Міністерство освіти і науки України
Харківський національний університет радіоелектроніки

Факультет Комп'ютерної інженерії та управління
(повна назва)

Кафедра Електронних обчислювальних машин
(повна назва)

КВАЛІФІКАЦІЙНА РОБОТА

Пояснювальна записка

Рівень вищої освіти другий (магістерський)

Модель веб-застосунків на базі мікросервісної архітектури

(тема)

Виконав:
студент II курсу, групи СПм-20-2
Осіпова Д. Ю.
(прізвище, ініціали)

Спеціальність
123 – Комп'ютерна інженерія
(код і повна назва спеціальності)

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

Освітня програма
Системне програмування
(повна назва освітньої програми)

Керівник: ст. в. Фомічов О. О.
(посада, прізвище, ініціали)

Допускається до захисту

В. о. зав. кафедри ЕОМ _____ Волк М. О.
(підпис) (прізвище, ініціали)

2022 р.

Харківський національний університет радіоелектроніки

Факультет Комп'ютерної інженерії та управління

Кафедра Електронних обчислювальних машин

Рівень вищої освіти другий (магістерський)

Спеціальність 123 – Комп'ютерної інженерії
(код і повна назва)

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

Освітня програма Системне програмування
(повна назва)

ЗАТВЕРДЖУЮ:
Зав. кафедри _____
(підпис)
"____" _____ 20___ р.

ЗАВДАННЯ

НА КВАЛІФІКАЦІЙНУ РОБОТУ

студентові Осіповій Дар'ї Юріївні
(прізвище, ім'я, по батькові)

1. Тема роботи Модель веб-застосунків на базі мікросервісної архітектури

затверджена наказом по університету від " 24 " березня 2022 р. № 413 Ст

2. Термін подання студентом роботи до екзаменаційної комісії 18 травня 2022р.

3. Вхідні дані до роботи 1) існуючі технології, архітектурні рішення, методи та інструменти взаємодії клієнта і сервера; 2) системи обміну повідомленнями Apache Kafka, RabbitMQ; 3) програмне забезпечення для автоматизації розгортання та керування програмами Docker; 4) відкриті системи автоматичного розгортання Kubernetes, Docker Swarm і Apache Mesos.

4. Перелік питань, що потрібно опрацювати в роботі
1) аналіз різних типів архітектури веб-додатків;
2) порівняльний аналіз монолітної та мікросервісної архітектури за основними критеріями;
3) математичну модель розподіленої мікросервісної архітектури веб-додатка;
4) порівняльний аналіз черги повідомлень та REST API;
5) аналіз систем оркестрації контейнерів Kubernetes, Docker Swarm, Apache Mesos;
6) дослідження пропускної спроможності та порівняльний аналіз брокерів повідомлень Apache Kafka і RabbitMQ.

5. Перелік графічного матеріалу із зазначенням креслеників, схем, плакатів, комп'ютерних ілюстрацій (слайдів) 12 слайдів

6. Консультанти розділів роботи (заповнюється за наявності консультантів згідно з наказом, зазначеним у п.1)

Найменування розділу	Консультант (посада, прізвище, ім'я, по батькові)	Позначка консультанта про виконання розділу	
		підпис	дата

КАЛЕНДАРНИЙ ПЛАН

№	Назва етапів роботи	Термін виконання етапів роботи	Примітка
1	Огляд видів архітектури web-додатків	28.03.22-03.04.22	
2	Вибір та обґрунтування методики дослідження	04.04.22-13.04.22	
3	Вибір аналітичних засобів	14.04.22-20.04.22	
4	Формування результатів аналізу	21.04.22-26.04.22	
5	Проведення експерименту	27.04.22-05.05.22	
6	Оформлення кваліфікаційної роботи	06.05.22-10.05.22	
7	Подання кваліфікаційної роботи керівникові	11.05.22-12.05.22	
	та її попередній захист		
8	Подання кваліфікаційної роботи на рецензу-	13.05.22-14.05.22	
	вання		

Дата видачі завдання 28 березня 2022 р.

