінності між бессерверним та традиційним підходом для розробника.

Таблиця 1.1. - Порівняння традиційної та Serverless архітектури для розробника ПЗ

Параметр	Serverless	Традиційна хмара
Коли програма виконується	При настанні певної події	Постійно
Мова програмування	JavaScript, Python. Java, Go, C#	Будь-яка
Стан програми	Зберігається в тимчасовому сховищі (stateless)	Будь-де (stateful або stateless)
Максимальний розмір оперативної пам'яті	0.125 - 3 GiB (на розсуд розробника)	0.5 - 1952 GiB (на розсуд розробника)
Максимальний об'єм локальної пам'яті	0,5 GiB	0 - 3600 GiB (на розсуд розробника)
Максимальний час виконання програми	~900 секунд	Відсутній
Мінімальна одиниця виміру	0.1 секунд	60 секунд
Вартість одиниці виміру	$0.0000002	$0.0000867 - $0.4080000
Операційна система	На розсуд провайдера послуг	На розсуд розробника

Підсумовуючи, слід зауважити, що головними відмінностями між бессерверними і традиційними обчисленнями є:

- слабко зв'язані обчислення та зберігання. Масштабування зберігання та обчислення є окремим - резервується та оцінюється незалежно. Загалом, зберігання забезпечується окремою хмарною службою, а обчислення не має збереженого стану;
- виконання коду без управління ресурсами. Замість того, щоб запитувати ресурси, користувач надає фрагмент коду, а хмара автоматично надає ресурси для виконання цього коду;
- оплата здійснюється пропорційно використовуваним ресурсам замість ресурсів, що резервуються. Платежі залежать від певних параметрів виконання програми, наприклад, часу виконання, а не від потужності та кількості виділених віртуальних машин.

У той же час існує ряд наступних обмежень платформи Serverless, що необхідно враховувати при розробці веб-серверу:

- повністю Serverless додаток не підходить для програм у реальному часі, які використовують WebSockets, або подібні технології, оскільки функції FaaS мають обмежений термін служби;
- через деякий час бездіяльності функція повинна буде пройти “холодний” старт, який може зайняти до декількох секунд;
- залежність від постачальника. Різні постачальники послуг FaaS можуть відрізнятися в деяких особливостях використання своїх послуг, що перешкоджає переходу на іншого постачальника;
- на даний момент відсутні стандарти по розробці бессерверной функцій, таких як git стандарт, code review та інше;
- складність інтеграційного тестування;
- відсутність міжпроцесового стану;
- необхідно адаптувати архітектуру додатків і процеси розвитку бізнесу.

1.3 Порівняння політики ціноутворення та функціональних параметрів послуг основних провайдерів бессерверних обчислень

На даний момент на ринку є кілька компаній, що пропонують Serverless. Розглянемо найбільш поширені, особливо варто звернути увагу на потрібні нам елементи та їх особливості. Постачальників можна розділити на основні та другорядні групи. Основна група складається з найбільших публічних хмарних постачальників архітектури безсерверних обчислень:

- AWS Lambda. Пропозиція FaaS, що належить до веб-сервісів Amazon, була представлена в 2014 році. З моменту виходу Lambda стала синонімом того, що означає Serverless, займаючи позицію провідного продукту на ринку з найширшим спектром доступних послуг. Напевно, найвідомішим прикладом прийняття загальнодоступних серверів є Netflix;
- Azure Functions by Microsoft. Сервіс, запущений у 2016 році, конкурував з AWS Lambda. Azure Functions пропонує аналогічний як у Amazon набір послуг з акцентом на сімейство мов та інструментів Microsoft. Одним із прикладів використання функцій Azure є «https://haveibeenpwned.com/»;
- Google Cloud Functions (GCF). Один із чотирьох найбільших провайдерів, Google випустив своє рішення лише у 2017 році. Служба GCF раніше відставала від Azure та Lambda, але протягом 2018 року Google вдалося виправити попередні помилки, про що свідчать нотатки до випуску GCF.

Також варто згадати, що нові провайдери швидко інтегруються в цей простір, але, як і слід було очікувати, найбільші провайдери хмар мають великий набір географічних місць розташування та підтримують ресурси для розміщення бессерверних додатків. Але в майбутньому можна очікувати величезну конкуренцію на ринку.

Всі згадані провайдери пропонують подібні послуги, яких достатньо для запуска програми на керованій інфраструктурі. Вони також пропонують достатні

можливості, щоб отримати всі переваги концепції FaaS, але вони можуть бути різними. Щоб визначити найкращий варіант для певного користувача, порівняємо доступні послуги, використовуючи наступні критерії:

- моделі ціноутворення та коефіцієнти розрахунків;
- мови програмування, що підтримуються;
- типи тригерів функції;
- тривалість виконання та паралельність;
- методи розгортання;
- методи моніторингу та ведення журналів.

Ціноутворення провайдерів серверлесс зведені у таблицю 1.2.

Таблиця 1.2 - Порівняння ціноутворення основних провайдерів Serverless

	Безкоштовний обсяг, місяць	Ціна, GB/c
AWS Lambda	1 млн 400,000 GB/c	$0.000016
Azure Function	1 млн 400,000 GB/c	$0.000016
Google Cloud Functions	2 млн 400,000 GB/c	$0.0000004

На даний час Lambda дозволяє отримувати мільйон запитів і 400 терабайт-секунд часу обчислень в місяць. Запити вище цього порогового значення виставляються за ціною $ 0,00001667 / ГБ - це найнижча ціна в хмарному просторі.

Microsoft поставив ціну на Azure Functions, рівну Amazon і відповідну вільному рівню Lambda. Коли трафік додається до вартості, оцінки витрат фактично є найнижчими серед великих провайдерів для одного і того ж робочого навантаження.

Google Cloud Functions має найвигідніший план ціноутворення, по-перше цей провайдер має найнижчу ціну за використання одного гігабайту трафіку, а також надає найбільший безкоштовний обсяг використання на місяць, а саме 2 мільйони безкоштовних запитів чи 400 гігабайт трафіку.

Функціональні параметри основних провайдерів наведені в таблиці 1.3.

Таблиця 1.3 - Порівняння функціональних параметрів основних провайдерів

Параметр	AWS Lambda	Azure Functions	GCF
Мови, що підтримуються	Node.js (JavaScript), Python, Java, C#, Go	C#, F#, Python, Java, Nodejs, Python, PHP	Node.js 6, Node.js 8, Python 3.7
Гранульований IAM	Політика IAM може бути приєднана до Lambda	RBAC підтримується в групі підписки та ресурсів. Функції знаходяться всередині.	Ще не підтримується
Persistent Storage	Повністю Serverless. S3 та DynamoDB	Змінні середовища, сховище blob	Cloud Storage, Cloud Datastore, Cloud SQL
Тип тригера	Широкий спектр AWS Services + API шлюзу	Microsoft service requests + HTTP webhooks, APIM, Function proxy and bindings	HTTP + Cloud Pub/Sub + Cloud Storage + Direct Triggers
Ведення журналу та моніторинг	Cloudwatch Logs & X-Ray	Azure Storage & App Insights	Stackdriver
Максимальний час виконання за запит, секунд	900	За замовчуванням 300 Преміум 600	За замовчуванням 60, до 540
Паралельне виконання, паралельних функцій	1000	Без обмежень (залежить від тригерів)	Немає обмеження для виконання HTTP. Для інших: 1000
Розгортання	Завантаження архіву до Lambda/S3, Serverless Framework, вбудоване редагування коду	Git, dropbox, visual studio, Kudu console, One Drive, завантаження архіву	CLI, завантаження архіву, вбудоване редагування коду, Cloud Storage
Максимальна кількість функцій	Без обмежень	Без обмежень	1000 функцій на проект
Залежності	Пакети розгортання на кожну мову	package.json, npm, nuget	npm package.json, pip requirements.txt
Масштабованість та доступність	Автоматичне масштабування	Ручне (App Service Plan) або швидке автоматичне масштабування (Consumption Plan)	Автоматичне масштабування

На сьогоднішній день AWS Lambda, Azure Functions та Google Cloud Functions готові до використання у серйозних проектах та є загальнодоступними. AWS Lambda стала загальнодоступною на початку 2015 року, Azure Functions наприкінці 2016 року, тоді як Google Cloud Functions, зовсім недавно, у липні 2018 року.

З такою великою різницею в часовій шкалі, очевидно, що AWS Lambda домінує над безсерверною архітектурою за рахунок величезної спільноти розробників та широким спектром відомих кейсів використання. Більше того, він має розширені функції, такі як AWS Step Functions [7].

1.4 Serverless бази даних

На даний момент існує три основних типи БД.

Перший тип – класичні OLTP бази: PostgreSQL, SQL Server, Oracle, MySQL. Вони написані давним-давно, але як і раніше актуальні, тому що добре знайомі спільноті розробників.

Другий тип – NOSQL [8]. Ці бази даних намагалися піти від класичних шаблонів шляхом відмови від SQL, традиційних структур і ACID, за рахунок додавання вбудованого шардірования. Наприклад, це Cassandra, MongoDB, Redis або Tarantool. Всі ці рішення хотіли запропонувати ринку щось принципово нове і зайняли свою нішу, тому що в певних задачах виявилися вкрай зручними.

Третій тип - managed бази. У цих баз ядро теж саме, що і у класичних OLTP баз або нових NoSQL. Але у них немає потреби в DBA і DevOps, вони адміністрюються на керованому сервері в хмарах. Це дозволяє розробнику не замислюватися про конфігурацію, адміністрацію баз даних, а просто використовувати її як джерело даних.

Незважаючи на те що база даних - це statefull сервіс, який не підходить для для serverless інфраструктури. При цьому, і state у бази даних теж є: гігабайти, терабайти, а в аналітичних базах навіть петабайт.

З іншого боку, практично всі сучасні бази - це величезна кількість логіки і компонентів: транзакції, узгодження цілісності, процедури, реляційні залежності і багато логіки. Гігабайти і терабайти безпосередньо використовуються тільки невеликою частиною логіки бази даних, пов'язаної з безпосереднім виконанням запитів.

Відповідно, виникла ідея: якщо частина логіки допускає stateless виконання, чому б не розділити базу на Statefull і Stateless частини. Подивимося, як можна розділити бази даних на Statefull і Stateless частини на практичних прикладах.

Наприклад, є аналітична база даних (див. рис. 1.5): зовнішні дані, ETL-процес, який завантажує дані в базу, і аналітик, який відправляє до бази SQL-запити.

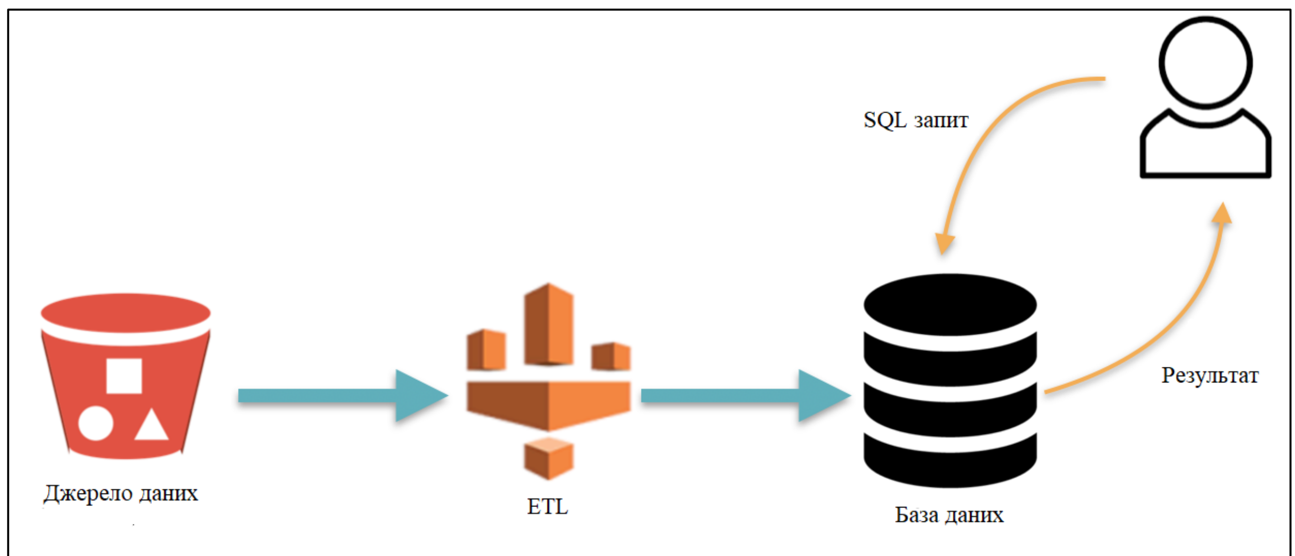

Рисунок 1.5 – Схема аналітичної бази даних

Це класична схема роботи сховища даних. У цій схемі, умовно, один раз виконується ETL. Далі потрібно весь час платити за сервери, на яких працює база з даними, в яких знаходиться ETL.

Розглянемо альтернативний підхід, реалізований в базі AWS Athena Serverless. Тут немає постійно виділеного сервера, на якому зберігаються завантажені дані. Замість цього використовується схема, що наведена на рисунку 1.6.

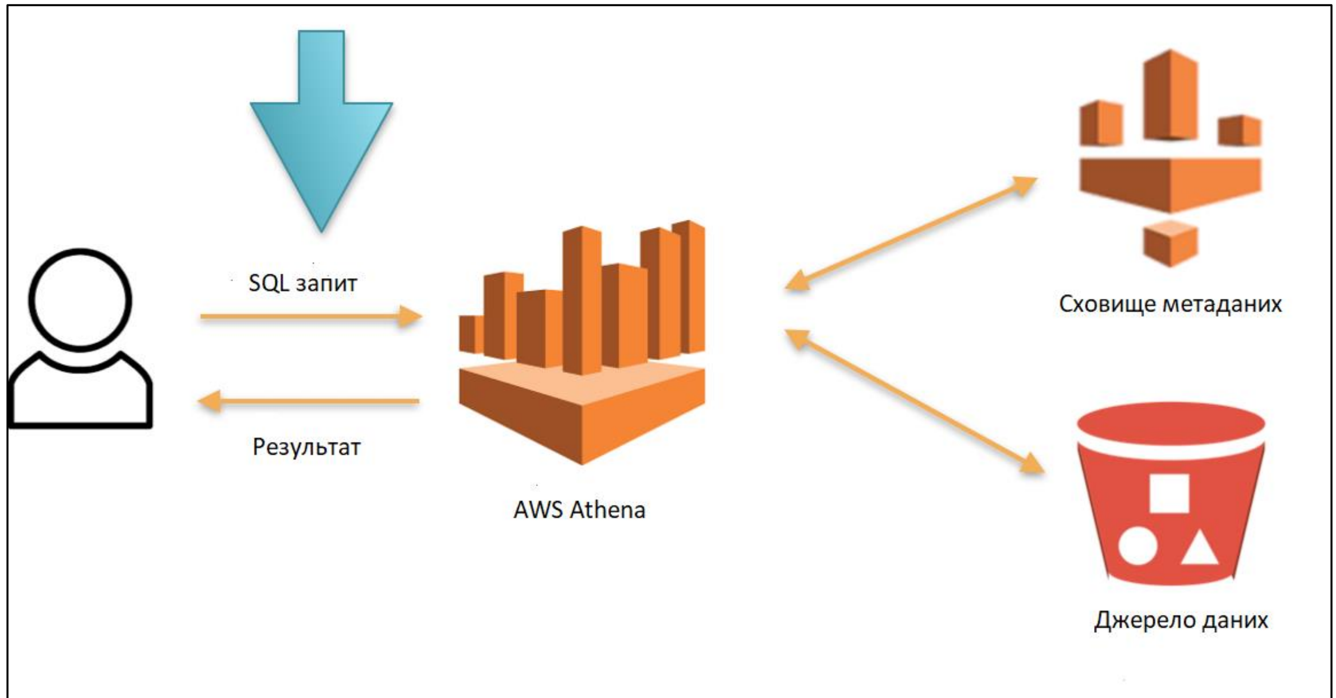

Рисунок 1.6 – Схема Serverless аналітичної бази даних

Користувач відправляє SQL-запит до Athena [8]. Оптимізатор Athena аналізує SQL-запит і шукає в сховище метаданих (Metadata) конкретні дані, потрібні для виконання запиту. Оптимізатор, на основі зібраних даних, вивантажує потрібні дані із зовнішніх джерел у тимчасове сховище (тимчасову базу даних). У тимчасовому сховищі виконується SQL-запит від користувача, результат повертається користувачеві. Тимчасове сховище очищується, ресурси вивільняються.

Завдяки такому підходу, користувач платить тільки за процес виконання запиту, тож якщо на сервер бази даних немає запитів, то користувачу непотрібно платити.

1.5 Особливості бессерверної обробки зображень

Якщо спостерігати за процесом попередньої обробки зображень, можна зрозуміти, що в більшості випадків немає необхідності в його виконанні безпосередньо на сервері додатка. Це особливо актуально, якщо здійснюються одні і ті ж перетворення, які не вимагають інших даних, через саме зображення.

Завдання з попередної обробки зображення можна відокремити від основної логіки додатка. Для цього можна використовувати Serverless функцію, яка буде брати вихідне зображення в якості вхідних даних і виконувати всі необхідні перетворення.

Створення різних версій зображення з одного вихідного починається з завантаження оригіналу в сховище. Потім запускається функція, яка відповідає за створення нових версій і їх повторне завантаження в сховище. Більш детально процес виглядає наступним чином:

- зображення завантажуються в певну папку всередині сховища;
- кожен раз, коли в цю папку завантажується нове зображення, сервіс відправляє повідомлення з ключем створеного об'єкта в бессерверну функцію [9];
- функція зчитує повідомлення і використовує цей ключ для вилучення нового зображення;
- функція обробляє зображення, виконуючи необхідні перетворення, і потім завантажує його назад в сховище;
- оброблені зображення можуть бути відображені користувачам.

У кожного провайдера цей алгоритм може відрізнятися в деталях, можуть використовуватися різні тригери, також кожен провайдер надає специфічні інструменти, такі як сховища, сервіси посилання повідомлень між мережевими компонентами, api gateway, бази даних та інше. Всі ці інструменти і підходи будуть проаналізовані в наступних розділах.

Також треба зазначити, що при створенні розподіленої системи обробка зображень стає менш тривіальною. За перших ознак збільшення масштабу це може стати справжнім больовим фактором у застосуванні. Причини складності обробки зображень у розподілених системах наступні:

- встановлення з'єднання кількох запущених вузлів до одного постійного сховища неможливо, тому користувачу доведеться турбуватися про уникнення пошкоджень даних під час запису;
- зберігання та обслуговування суттевої кількості зображень на диску часто спричиняє помилки та потребує постійних ресурсів;
- маніпулювання зображеннями (зміна розміру, обрізання, автоорієнтація) вимагає значних комп'ютерних потужностей. У розподіленій системі важливо, щоб вузли мали максимальну пропускну здатність.

Всі ці складності вирішують провайдери бессерверних обчислень. Обробка зображень є загальним випадком використання бессерверних технологій. AWS навіть має свій окремий сервіс для обробки зображень, який називається «Serverless Image Handler», але інші постачальники також мають інстументарій для роботи з зображенями.

При реалізації обробки зображень за допомогою бессерверних обчислень в першу чергу треба виділити основні три крока:

- створення кінцевої точки, для отримання зображення в повному розмірі та збереження його у сховищі даних;
- створення бессерверної функції для обробки зображень, після того як зображення було завантажено до сховища;
- оновлення сховища після того, як зображення було оброблено, та надсилання повідомлення про завершення обробки.

Такий підхід є зручним, тому що не вимагає від розробника додаткових знань про операційні складові та є досить масштабованим.

1.6 Постановка задачі

Здійснений аналіз предметної області дозволив конкретизувати мету роботи, яка полягає у визначенні найбільш ефективних методів та засобів розробки бессерверних додатків для обробки зображень, виявлені переваг і недоліків цих методів та порівнянні між собою різних провайдерів бессерверних обчислень з урахуванням потреб бізнесу. Також необхідно проаналізувати інфраструктуру описаних провайдерів, порівняти між собою різні сховища, бази даних, тригери, тощо.

Для вирішення поставленої задачі необхідно:

- проаналізувати методи та засоби розробки бессерверних додатків для обробки зображень;
- визначити критерії оцінювання ефективності методів та засобів розробки бессерверних додатків;
- розробити методику дослідження ефективності;
- розробити програмну систему для дослідження ефективності методів та засобів розробки бессерверних додатків та провести її тестування;
- провести оцінювання ефективності методів та засобів розробки бессерверних додатків відповідно до запропонованої методики за допомогою розробленої програмної системи;
- узагальнити отримані результати та розробити рекомендації щодо використання методів та засобів розробки бессерверних додатків для обробки зображень.

2 АНАЛІЗ ІНФРАСТРУКТУРИ БЕССЕРВЕРНОЇ АРХІТЕКТУРИ

2.1 Принцип роботи бессерверних функцій

Коли спрацьовує тригер, наприклад, «об'єкт завантажений у сховище даних», провайдер бессерверних обчислень ініціює відповідну функцію, якщо вона має реагувати на цей тригер. При виконані функції передаються дані про тригер, щоб функція могла розшифрувати те, що спричинило тригер, а саме джерело події. Також бессерверна функція також може викликати інші API [10]. Це можуть бути внутрішні API провайдеру або зовнішні. Функція навіть може викликати іншу бессерверну функцію. Закінчивши виконувати свою логіку, бессерверна функція повідомляє про виконанння. Декілька тригерів призводять до відповідної кількості викликів бессерверних функцій. Це дозволяє розробнику зосереджується на логіці всередині функції та зберегти спеціалізацію функції, використати мінімальний час виконання та не використати часу більше, ніж це дозволено інфраструктурою.

Така модель найкраще працює в умовах відсутності стану, що необхідно зберігати між викликами бессерверних функцій. Можливо, однак, тримати стан поза межами провайдеру, для цього можна використовувати стороню службу, наприклад, Shared Memcache [11]. На рисунку 2.1 наведена схема роботи Google Cloud Function з офіційної документації.

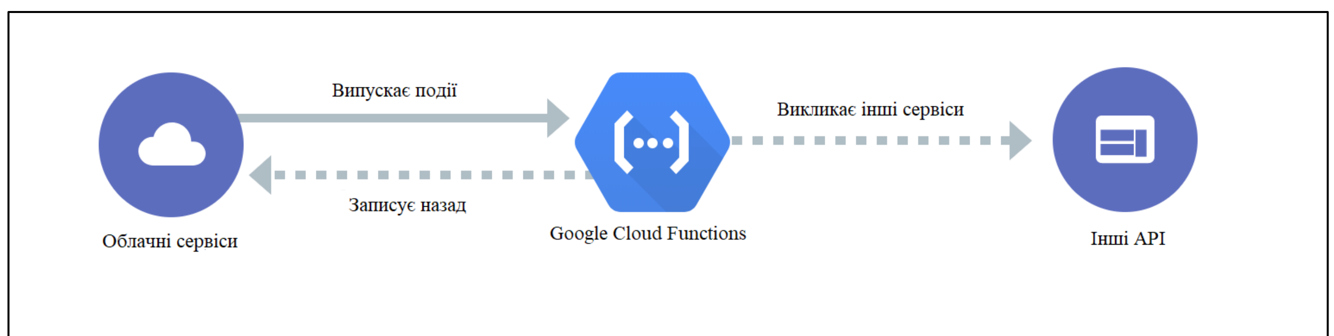

Рисунок 2.1 - Реалізація Google Cloud Functions

Зліва розміщено облачні сервіси, що є хмарною платформою Google та різні її сервіси, наприклад Google Cloud Storage, Cloud Datastore, Google Cloud Pub/Sub, Stackdriver, тощо. Ці сервіси мають тригери, що відбуваються всередині них. Наприклад, якщо до сховища завантажено новий об’єкт, видалено об’єкт або певні метадані оновлено. Аналогічно, тригер відбувається, якщо до логів Stackdriver або Cloud Pub/Sub [12] отримано повідомлення, опубліковане в певній темі. Не всі тригери підтримуються на даний момент у Google Cloud Functions.

Нижче наведено основні тези про події, тригери та дані про події:

- події: вони трапляються в сервісах Google Cloud Platform, наприклад, «Файл, завантажений у сховище», «повідомлення, опубліковане у черзі», «прямий запит HTTP» тощо;
- тригери: користувач може налаштувати функції відповідати на певні тригери. Тригер - це дані, що відносяться до події;
- дані про подію: це дані, які передаються бессерверній функції, коли тригер події призводить до виконання функції.