Студент _____
(підпис)

Керівник роботи _____ ст. в. Фомічов О. О.
(підпис) (посада, прізвище, ініціали)

РЕФЕРАТ

Пояснювальна записка атестаційної роботи: 86 с., 7 рис., 4 табл., 1 дод., 22 джерела.

CLIENT, SERVER, HTTP, MONOLIT, MICROSERVICE, REST API, MESSAGE BROKER, CONTAINERIZATION, ORCHESTRATION

Метою атестаційної роботи є аналіз існуючих видів архітектури веб – додатків з ухиленням мікросервісної архітектури і безпосередньо інструментів для роботи мікросервісів.

У процесі роботи будуть описані основи функціонування веб-додатків, розглянеться HTTP-протокол та RESTful API.

Будуть розглянуті і порівняні архітектури монолітної, сервіс-орієнтованої та мікросервісної систем. Розглянуться фактори впливу на масштабованість веб-додатків, принципи проектування

У ході виконання атестаційної роботи велике увагу приділялося основним мікросервісним інструментам взаємодії та розвертання. Буде надано загальний аналіз лідуючих сервісних шин, аналіз популярних інструментів контейнеризації та оркестрації контейнерів.

Як результат атестаційної роботи буде сформовано порівняльні таблиці з аналітичними висновками, буде проведено експеримент вимірювання пропускної здатності шин повідомлень.

ABSTRACT

Master’s thesis: 86 pages, 7 figures, 4 tables, 1 appendice, 22 sources.

ACCESS CONTROL AND MANAGEMENT SYSTEM, SOFTWARE, DATABASE, ARCHITECTURE, IP, REST

The major goal of this thesis is the analysis of existing types of architecture of web applications with evasion of microservice architecture and directly tools for work of microservices.

In the course of work, the basics of web applications will be described, the HTTP protocol and the RESTful API will be considered.

The architectures of monolithic, service-oriented and microservice systems will be considered and compared. Factors influencing the scalability of web applications, design principles will be considered.

During the certification work, much attention was paid to the main micro-service tools of interaction and deployment. A general analysis of leading service tires, analysis of popular containerization and orchestration tools will be provided.

As a result of the attestation work, comparative tables with analytical conclusions will be formed, and an experiment of measuring the bandwidth of message buses will be conducted.

ЗМІСТ

ПЕРЕЛІК УМОВНИХ ПОЗНАЧЕНЬ, СИМВОЛІВ, ОДИНИЦЬ, СКОРОЧЕНЬ І ТЕРМІНІВ 8
ВСТУП 9
1 ЗАГАЛЬНІ ХАРАКТЕРИСТИКИ ТА ТЕОРЕТИЧНІ ОСНОВИ WEB-ДОДАТКІВ 10
1.1 Основи функціонування 11
1.1.1 Концептуальне представлення web-додатку 12
1.1.2 Мережева взаємодія 13
1.1.3 Механізм роботи веб-сервера 16
1.1.5 Типи web-додатків 22
1.2 Критерії оцінки 25
1.3 Постановка задачі 27
2 АРХІТЕКТУРА МОНОЛІТНОГО ТА МІКРОСЕРВІСНОГО ПІДХОДІВ 28
2.1 Визначення поняття масштабованої веб-архітектури та її основні характеристики 28
2.2 Основні типи архітектури сучасного програмного забезпечення 32
2.3 Розрахунок математичної моделі порогового значення масштабування розподіленої мікросервісної системи 37
3 АНАЛІЗ ТЕХНОЛОГІЙ ТА ІНСТРУМЕНТІВ ДЛЯ СТВОРЕННЯ МІКРОСЕРВІСНОЇ АРХІТЕКТУРИ 41
3.1 Мікросервісні комунікації REST API та черги повідомлень 42
3.2 Огляд фреймворків передачі даних за допомогою черги повідомлень 49
3.3 Інструменти розгортання мікросервісів 57
3.3.1 Визначення контейнеризації та переваги її використання 58