Приклад тригеру зображено схематично на рисунку 2.2.

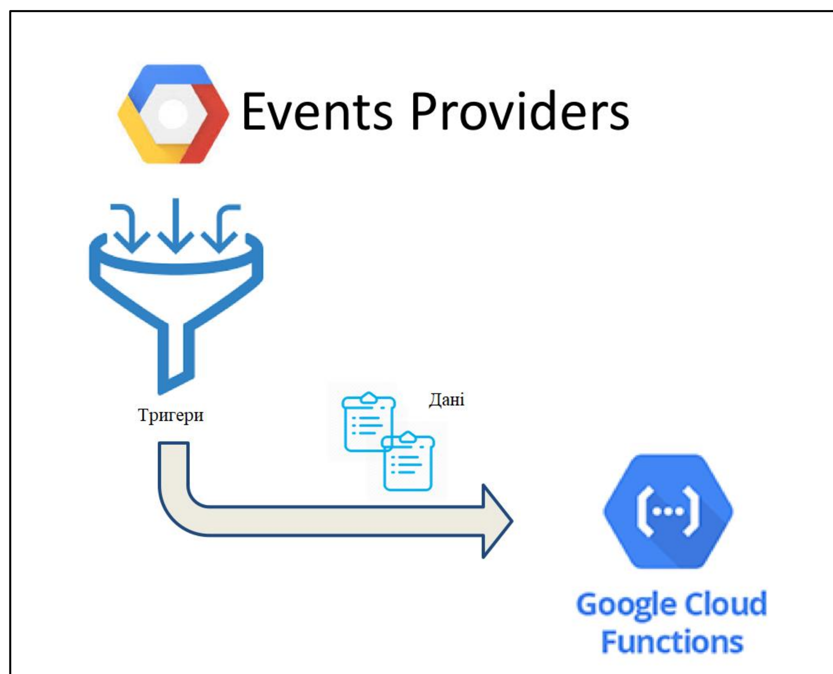

Рисунок 2.2 - Потік даних в Google Cloud Functions

На даний момент Google Cloud Functions підтримує таких постачальників подій:

- HTTP - виклик функції безпосередньо через HTTP-запити:
- Cloud Storage;
- Cloud Pub/Sub;
- Firebase (DB, Analytics, Auth);
- Stackdriver;
- Cloud Firestore (бета);
- Google Compute Engine (Альфа);
- BigQuery (Alpha).

Також зараз підтримуються два типи функцій Google Cloud:

- функції переднього плану (синхронні): викликаються безпосередньо через кінцеву точку HTTP, яка надається для функції. Вони також відомі як HTTP-тригер функції;
- фонові функції (асинхронні): вони опосередковано викликаються через подію, яка запускає функцію.

2.2 Метадані бессерверних функцій

Бессерверна функція пов'язана з низкою сутностей на платформі, включаючи її метадані, код, запущені контейнери та "теплу чергу". Метадані функції є джерелом істинності існування функції і визначаються чотирма полями [13]:

- ідентифікатор функції - це випадково сформований GUID, який надається функції під час її створення. Ідентифіктор використовується для ідентифікації та пошуку функціональних ресурсів;

- мова виконання – це мова програмування, на якій реалізована бессерверна функція;
- розмір пам'яті – обсяг пам'яті функції визначає максимальний обсяг пам'яті, який може споживати контейнер функції. На даний момент максимальний обсяг оперативної пам'яті встановлений на рівні 1 Гб. Ядра процесора, призначені контейнеру функції, встановлюються пропорційно розміру пам'яті;
- код Blob URI – під час створення функції надається zip-архів, що містить код функції. Цей код копіюється в BLOB-пам'ять всередині облікового запису сховища платформи, а URI цього BLOB-файлу розміщується в метаданих функції.

Контейнери функцій будуть детально обговорені нижче, як і теплі черги, які є індексованими ідентифікаторами функції та містять доступні запущені контейнери кожної функції.

2.3 Виконання бессерверної функції

Функції виконуються шляхом виклику маршруту ”/invoke” з ресурсів функцій на REST API. Тіло запиту виклику надається функціям як входи, а тіло відповіді містить виходи функції. Виконання починається у веб-службі, яка отримує виклик та згодом отримує метадані функції із сховища таблиці. Створюється об'єкт запиту на виконання, що містить метадані функції та вхідні дані, після чого веб-служба намагається знайти доступний контейнер у службі воркера для обробки запиту на виконання.

Взаємодія між Інтернетом та робочими службами контролюється за допомогою спільного рівня обміну повідомленнями. Зокрема, існує глобальна "холодна черга", а також "тепла черга" для кожної функції на платформі. Ці черги містять доступні контейнери повідомлень, які складаються з URI, що містить

адресу екземпляра робочії служби та ім'я доступного контейнера. Повідомлення в холодній черзі вказують на те, що робоча служба має нерозподілену пам’ять, в якій вона може запустити контейнер, а видимі повідомлення в теплій черзі функції вказують на існуючі контейнери функцій, які в даний час не обробляють запити на виконання [14].

На рисунку 2.3 схематично зображено алгоритм обробки виклику функції.

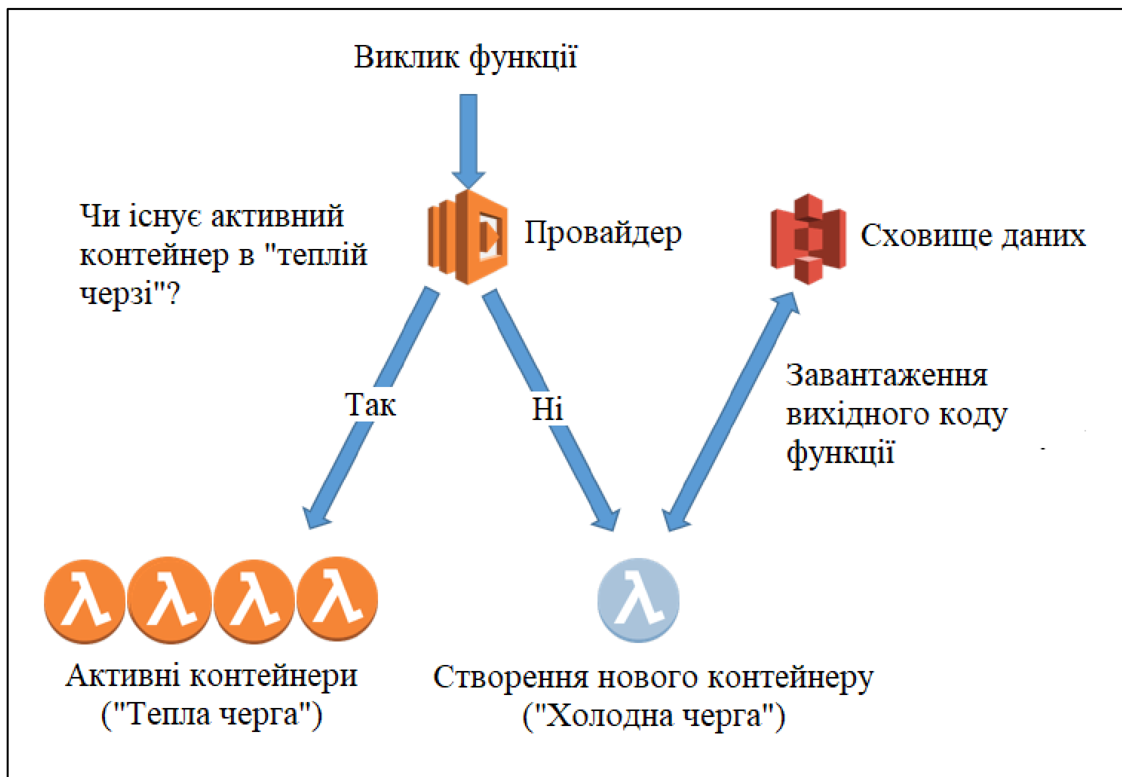

Рисунок 2.3 – Схема алгоритма обробки виклику функції

Веб-служба спочатку намагається обробити повідомлення з теплої черги функції. Якщо повідомлень не знайдено, веб-служба виводить із холодної черги повідомлення, яке буде оброблятися контейнером функції. Якщо всі робочі служби повністю забезпечені запущеними контейнерами, холодна черга буде порожньою.

Отже, якщо веб-служба не може знайти доступний контейнер як у теплій, так і в холодній черзі, вона поверне статус «HTTP 503 Service Unavailable», оскільки немає ресурсів для виконання запиту. З цієї причини глибина холодної черги є головною метою автоматичного масштабування, оскільки вона відображає обсяг виконання викликів на платформі.

Як тільки повідомлення буде знайдено в черзі, веб-служба надсилає HTTP запит до робочій службі за допомогою URI, що міститься в повідомленні. Потім робоча служба виконує функцію і повертає результат веб-службі, яка, у свою чергу, відповідає на виклик.

2.4 Незмінність бессерверних функцій

Бессерверні функції незмінні, а саме в REST API підтримуються лише операції створення, видалення та читання. Незмінність є бажаним атрибутом для логіки виконання, але насправді корисно мати багато версій функції та оновлювати існуючий код функції. Цю поведінку можна підтримати, розглядаючи версії функцій як об'єкти, де функції можуть бути представлені як поточна версія та список ідентифікаторів минулих версій, а версії функцій можуть зберігати інформацію, що на даний момент міститься у функції метадані. Запити на виконання за замовчуванням будуть націлені на ідентифікатор конкретної версії або останню версію, а оскільки остання версія включена в об’єкт функції верхнього рівня, для отримання метаданих все одно потрібно буде прочитати лише одну таблицю. Отже, хоча оновлення функції було б корисним доповненням до платформи, підтримка декількох версій функцій не вплине негативно на продуктивність виконання.

2.5 Генерація контейнерів

Кожна робоча служба управляє пулом нерозподіленої пам'яті, яку вона може призначити функціональним контейнерам. Коли пам’ять резервується, генерується ім’я контейнера, яке однозначно ідентифікує контейнер та його пам’ять та

вбудовується в URI, що надсилається у повідомленнях про розподіл контейнерів. Отже, кожне повідомлення в чергах можна однозначно ідентифікувати і може бути пов’язане з певним контейнером в екземплярі робочої служби. Пам'ять розподіляється порівну, і робочі служби припускають, що всі функції будуть споживати виділений обсяг пам'яті.

Коли згенеровані контейнери надсилаються до холодної черги, вони ще не призначені функції. Щоб робоча служба не використовувала зайвий пул пам'яті, передбачається, що призначена функція матиме максимальний обсяг пам'яті функції. Потім, коли робоча служба отримує запит на виконання розподілу пам'яті, вона зменшує обсяг пам'яті для функції, якщо призначена функція вимагає менше максимального розміру. Після створення контейнера та виконання його функції вперше, повідомлення про генерацію контейнера поміщається в теплу чергу цієї функції.

2.6 Видалення контейнера

Є два способи видалення контейнера. По-перше, коли функція видаляється, веб-служба видаляє теплу чергу функції, яка періодично контролюється на наявність екземплярів робочої служби, що містять контейнери цієї функції. Якщо робоча служба виявляє, що черга видалених функцій вичерпана, вона видаляє запущені контейнери цієї функції та повертає їх пам'ять до загального пулу. По-друге, в реалізації контейнер може бути видалений, якщо він не працює протягом довільно встановленого періоду, після чого він видаляється і його пам'ять відновлюється. Щоразу, коли пам’ять вилучається, робоча служба надсилає нові розподіли контейнерів до холодної черги, якщо їх невикористана пам’ять перевищує максимальний обсяг пам’яті функцій.

Закінчення строку дії контейнера має наслідки для веб-служби, оскільки можна виключити контейнер з вичерпаним терміном дії з гарячої черги функції. У

цьому випадку, коли веб-служба надсилає запит на виконання, робоча служба поверне HTTP статус «404 Not Found». Потім веб-служба видалить повідомлення з черги та закінчить спробу.

2.7 Образ контейнера

Платформа використовує Docker для запуску контейнерів Windows Nano Server та взаємодіє зі службою Docker через API Docker Engine [15]. Зображення контейнера побудовано для включення середовища виконання функції та обробника виконання. На зображенні відсутній будь-який код функції. Спеціальні контейнери не будуються для кожної функції на платформі, натомість при запуску контейнера приєднується лише функція читання, що містить код функції. Підхід з одним зображенням був обраний з кількох причин: простіше керувати одним зображенням, приєднання томів - це швидка операція, а зображення контейнерів Windows Nano Server значно більше, ніж легкі образи Linux, такі як Alpine Linux, що впливає на витрати на зберігання та час запуску. На додаток до обсягу лише для читання, розмір пам'яті та відсоток процесора контейнера пропорційно встановлюються залежно від обсягу пам'яті функції.

Обробник виконання контейнера - це простий сервер Node.js, який отримує вхідні дані функції від робочої служби. Робоча служба надсилає вхідні дані функції до обробника в тілі HTTP-запиту, обробник викликає функцію із зазначеними входами і відповідає робочій службі виходами функції. Контейнер можна адресувати в локальній мережі робочої служби, оскільки контейнери додаються до мережі "nat" за замовчуванням, яка є еквівалентом мережі мережі "Bridge" контейнера Linux.

2.8 Асинхронне виконання

В даний час прототип підтримує лише синхронні виклики. Іншими словами, запит на виконання функції поверне результат її виконання, він не просто запустить функцію і повернеться. Асинхронні виконання самі по собі прості в підтримці, веб-служба може просто відповісти на виклик, а потім обробити запит на виконання. Складність асинхронного виконання полягає в тому, щоб гарантувати виконання принаймні один раз. Важливо розуміти, що синхронне або асинхронне виконання гарантується лише після повернення запиту на виклик із успішним кодом стану. Отже, подальша робота не потрібна для синхронних запитів на виконання (як у реалізації), оскільки успішний код стану повертається лише після завершення виконання. Однак асинхронні виконання відповідають клієнтам перед виконанням функції, тому необхідно мати додаткову логіку, щоб забезпечити успішне виконання коду.

Прототип може підтримати цю вимогу, зберігаючи активні виконання у наборі черг та вводячи третю службу, відповідальну за моніторинг стану цих повідомлень черги. Робочі служби постійно оновлюватимуть затримки видимості повідомлень під час виконання функції, а служба моніторингу виявлятиме збої шляхом пошуку видимих повідомлень. Такий підхід забезпечує обробку помилок виконання платформи, а не винятків, викинутих функцією під час виконання, для яких також може знадобитися повторна спроба.

2.9 Контейнери для Windows

Контейнери Windows мають деякі обмеження порівняно з контейнерами Linux, здебільшого тому, що контейнери Linux були розроблені навколо груп Linux, які підтримують корисні операції, недоступні в Windows. Найбільш

помітним у контексті бессерверних обчислень є підтримка оновлення ресурсів контейнера та пауза контейнера. Поширеним шаблоном у реалізації бессерверної платформи є призупинення контейнерів в режимі очікування, щоб запобігти споживанню ресурсів, а потім відміна їх перед відновленням виконання.

Ще однією потенційно корисною операцією є оновлення ресурсів контейнера. Оскільки платформа резервує ресурси для контейнерів перед початком виконання, було б корисно для холодного запуску, якби контейнери запускалися до того, як вони були призначені функції і можна було б змінити розмір контейнера після отримання запиту на виконання. У майбутній роботі можна вивчити, як підтримувати цю семантику в контейнерах Windows, можливо, обмежуючи або оновлюючи ресурси до самого процесу функції, а не до контейнера в цілому. Як варіант, прототип може експериментувати з контейнерами Linux, щоб порівняти продуктивність запуску та перевірити життєздатність зміни розміру контейнера під час холодних пусків.

2.10 Безпека

Безпека бессерверних систем - також відкрите питання дослідження. Розміщення довільного коду користувача в контейнерах в системах із декількома користувачами є небезпечним, і при розробці та запуску функціональних контейнерів потрібно бути обережним, щоб запобігти уразливостям. Цей перетин віддалених викликів процедур (RPC) та безпеки контейнера представляє важливий реальний тест загальної безпеки контейнера. Отже, хоча бессерверні платформи здатні створювати контейнери функцій та довільно обмежувати права функцій, збільшуючи шанси безпечного виконання, необхідні подальші дослідження для оцінки безпеки в середовищах виконання функції.

Крім того, як і у багатьох хмарних сервісах, слід враховувати можливість побічних атак. Це особливо небезпечно при розгляді реалізації кластера Redis,

оскільки кластер Redis використовує відому стратегію сегментування для розподілу елементів. Тому зловмисник може пошкодити роботу функції з відомим ідентифікатором, створюючи значне навантаження на функцію з однаковим хешем CRC-16. Цього можна уникнути, якщо використовувати надійно зашифровані ідентифікатори функцій як ключі Redis, щоб усунути можливість колізій. Подібні запобіжні заходи слід вжити, щоб забезпечити неможливість використання розміщенних екземплярів функцій третьою особою.

2.11 Логування і налагодження бессерверних додатків

Монолітні додатки поступово втрачають популярність і поступаються бессерверним додаткам, з точки зору простоти налагодження вони безумовно домінують. Налагодження монолітних додатків проводиться локально, в інтегрованому середовищі розробки, при цьому розробник створює точку зупину в будь-якому рядку коду [16]. Оскільки IDE-системи поставляються вже з готовим режимом налагодження — це все, що потрібно від розробників для виявлення помилок в коді. У бессерверній архітектурі, навпаки, найбільш популярним методом налагодження є логування. Розробники відстежують і записують всі дії сервісу в лог-файли і таким чином отримують уявлення про поведінку програми.

Бессерверні додатки управляються подіями. Це означає, що бессерверні функції можуть звертатися до різних сервісів в різні моменти часу, у міру необхідності. Незважаючи на те, що деякі сервіси можна змоделювати локально, налагодження таких додатків, як правило, включає повну перевірку наявних логів.

Таким чином, налагодження бессерверних додатків грунтуються на двох різних підходах з використанням Google Cloud Functions:

- локальне налагодження бессерверних додатків;
- налагодження бессерверних додатків у хмарі.

При розробці програмного забезпечення найважливіша частина налагодження відбувається при локальному запуску. Модель бессерверних додатків — це рішення, яке дозволяє розробникам запускати свої додатки локально на бессерверній платформі провайдера. У пакет рішення входять два компоненти: специфікація шаблону і інтерфейс командного рядка. Специфікація шаблону дозволяє визначити програму за допомогою прав доступу, подій, API-інтерфейсів і функцій. При цьому інтерфейс командного рядка дозволяє викликати функції локально, здійснювати налагодження функцій покроково, а також упаковувати і розгортати додатки в хмарі.

Після того, як додаток було розгорнуто у хмарі, необхідно зрозуміти, які проблеми виникають при підключенні функцій до сервісів. Деякі з цих підключень зазвичай неможливо змоделювати в локальному середовищі тестування, тому потрібна розробка процедури для тестування в реальному часі. Нижче наведено кроки найбільш популярної процедури тестування в реальному часі:

- перший крок — тестування API-шлюзу. Найбільш популярні інструменти для тестування кінцевих точок API і шлюзів: SoapUI, Postman та Curl;
- другий крок — тестування бессерверних функцій безпосередньо в консолі, без використання джерела подій. Цей крок допоможе з легкістю діагностувати проблеми в роботі функцій при першому розгортанні програми. На цьому етапі також потрібно створити тестові події, які емулюють сервіси провайдеру;
- на наступному кроці вивчається поведінка програми за допомогою логування. Наприклад, функції AWS Lambda за замовчуванням налаштовані на відправку логів в CloudWatch. Такий сервіс є у кожного провайдера;
- на заключному етапі можна провести аналіз роботи і налагодження програми за допомогою X-RAY сервісу. Це рішення забезпечує наскрізний моніторинг подій і створює графічну візуалізацію всіх компонентів програми.

3 КРИТЕРІЇ, МЕТОДИКА ТА РЕЗУЛЬТАТИ ОЦІНЮВАННЯ ЕФЕКТИВНОСТІ МЕТОДІВ ТА ЗАСОБІВ РОЗРОБКИ БЕССЕРВЕРНИХ ДОДАТКІВ ДЛЯ ОБРОБКИ ЗОБРАЖЕНЬ

3.1 Методика та критерії ефективності методів та засобів розробки бессерверних додатків для обробки зображень

Методика оцінювання ефективності методів та засобів розробки бессерверних додатків полягає у наступному:

- визначені критеріїв оцінювання ефективності;
- розробки процедури моделювання функціональних параметрів провайдерів бессерверного обчислення;
- оцінюванні наступних параметрів: масштабованості, затримки холодного старту, швидкості обробки запитів;
- розробки рекомендації щодо вибору провайдера бессерверного обчислення та удосконалення окремих функцій провайдера.

Для визначення критеріїв оцінювання ефективності наведемо значення терміну ефективності.

Ефективність — здатність ПЗ забезпечувати необхідний рівень продуктивності у відповідність з виділеними ресурсами, часом і іншими позначеними умовами.

Таблиця 3.1 Метрики ефективності

Параметр	Одиниці виміру	Пояснення параметру
Затримка	Час (мілісекунди)	Затримка перед виконанням запиту
Масштабованість	Кількість виконань / секунда	Здатність системи підтримувати нагрузку
Затрати на обслуговування	людино-години	Затрати на розгортання та підтримання інфраструктури

Ефективність бессерверних додатків напряму залежить від продуктивності, а саме здатності програмного забезпечення при заданих умовах забезпечувати необхідну працездатність стосовно виділюваних для цього ресурсів.

Продуктивність вимірює наскільки швидкою є відповідь бессерверної функції, при виконані тригеру. Тригери можуть потрапляти до провайдеру в різних моделях, які можна охарактеризувати як періодичні або стохастичні. Користуючись даним визначенням, було розроблено список метрик, які наведені у таблиці 3.2 [17].

Таблиця 3.2 Метрики продуктивності

Метрика	Одиниці виміру	Пояснення параметру
Час відповіді	Час (мілісекунди)	Час від надсилання запиту до отримання відповіді
Пропускна здатність	запит/секунда	Кількість запитів, які обробляються додатком за одиницю часу
Використання ресурсів	відсоток	Використання ресурсів, таких як центральний процесор, пам'ять, пропускна здатність мережі

Основна інфраструктура та деталі реалізації комерційних платформ FaaS часто приховані від користувача. Це робить платформи FaaS чорними ящиками та підкреслює важливість тестів ефективності на цих платформах. У попередньому розділі був проведений аналіз метрик продуктивності платформ FaaS та функцій, але безсерверні платформи розвиваються та оновлюються кожен день, загрожуючи актуальності деяких досліджень в цій темі.

Холодний старт та пом’якшення його наслідків є постійною темою досліджень. Аналіз показав, що холодний старт може мати значний вплив на затримку, і що ступінь затримки також залежить від постачальника хмарних служб та мови програмування.

Затримка, сприйнята користувачем, є важливою частиною продуктивності та може вплинути на використання веб-програми. Чим більша затримка, тим рідше користувачі натикаються на розроблений продукт у веб-браузері. Експерти стверджують, що затримка менше 500 мс не помітна, але якщо затримка перевищує 1000 мс, дуже імовірно, що користувачі помітять таку затримку [18].

Таким чином, процедура моделювання функціональних параметрів провайдерів бессерверних обчислень полягає в наступному:

- моделювання показників масштабування бессерверних функцій;
- моделювання показників затримки холодного старту бессерверних функцій;
- моделювання показників швидкості обробки запитів бессерверних функцій;
- моделювання показників масшатабованості бессерверної архітектури в порівнянні з монолітною архітектурою.

3.2 Результати оцінюванння ефективності методів та засобів розробки бессерверних додатків

Автором був розроблений прототип бессерверної обчислювальної платформи, орієнтований на оцінювання показників продуктивності: масштабованість, затримка холодного старту та швидкість обробки запитів. Прототип програмної системи симулює роботу провайдера для перелічених показників продуктивності.

Функціональні параметри вимірювалися за допомогою тестової функції, яка змінює розмір і проводить компресію зображень. Ця функція викликається синхронно з подіями чи HTTP тригерами, як це підтримується різними платформами, а також шляхом виклику функції провайдера. Ручні виклики не

використовувались при тестувані, оскільки виробнича подія або тригер, як кінцева точка HTTP, краще відображатиме існуючу продуктивність платформи. Об'єм пам'яті функцій дорівнює 512 МБ та використовується на всіх платформах, окрім Microsoft Azure, яка динамічно виявляє вимоги до пам'яті функцій.

Також було оцінено функціональні параметри продуктивності для наступних провайдерів бессерверних обчислень: Azure Functions, Apache OpenWhisk, AWS Lambda та Google Cloud Functions. Для цього був розроблений проект для проведення експериментів, який застосовує функцію тестування Node.js до різних служб за допомогою бессерверної платформи.