3.3.2 Визначення оркестрації контейнерів, переваги та принцип роботи 66
3.4 Дослідження пропускної спроможності шин повідомлень 72
ВИСНОВКИ 76
ПЕРЕЛІК ДЖЕРЕЛ ПОСИЛАННЯ 77
ДОДАТОК А 80
ГРАФІЧНИЙ МАТЕРІАЛ КВАЛІФІКАЦІЙНОЇ РОБОТИ 80

ПЕРЕЛІК УМОВНИХ ПОЗНАЧЕНЬ, СИМВОЛІВ, ОДИНИЦЬ, СКОРОЧЕНЬ І ТЕРМІНІВ

СУБД – система управління базами даних

AMQP – advanced message queuing protocol

API – application programming interface

CI/CD – continuous integration and continuous delivery

DDOS – distributed denial-of-service

DHCP – dynamic host configuration protocol

DNS – domain name system

FTP – file transfer protocol

IP – internet protocol

MQ – message queue

NAT – network address translation

OAuth – open authorization

REST – representational state transfer

SMTP – simple mail transfer protocol

TCP – transmission control protocol

ВСТУП

В даний час найпопулярнішим видом програм є web-додатки. Їх не потрібно встановлювати та оновлювати на кожному комп'ютері, та головною умовою користування є доступ до Інтернету. Першим веб-додатком вважають перший сайт, який з'явився в 1991 році, коли була представлена технологія World Wide Web, яка використовує HTTP протокол. Тоді це була сторінка з текстом та посиланнями. Через більше 30 років веб-додатки дуже сильно розвивалися і з кожним роком вимоги до програмного забезпечення зростають і вимагають ще більшого розвитку в проектуванні та розробці.

На початку переважним способом розробки веб-додатків була монолітна архітектура програми, яка не мала ні рівня бізнес-логіки, ні рівня обслуговування даних [1]. Згодом монолітна «клієнт-серверна» архітектура стала поділятися на рівні:

- презентації;
- бізнес-логіки;
- доступу до бази даних.

Головний недолік монолітної архітектури – що логіка всіх трьох вище перелічених пунктів генеруються в одній кодовій базі. Виникають труднощі зміни або оновлення компонентів, розробники програмного забезпечення повинні вносити зміни до всієї програми. В результаті, програма застаріває від складності.

Багато великих компаній, таких як Airbnb, Netflix, Spotify, Intagram перенесли свої програми та системи на мікросервіси. Мікросервісна архітектура дозволяє масштабувати потужності системи та стабілізувати роботу залежно від напливу користувачів.

Мета дипломної роботи полягає в аналізі можливості мікросервісної архітектури та показати її сильні та слабкі сторони з точки зору розробки програмного забезпечення.

1 ЗАГАЛЬНІ ХАРАКТЕРИСТИКИ ТА ТЕОРЕТИЧНІ ОСНОВИ WEB-ДОДАТКІВ

Історично склалося три типи додатків – настільні, мобільні та web-додатки. Настільні програми запускаються на комп'ютері клієнта та виконують свій код саме там. Вони часто мають більш багатий і чуйний інтерфейс користувача і дозволяють реалізовувати більш складні сценарії. Залежно від типу операційної системи, процесора, відео карти та інших параметрів можуть знадобитися різні версії програми. Це створює певні незручності розробникам, потрібно постійно враховувати різні чинники, розширювати обсяг коду обліку всіх можливих комбінацій апаратних даних. Також користувачам необхідно завантажити постійні оновлення, і відповідати вимогам додатка про апаратну частину комп'ютера, версію операційної системи, обсягом пам'яті та інше.

Мобільні програми створені виключно для смартфонів, планшетів та інших пристроїв із встановленою там операційною системою Android, iOS та ін. Це також завдає незручностей розробникам з урахуванням усіх особливостей мобільних додатків. Важливо відзначити, що багато мобільних програм часто є web-додатками, адже багато браузери дозволяють перетворити web-додаток в мобільний.