Затримки мережі не враховувались в результатах, але для зменшення їх ефекту експерименти проводилися на віртуальних машинах всередині того ж регіону, що і цільова функція.

3.2.1 Оцінювання показників масштабування бессерверних функцій

Був проведений тест для оцінювання показників масштабування функцій, який призначений для вимірювання здатності бессерверних платформ ефективно масштабувати функцію. Розроблений проект підтримує функції тестування, викликаючи кожен запит одразу після отримання відповіді від попереднього виклику. Тест починається з одного виклику і кожні 10 секунд додає додатковий паралельний виклик, щонайбільше до 15 одночасних запитів до функції. Інструмент вимірює кількість відповідей, отриманих за секунду, яка повинна зростати із рівнем кількості викликів. Цей тест повторювався 10 разів на кожному з провайдерів, таких як Azure Functions, Apache OpenWhisk, AWS Lambda, Google Cloud Functions, а також на розробленому прототипі платформи. На рисунку 3.1 зображені результати тестування показників масштабування бессерверних функцій.

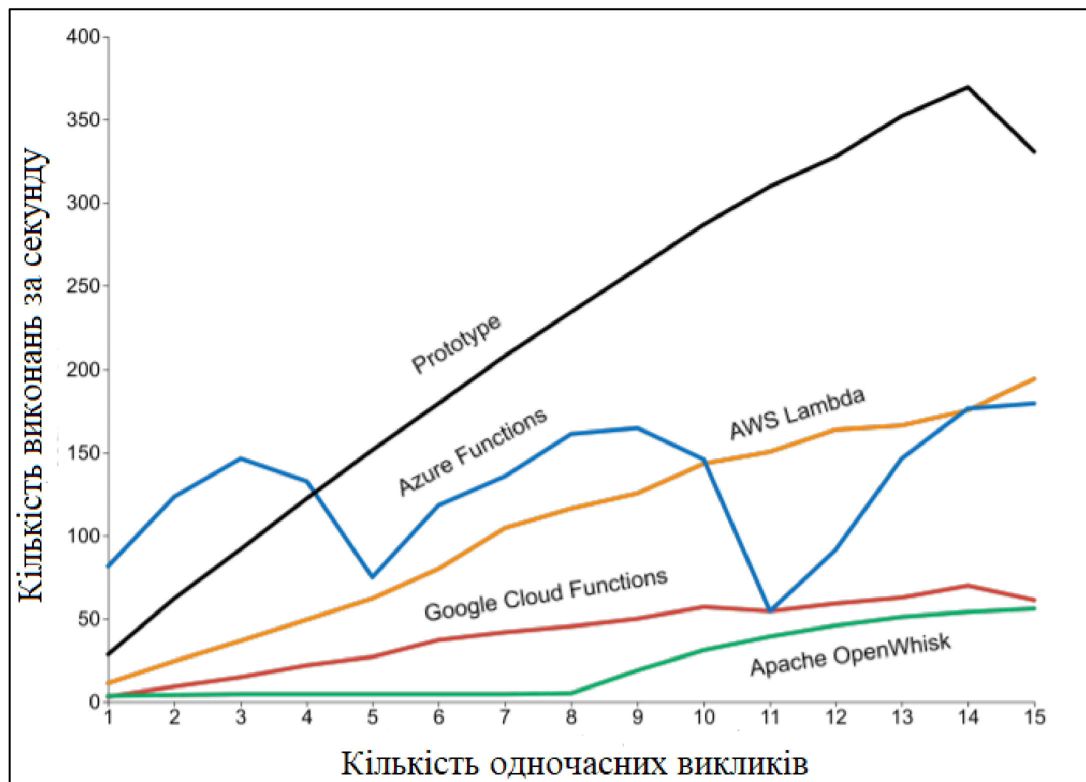

Рисунок 3.1 - Результати оцінювання показників масштабованості. Графік залежності середньої кількості виконань в секунду від кількості одночасних запитів на виконання функції

Прототип демонструє майже лінійне масштабування між рівнями паралельності 1 і 14, але продуктивність значно знижується при 15 паралельних запитах. Це падіння зумовлене збільшенням затримок з теплої черги, що вказує на наближення до максимального навантаження однієї черги сховища Azure. AWS Lambda масштабується сублінійно і демонструє найбільшу пропускну здатність з розглянутих комерційних платформ за 15 одночасних запитів. Google Cloud Functions демонструє сублінійне масштабування, яке зменшується, коли кількість одночасних запитів наближається до 15. Ефективність функцій Azure надзвичайно варіативна, хоча пропускна здатність місцями досить висока, перевершуючи інші платформи на нижчих рівнях паралельності. Ця мінливість потребує окремих досліджень, особливо тому, що вона зберігається протягом ітерацій тесту. Продуктивність OpenWhisk демонструє низьку пропускну здатність до восьми одночасних запитів, після чого функція починає сублінійно масштабуватися. Ця поведінка може бути спричинена пулом контейнерів OpenWhisk, що запускає

кілька контейнерів перед початком повторного використання, але ця поведінка залежить від конфігурації розгортання IBM.

3.2.2 Оцінювання показників затримки холодного старту бессерверних функцій

Був проведений тест відмови, який призначений для аналізу затримки холодного старту та поведінки закінчення терміну дії екземплярів функцій на різних платформах. Тест на відмову надсилає одиничні запити на виконання функції тестування із збільшеними інтервалами, від однієї до тридцяти хвилин. Результати оцінювання зображені на рисунку 3.2.

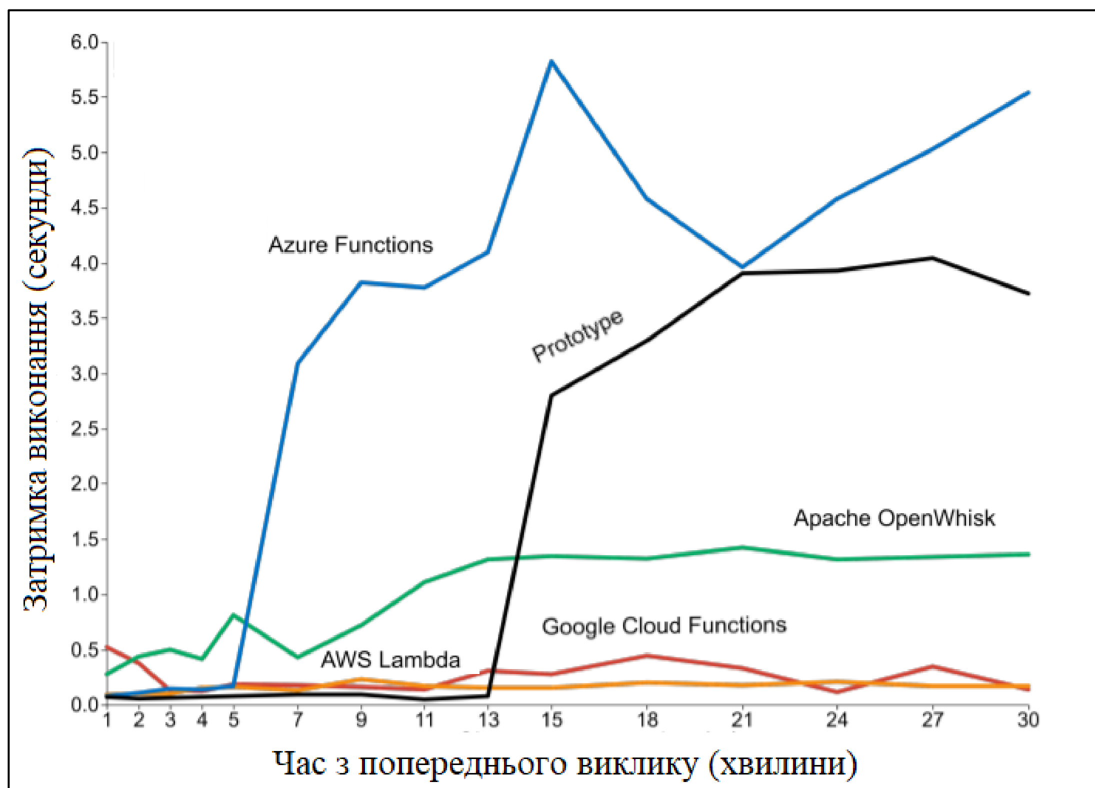

Рисунок 3.2 - Результати оцінювання затримки холодного старту функцій. Графік залежності затримки виконань від часу з моменту попереднього виклику функції

Функціональні контейнери видаляються через 15 хвилин невикористання. На рисунку 3.2 наявна затримка виконання через 15 хвилин, що є наслідком холодного запуску прототипу. Azure Functions також закінчує термін дії ресурсів функцій через кілька хвилин і відображає схожі показники холодного старту як у прототипі. Важливо зазначити, що хоча прототип і функції Azure є реалізацією Windows, середовища виконання їх функцій дуже різні, оскільки прототип використовує контейнери Windows, а функції Azure працюють на віртуальних машинах служби додатків Azure. Крім того, OpenWhisk також звільняє контейнери приблизно через 10 хвилин і має значно менші показники холодного старту, ніж Azure Functions або прототип. Найголовніше, AWS Lambda та Google Cloud Functions значною мірою не залежать від простою функцій. Можливими поясненнями такої поведінки може бути надзвичайно швидкий час запуску контейнера або попередній розподіл контейнерів.

3.2.3 Результати оцінювання показників швидкості обробки запитів бессерверних функцій

Враховуючи, що тепла черга є чергою FIFO, є проблематичним припинення терміну дії контейнера. Наприклад, функція перебуває під великим навантаженням і має 10 контейнерів, виділених для виконання. Потім завантаження падає до такої міри, що один контейнер зможе обробляти всі виконання функції. В ідеалі зайві 9 контейнерів видаляються через короткий час, але через характер черги FIFO, доки виконується 10 функцій за період закінчення терміну дії контейнера, усі 10 контейнерів залишаться призначеними для функції. Рішення цієї проблеми, звичайно, полягає у використанні "теплих стеків" замість "теплих черг", але Azure Storage наразі не підтримує чергу LIFO.

Одним із можливих рішень є використання кешу Redis для зберігання теплих стеків, де тип даних «список» підтримує операції з чергою LIFO нестандартно.

Однак Azure Storage та Redis - це дуже різні системи як за призначенням, так і за реалізацією [19]. Azure Storage забезпечує високу доступність та гарантію узгодженості, а також довговічність повідомлень та багаторегіональну реплікацію. І навпаки, при використані операцій Redis, особливо під час роботи на кластері Redis, сценарії відмов можуть призвести до втрати даних, невідповідності та тимчасової недоступності. Отже, розміщення та використання даних у Redis повинно здійснюватися обережно, враховуючи вимоги до стійкості та доступності цих даних.

Теплі контейнери прототипу самі по собі є кешем екземплярів функцій, що використовуються для уникнення холодного старту при кожному виконанні. Дійсно, прототип міг би функціонувати повністю без теплих контейнерів, змушуючи для кожного виконання функції створювати новий контейнер. Отже, зберігання теплих екземплярів у кеші Redis може бути життєздатним, оскільки рідкісна втрата даних екземпляра не впливає на коректність служби і призведе до закінчення терміну дії забутого екземпляра через короткий час. Послідовність потребує визначення, оскільки джерелом істинності статусу інстанції є робочі служби. Якщо трапляється помилка послідовності, то повідомлення двічі виводиться зі списку, робоча служба поверне HTTP статус «409 Conflict» до другого запиту на виконання. Потім веб-служба, яка отримала дублікат повідомлення, видалить його та перейде до наступного пункту у списку.

Для удосконалення показників швидкості обробки запитів було розроблену модифіковану версію прототипу платформи, в якій теплі черги в сховищі Azure замінюються списками LIFO в екземплярі Redis. Веб-служба знаходить доступний «теплий» екземпляр, видаляючи елемент з початку списку, а робоча служба вставляє повідомлення про доступність, як тільки контейнер звільняється, також на початок списку. Усі інші артефакти сховища, включаючи холодну чергу, залишаються в сховищі Azure. На рисунку 3.3 показані результати оцінювання швидкості обробки запитів за допомогою модифікованої версії прототипу платформи.

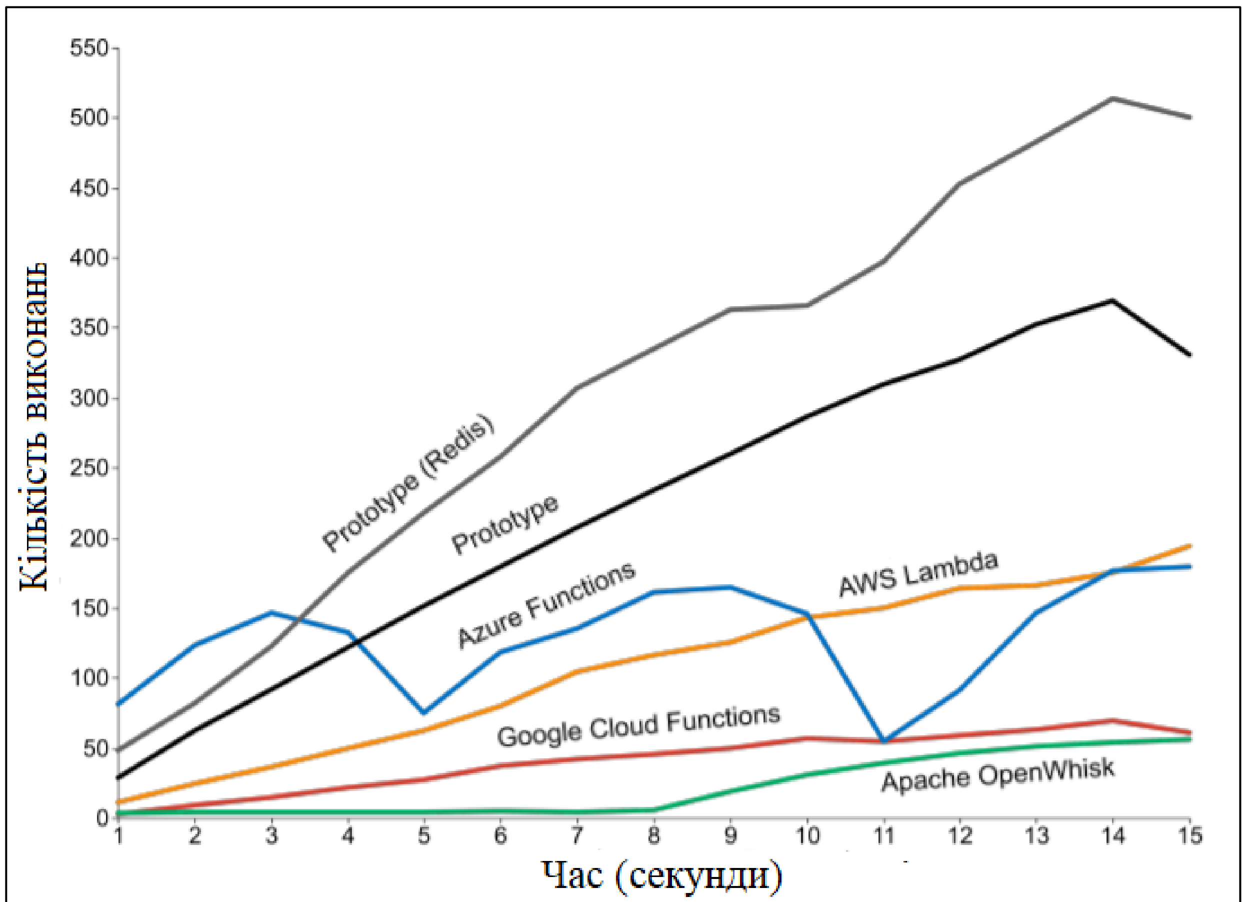

Рисунок 3.3 - Результати оцінювання швидкості обробки запитів за допомогою модифікованого прототипу платформи. Графік залежності кількості виконань викликів від кількості часу.

Модифікований прототип здатний досягти коротшої затримки порівняно з оригінальною версією, покращуючи загальну пропускну здатність прототипу. Однак необхідне подальше вивчення, щоб визначити, чи можливо використовувати рішення Redis за сценаріїями відмов. Якщо Redis виявиться нежиттєздатним рішенням через умови відмови, інші рішення, такі як запровадження послідовного хешування, є також перспективними. Але найголовніше, що рішення Redis поводиться як черга LIFO і дозволяє правильно обробляти екземпляри функцій. Тож цю модифікацію можна використовувати для обробки зображень, щоб прискорити швидкість обробки запитів.

3.2.4 Результати оцінювання показників масшатабованості бессерверної архітектури в порівнянні з монолітною архітектурою

Окрім порівняння різних провайдерів бессерверних обчислень, також було проведене оцінювання показників масштабованості бессерверної архітектури в порівняні з монолітною архітектурою подібно до оцінювання продуктивності, детально описаних у підрозділах 3.2.1, 3.2.2, 3.2.3.

У цьому дослідженні проводився стрес-тест бессерверної та монолітної архітектур, створюючи навантаження із імітованими користувачами, які одночасно роблять запити до програми.

Оскільки бессерверна архітектура обіцяє автоматичне масштабування та розподіл ресурсів, цікаво порівняти це з монолітною архітектурою. Для цього були сгенеровані робочі навантаження розділені на три фази: P0, P1, P2. P0 - це фаза розминки, коли постійна кількість віртуальних користувачів кожну секунду надсилає запити до системи протягом 60 секунд. Для P1 навантаження лінійно збільшується протягом однієї хвилини. Після фази масштабування починається P2, і навантаження залишається постійним протягом 180 секунд. Це дозволяє побачити, чи змінює платформа свою поведінку, якщо вона навантажена протягом тривалого періоду.

Тест проводився кілька разів і для більшої наочності під час експерименту вираховувалася медіана всіх спроб і встановлювався найгірший результат, це буде відображено на графіках.

В першу чергу було протестовано перше навантаження для монолітної архітектури. На рисунку 3.4 зображено середній час відгуку першого робочого навантаження, де кількість віртуальних користувачів масштабується від 0 до 60 нових користувачів на секунду.

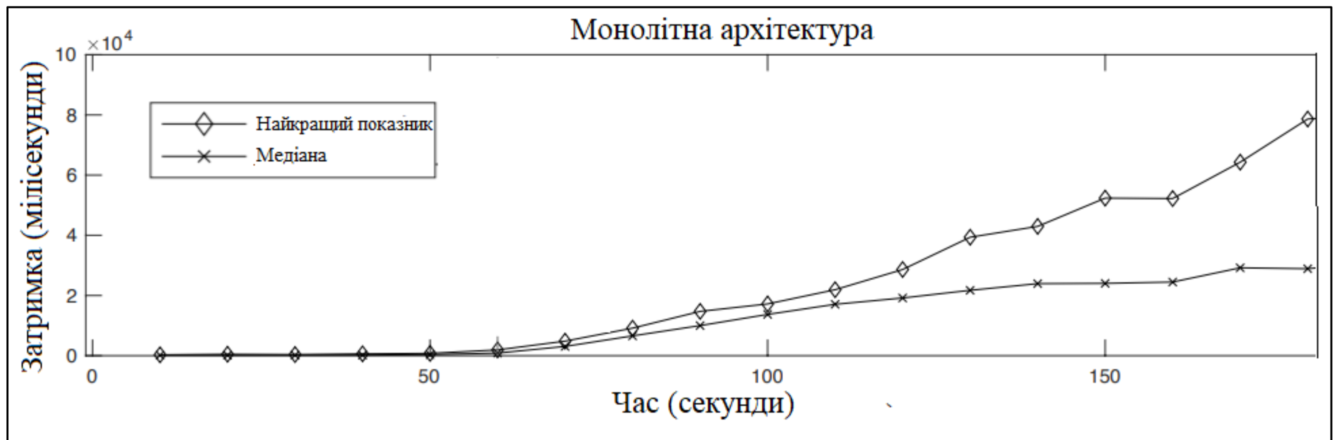

Рисунок 3.4 – Графік залежності затримки від часу виконання першого робочого навантаження монолітної архітектури

Таке саме оцінювання було проведено для прототипу платформи, результати оцінювання наведені на рисунку 3.5.

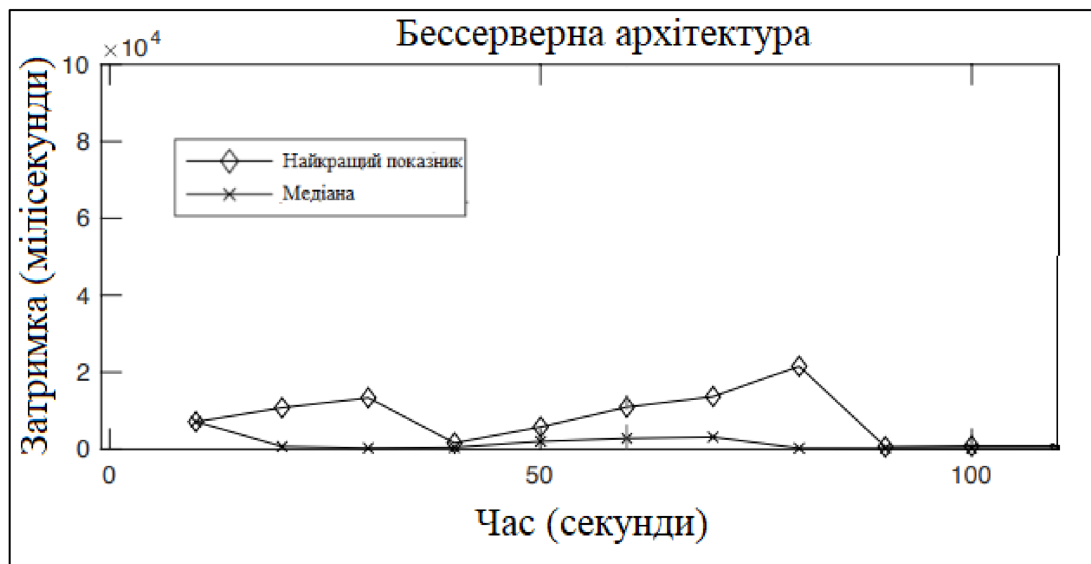

Рисунок 3.5 - Графік залежності затримки від часу виконання першого робочого навантаження бессерверної архітектури

Очевидно, що послідовність запитів генерує більш інтинсивне навантаження, ніж одиничний запит. При наближенні до 60 користувачів у секунду затримка для монолітної архітектури стабільно зростає і не може стабілізуватися. Підвищена затримка спостерігається також і для бессерверної архітектури, однак вона стабілізується досить швидко, а саме починаючи з 80 секунди.

Далі наведені результати оцінювання для другого робочого навантаження. На рисунку 3.6 зображені результати, які отримані для монолітної архітектури.

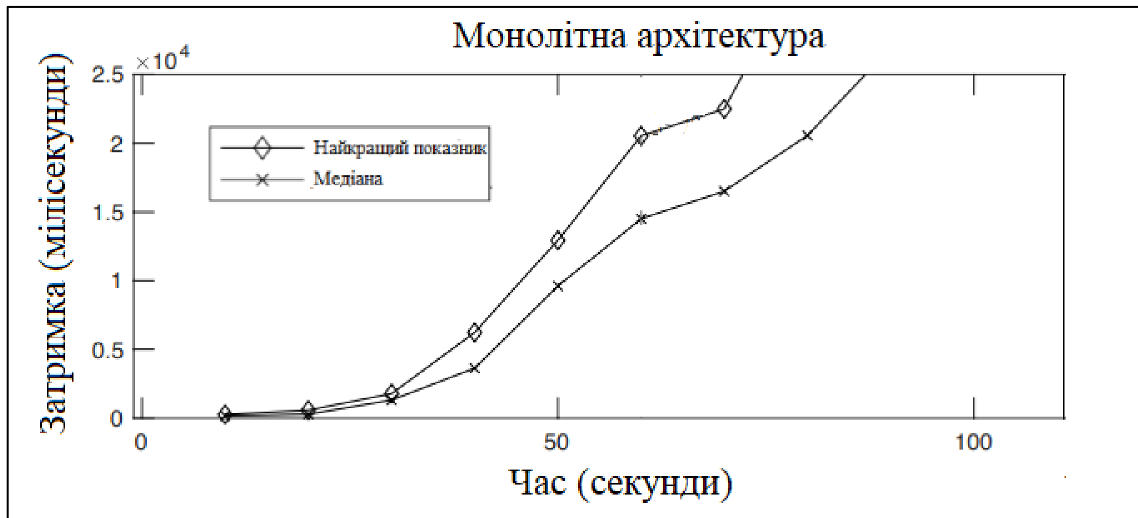

Рисунок 3.6 - Графік залежності затримки від часу виконання другого робочого навантаження монолітної архітектури

На рисунку 3.7 наведені результати оцінювання другого робочого навантаження для бессерверної архітектури.

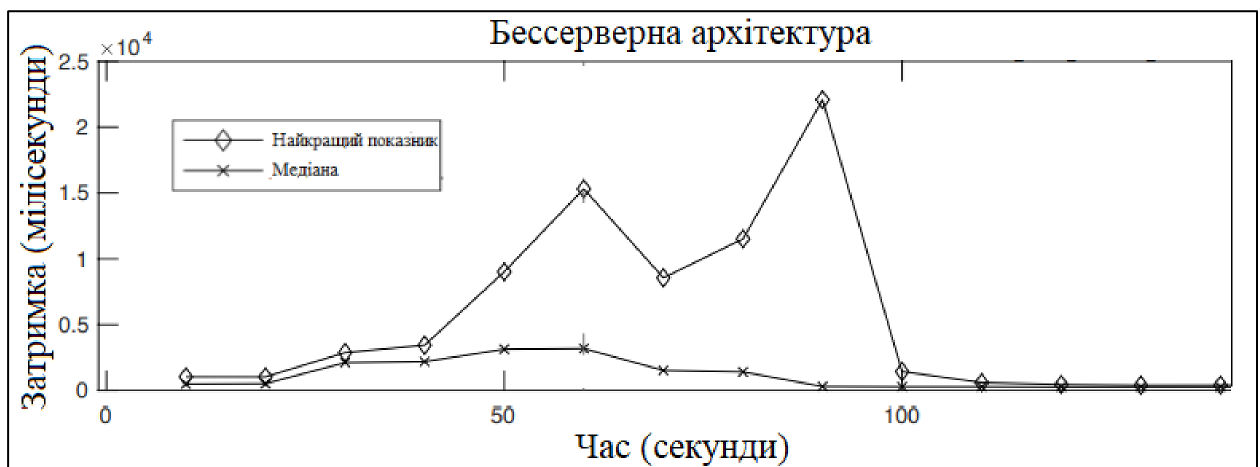

Рисунок 3.7 - Графік залежності затримки від часу виконання другого робочого навантаження бессерверної архітектури

Щоб краще продемонструвати різницю між бессерверною та монолітною архітектурою, максимальна планка для затримки встановлюється на рівні 25000

мс. Подібно до першого робочого навантаження, час відгуку для монолітної архітектури швидко збільшився, тоді як бессерверна архітектура змогла адаптуватися до навантаження та стабілізувати затримку.

З цього можна зробити висновки, що бессерверна архітектура краще підходить для обробки зображень при високих навантаженнях, тоді як монолітна архітектура погано адаптується до інтенсивних навантажень і вимагає втручання операційної команди.

3.3 Рекомендацій щодо вибору провайдера бессерверного обчислення та можливості удосконалення окремих функцій провайдера

Після оцінювання функціональних параметрів провайдерів бессерверного обчислення та виявлення переваг та недоліків кожного з провайдерів, можна розробити рекомендації щодо вибору того чи іншого провайдера, а також запропоновути пропозиції щодо удосконалення деяких параметрів бессерверних обчислень.

Перш за все необхідно згадати цінову політику провайдерів бессерверних обчислень. У таблиці 1.2 підрозділу 1.3 вказано ціну за використаний гігабайт трафіку для кожного провайдера. Виходячи з цієї таблиці, найвигіднішу пропозицію надає Google Cloud Functions, а також цей провайдер надає 2 мільйони бескоштовних запитів, інші провайдери надають лише один мільйон бескоштовних запитів, що теж забезпечує бескоштовне використання для невеликих додатків чи модулей для обробки зображень. Тож якщо додаток користувача потребує до одного мільйона запитів, варто обирати провайдера переважно за іншими критеріями.

Іншим критерієм вибору провайдера є повнота функціоналу, спільнота і підтримка існуючих сервісів. У таблиці 1.3 підрозділу 1.3 було наведено порівняння провайдерів бессерверних обчислень за повнотою функціоналу.

Найширший функціонал надає AWS Lambda, а також цей провайдер має найбільшу спільноту серед конкурентів, але Azure Functions теж надає широкий функціонал, особливо для продуктів Microsoft. Так як Google Cloud Functions є наймолодшим провайдером серед конкурентів, він має тільки основний функціонал, але не більш того. Тож за критерієм функціоналу та спільноти найкращим вибором є AWS та Azure, для технології .NET.

Також було розглянуто параметри ефективності провайдерів, такі як масштабованість, затримка холодного старту та швидкість обробки запитів. За критерієм масштабованості найстабільнішим провайдером є AWS Lamda, його показники є лінійними та найвищими. Azure Functions також показали ефективні результати, але вони досить мінливі. Google Cloud Functions хоч і мають досить стабільні результати, але пропускна здатність є низькою по відношенню до конкурентів. Слід зазначити, найкращий результат показав прототип платформи, це означає, що навіть Lamda сервіс має недоліки, але все одно за цим показником слід обирати AWS Lambda серед розглянутих комерційних провайдерів.

За параметром затримки холодного старту бессерверних функцій (див. рис. 3.2), Google Cloud Functions та AWS Lamda мають найкращі показники, тому що затримка для цих провайдерів мінімальна. Azure Functions має досить довгу затримку, що іноді дорівнює шести секундам, тож за цим показником найкращим вибором є AWS Lamda та Google Cloud Functions.

Останім показником є швидкість обробки запитів, цей параметр має найнижчий приорітет, тому що усі провайдери мають досить рівні показники, але все одно можна виділити AWS Lamda, так як цей провайдер обробляє запити найшвидше навіть при великих навантаженнях.

На основі результатів дослідження запропоновано шлях до збільшення швидкості обробки запитів через заміну теплих черг в сховищі Azure на списки LIFO в екземплярі Redis. Це дозволить веб-службі знаходити доступний «теплий» екземпляр, видаляючи елемент з початку списку, а робочій службі вставляти повідомлення про доступність, як тільки контейнер звільняється, також на початок списку, що прискорить обробку запитів.

Також для обробки зображень було порівняно між собою бессерверну та монолітну архітектури. Порівняння показало, що бессерверна архітектура краще адаптується до високих навантажень і не потребує втручання розробників або операційної команди, так як провайдер бессерверних обчислень автоматично масштабує кількість виділених контейнерів функцій.

Підводячи підсумки, бессерверна архітектура має перспективи для удосконалення, але навіть на даний момент для обробки зображень її використання є обґрунтованим. Вибір провайдера залежить від ресурсів, стека і потреб розробників, але на теперішній час AWS Lamda не поступається в функціоналі конкурентам, має найбільшу спільноту і показує кращі результати за більшістю показників.

4 ОПИС ПРОГРАМНОЇ СИСТЕМИ ДЛЯ ОЦІНЮВАННЯ ЕФЕКТИВНОСТІ МЕТОДІВ ТА ЗАСОБІВ РОЗРОБКИ БЕССЕРВЕРНИХ ДОДАТКІВ ДЛЯ ОБРОБКИ ЗОБРАЖЕНЬ

4.1 Компоненти прототипу бессерверної обчислювальної платформи

Для оцінювання результатів функціональних параметрів провайдерів бессерверного обчислення було розроблено прототип платформи, яка складається з двох компонентів: веб-служби, яка надає загальнодоступну API REST платформу, та робочої служби, яка управляє та виконує контейнери функцій. Веб-служба виявляє доступні робочі служби через рівень обміну повідомленнями, що складається з різних черг Azure Storage. Метадані функції зберігаються в таблицях Azure Storage, а код функції зберігається у двійкових великих об’єктах Azure Storage. На рисунку 4.1 зображені компоненти реалізації прототипу платформи.

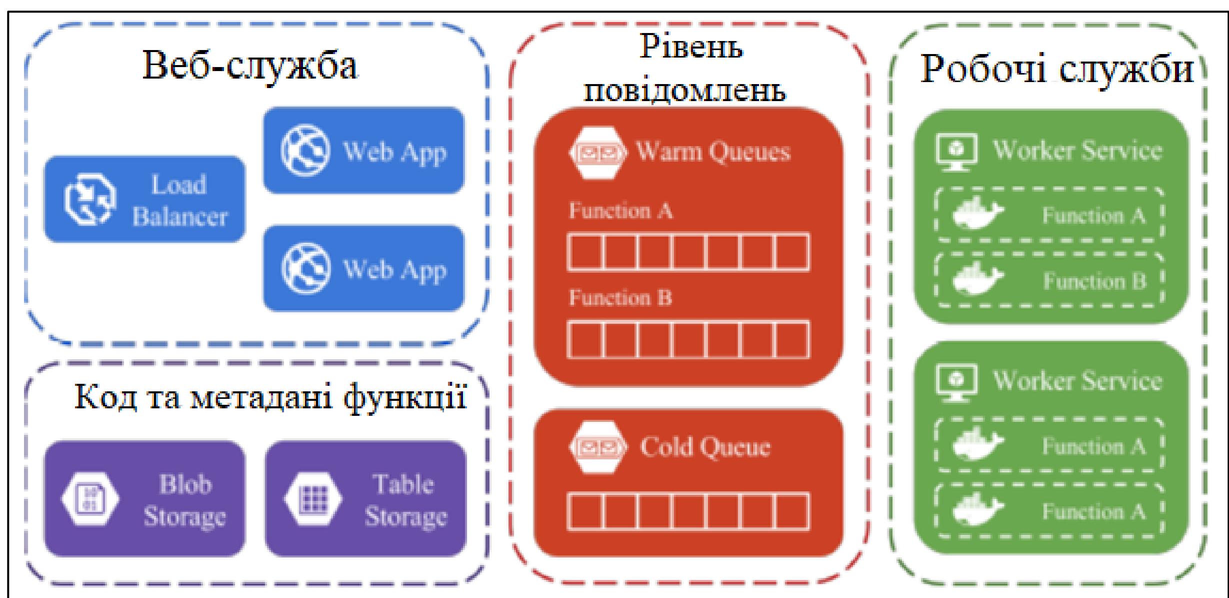

Рисунок 4.1 – Компоненти реалізації прототипу прототипу платформи

Azure Storage було обрано, оскільки це сховище даних забезпечує високомасштабовані підходи до зберігання даних із низькою затримкою за допомогою простого API, що добре підходить до цілей прототипу платформи.

4.2 Функціональні особливості прототипу бессерверної обчислювальної платформи та обгрунтування обраних технологій

Для реалізації прототипу бессерверної платформи була обрана технологія .NET з наступних причин: обширний об’єм технічної документації по роботі з бессерверними обчисленями, github репозиторій з відкритим кодом реалізації Azure Functions та інтегроване середовище розробки, яке має підтримку розробки бессерверних функцій. Також для розробки прототипа була обрана мова програмування C#, оскільки це передова мова програмування технології .NET. В якості хмарної платформи був обраний сервіс Microsoft Azure і контейнери Windows.

Вимірювання продуктивності розробленого прототипу та інших провайдерів безсерверної платформи проводилось за допомогою розробленого Node JS додатку, що дозволяє вимірювати характеристики продуктивності бессерверних платформ, таких як AWS Lambda, Azure Functions, Google Cloud Functions та IBM OpenWhisk.

Для використання додатку для вимірювання продуктивності необхідно налаштувати конфігураційні файли та ввести особливі облікові дані. На прикладі Google Cloud Functions, потрібно створити акаунт на платформі, після чого необхідно ввімкнути необхідні API. Для цього потрібно перейти на інформаційну панель API і натиснути посилання «Enable APIs and Services». Це посилання зображене на рисунку 4.2.

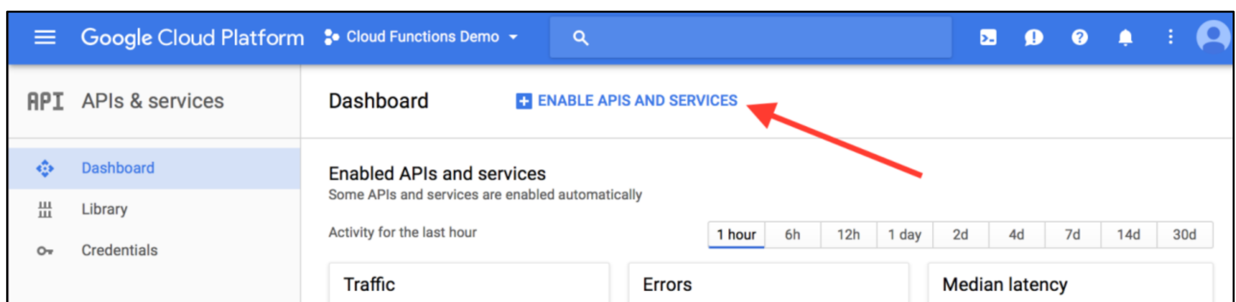

Рисунок 4.2 – Посилання «Enable APIs and Services»

Після цього треба знайти пункт «Google Cloud Functions» та натиснути кнопку «Увімкнути» та повторити те ж саме для пункту «Google Cloud Deployment Manager». Останім кроком є створення власних облікових даних, для цього потрібно натиснути на кнопку на «Create credentials» та обрати опцію «Service Account Key». На рисунку 4.3 зображені основні елементи сторінки «Credentials».

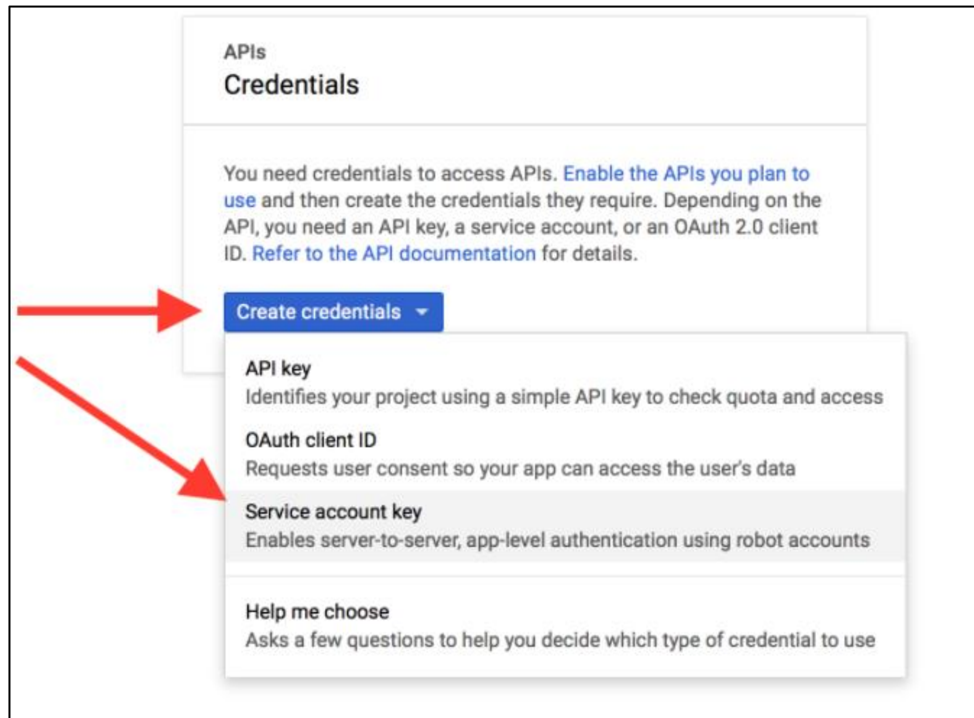

Рисунок 4.3 – Меню «Credentials»

Коли облікові дані будуть створені, їх необхідно перемістіти у файл «~/.gcloud / keyfile.json». Виконавши ці кроки буде створена бессерверна функція, яка готова до виконання коду.

4.3 Тестування прототипу бессерверної обчислювальної платформи

Найважливішим функціональним параметром прототипу бессерверної платформи є продуктивність, тому було обрано наступні види тестування:

- тестування навантаження - визначення максимально можливого навантаження, при якому система працює правильно;
- стрес тестування - призначене для перевірки працездатності системи при нестандартних навантаженнях.

Для обох видів тестування прототипу бессерверної обчислювальної платформи використовувалися інструменти порівняльного аналізу, такі як:

- «Artillery.io» - це інструмент, який використовується та пропонується при тестуванні платформ та мікросервісів FaaS. Цей інструментарій дозволяє тестувати навантаження та проводити функціональне тестування з відкритим кодом. Він може імітувати користувачів веб-програми, надсилаючи велику кількість мережевих запитів на вказаний веб-сайт або програму. Інструмент CLI (інтерфейс командного рядка) дозволяє визначати складні сценарії тестування, де користувачі можуть вказувати HTTP-запити та корисні навантаження даних, які будуть доставлені до програми. Це робить його ідеальним інструментом для перевірки продуктивності та поведінки програм, які взаємодіють через REST API. Інструмент також легко піддається програмуванню та пропонує легку установку за допомогою популярного менеджера пакунків npm;
- «JMeter» - інструмент з відкритим кодом, що використовується для порівняльного тестування. JMeter також використовується у тестуванні веб-додатків і є додатком Java для тестування продуктивності статичних та динамічних веб-додатків. Як і Artillery, він може імітувати великий користувальницький трафік до програми і підтримує широкий спектр мережевих протоколів, наприклад, HTTP, HTTPS, REST тощо.

Для навантажувального і стрес тестувань використовувалися різні робочі навантаження, щоб визначити допустиме і надмірне навантаження. Для цього протягом 10 секунд відбувався виклик бессерверної функції прототипу фіксовану кількість разів в секунду. Результати тестування відображені в таблиці 4.1.

Таблиця 4.1 – Результати навантажувального і стрес тестувань

Кількість викликів в секунду	Затримка за версією Artillery.io	Затримка за версією JMeter
200	17344 мілісекунд	16291 мілісекунд
500	34785 мілісекунд	31765 мілісекунд
800	Помилка «OutOfMemory»	Помилка «OutOfMemory»

Результати тесту показали, що прототип не витримує навантаження в 800 запитів в секунду, так як прототип платформи не встигає масштабувати кількість екземплярів функцій і у створених бессерверних функцій не вистачає пам'яті щоб обробити запит. Однак розроблений продукт може обробити 500 запитів в секунду, затримка виконання в середньому дорівнює 32 секундам.

ВИСНОВКИ

В роботі був проведений аналіз предметної області, а саме клієнт-серверної та бессерверної архітектур програмного забезпечення та компонентів цих архітектур. Також були виявлені переваги та недоліки бессерверної архітектури програмного забезпечення в порівняні з класичною архітектурою. Порівняно політику ціноутворення та функціональні параметри послуг основних провайдерів бессерверних обчислень. Також були розглянуті особливості бессерверної обробки зображень.

Запропонована методика оцінювання ефективності методів та засобів розробки бессерверних додатків для обробки зображень, визначені критерії оцінювання ефективності та розроблена процедура моделювання функціональних параметрів провайдерів бессерверних обчислень.

Для моделювання функціональних параметрів провайдерів був побудований прототип платформи бессерверних обчислень, призначений для оцінювання показників масштабованості, затримки холодного старту, швидкості обробки запитів та порівняння з відповідними показниками монолітної архітектури.

Отримані результати оцінювання ефективності функціональних параметрів методів та засобів розробки бессерверних додатків для обробки зображень, а саме результати масштабованості бессерверних функцій, затримки холодного старту, швидкості обробки запитів в порівняні з клієнт-серверною архітектурою. На основі отриманих результатів оцінювання були розроблені рекомендацій щодо вибору провайдера бессерверного обчислення та запропоновано шлях до збільшення швидкості обробки запитів через зміну теплих черг у сховищі Azure на списки LIFO в екземплярі Redis.

Отримані результати оцінювання ефективності методів та засобів розробки бессерверних додатків для обробки зображень та розроблені рекомендації щодо їх використання дозволять здійснити обгрунтований вибір найбільш ефективних

методів та засобів бессерверних обчислень в залежності від висунутих вимог для певної бізнес задачі, а також збільшити швидкість обробки запитів.

За результатами дослідження, проведеного в атестаційній роботі, були опубліковані тези доповіді на тему “Дослідження ефективності методів та засобів розробки бессерверних додатків для обробки зображень” в матеріалах XXIV міжнародного молодіжного форума «Радіоелектроніка та молодь у XXI столітті» та опублікована в журналі «Наука онлайн» стаття на тему «Результати дослідження ефективності методів та засобів розробки бессерверних додатків для обробки зображень».

ПЕРЕЛІК ДЖЕРЕЛ ПОСИЛАНЬ

1. Andrey Arsenov, Igor Ruban, Kyrylo Smelyakov, Anastasiya Chupryna Evolution of Convolutional Neural Network Architecture in Image Classification Problems Selected Papers of the XVIII International Scientific and Practical Conference on IT and Security // CEUR Workshop Processing. 2018.
2. International Organization for Standardization, International Electrotechnical Commission, Institute of Electrical and Electronics Engineers, and IEEE-SA Standards Board, ISO/IEC/IEEE 42010:2011: Systems and software engineering - architecture description. // Geneva; New York; Institute of Electrical and Electronics Engineers – 2011 – ISBN: 978-0-7381-7142-5.
3. Jonas E., Cloud Programming Simplified: A Berkeley View on Serverless Computing [Електронний ресурс] / Eric Jonas, Johann Schleier-Smith, Vikram Sreekanti, Chia-Che Tsai, Anurag Khandelwal, Qifan Pu, Vaishaal Shankar, Joao Menezes Carreira, Karl Krauth, Neeraja Yadwadkar, Joseph Gonzalez, Raluca Ada Popa, Ion Stoica and David A. Patterson // EECS Department University of California, Berkeley Technical Report No. UCB/EECS-2019-3 – 2019 – Режим доступу: https://www2.eecs.berkeley.edu/Pubs/TechRpts/2019/EECS-2019-3.html.
4. Papazoglou M. P. Web Services: Principles and Technology / M. P. Papazoglou // Tilburg: Pearson Education – 2008Sami Bhir. Overview of Transactional Patterns: Combining Workflow Flexibility and Transactional Reliability for Composite Web Services // Business Process Management. 2005.
5. Kyriy, V., Sheiko, I., Petrova, R., Optimization of management information support as a basis for organizational transformations at an enterprise periodical of Engineering and Natural Sciences. 2019. V.7. N. 2. P. 679-689.
6. Hellerstein, J.M., et al., Serverless computing: One step forward, two steps back. arXiv preprint arXiv:1812.03651, 2018.
7. Baldini, I., et al., Serverless computing: Current trends and open problems, in Research Advances in Cloud Computing. 2017. Springer. P. 1-20.

8. Kuzochkina A., Z. Dudar, Shirokopetleva M., Analyzing and Comparison of NoSQL DBMS // International Scientific and Practical Conference «Problems of Infocommunications. Science and Technology». 2018. P. 560-565.

9. Google Cloud Functions. [Електронний ресурс] // Режим доступу: https://cloud.google.com/functions/.

10. Березко Л. О. Ефективний метод обробки запитів до веб-сервісів / Л. О. Березко, А. І. Якимець // Вісник Національного університету "Львівська політехніка". – 2012. – № 745: Комп’ютерні системи та мережі. – С. 11–13.

11. Serhii Chalyi, Volodymyr Leshchynskyi. METHOD OF CONSTRUCTING EXPLANATIONS FOR RECOMMENDER SYSTEMS BASED ON THE TEMPORAL DYNAMICS OF USER PREFERENCES. «EUREKA: Physics and Engineering».- 2020.- Number 3.-p.43-50.

12. Roberts, M. Serverless Architectures. 2018 2020-01-30]; Режим доступу https://martinfowler.com/articles/serverless.html.

13. Albuquerque Jr, L.F., et al. Function-as-a-Service X Platform-as-aService: Towards a Comparative Study on FaaS and PaaS. in ICSEA. 2017.

14. Hendrickson, S., et al. Serverless computation with openlambda. in 8th Workshop on Hot Topics in Cloud Computing (HotCloud 16). 2016.

15. Kreger H., Web Services Conceptual Architecture (WSCA 1.0) / H. Kreger // IBM Software Group, Somers – 2001.

16. Clements P. Documenting software architectures: Views and beyond, 2nd ed., ser. SEI series in software engineering / P. Clements, F. Bachmann, L. Bass, D. Garlan, J. Ivers, R. Little, P. Merson, R. Nord, and J. Stafford, Eds. // Upper Saddle River, NJ: Addison-Wesley – 2011 – ISBN: 978-0-321-55268-6

17. ГОСТ 28806-90. Качество программных средств.

18. Ногін В.Д. Лінійна згортка критеріїв у багатокритеріальній оптимізації // Штучний інтелект та прийняття рішень. 2014.

19. Redis. Redis Cluster Specification. Режим доступу: https://redis.io/topics/ cluster-spec, 2017.