Незважаючи на те, що існує низка технологій, що спрощують створення динамічних веб-додатків, їх розробка, як і раніше, залишається досить трудомістким завданням. Розробка веб-програм істотно відрізняється від розробки настільних систем. Тому є дві головні причини:

- веб-програми виконуються на сервері. Весь програмний код виконується в рамках веб-сервера, а клієнту доставляється готова розмітка HTML, яка відображається всередині браузера;
- веб-програми не зберігають стану. Сервер не зберігає дані про користувача після того, як обробив запит.

Обидва ці фактори суттєво впливають на процес розробки веб-додатків. Через це при побудові будь-якого веб-програми доводиться вирішувати типові завдання – способи зберігання інформації про користувача, організація сеансів роботи користувача, способи переходів від сторінки до сторінки, історія навігації, механізми оптимізації ефективності, такі як кешування.

Набір цих завдань є досить стандартним і однаково вирішується для більшості веб-додатків, його реалізація винесена в окремі технології, які називаються технологіями для розробки веб-додатків. До таких технологій відносяться технологія Microsoft ASP.NET, ASP.NET Core, JavaEE, PHP, Ruby on Rails та інші ці технології мають готову реалізацію рішень і містять всі компоненти, необхідні для реалізації веб-додатків і враховують їх специфіку.

1.1 Основи функціонування

Web-додаток – це комп'ютерна програма, що працює на сервері в глобальній мережі Інтернет і як інтерфейс взаємодії з клієнтом використовує веб-браузер, програму здатну розбирати мову розмітки HTML, таблиці стилів CSS, додатково розпізнавати скрипти мови JavaScript. браузер також називають просто браузером або тонким клієнтом і або не містить, або містить частину бізнес-логіки. Крім тонкого клієнта, програма також містить веб-сервер, спеціальну програму, відповідальну за обробку запитів з мережі та повернення даних браузеру. Web-додаток часто називають сайтом, але на відміну від нього відображає користувачеві динамічний вміст, що генерується виконуваним кодом програми.

1.1.1 Концептуальне представлення web-додатку

Позначимо необхідні компоненти веб-програми – браузер, веб-сервер, протокол взаємодії клієнта і сервера HTTP, мова розмітки для відображення вмісту HTML. Щоб веб-програма стала доступною для клієнтів, її необхідно розмістити в рамках веб-сервера. Далі програма отримає свою унікальну адресу в рамках протоколу HTTP, наприклад, «http://application.ua». Використовуючи цю адресу, користувач може звернутися до програми. Для цього користувач повинен запустити браузер і ввести адресу програми в рядку запиту. Після цього браузер згенерує запит на сервер і відправить його, використовуючи протокол взаємодії HTTP [2]. У момент, коли сервер приймає цей запит, він розпізнає, що саме вимагає користувач на основі інформації, що містить запит. У разі простого статичного застосування, відповідь містить гіпертекстову розмітку HTML, що містить структуру документа, який передається користувачеві. Після того, як браузер отримає відповідь у вигляді HTML-документа, він перетворить документ на сторінку та відобразить його користувачеві. Статичний додаток вважають простим так як при запиті на сервер готовий HTML документ просто зчитується з жорсткого диска і повертається клієнту. Альтернативний підхід називають динамічним – коли HTML код генерується у процесі обробки запиту. Зараз майже всі існуючі веб-додатки динамічні, вони виглядають інтерактивними, чуйні на дії клієнта. У цьому полягає завдання веб-додатка – генерувати необхідного HTML-код, залежно від дій користувача й у момент підготовки ресурсів на сервері.

Щоб описати роботу веб-програм, слід розділити вузли на сервери та клієнти. За серверну частину відповідає backend-розробка, а клієнтську – frontend.

Frontend мається на увазі створення візуальної частини програми, яка виконує функції на стороні користувача.

Backend пов'язаний з тим, що користувач візуально ніяк оцінити не

може і чого не має доступу. Це логіка роботи програми, що здійснюється на віддаленому сервері.

Розробка серверної частини програм пов'язана з низкою мов програмування (Java, Python, C#, C++), а спрощення такої розробки досягається використанням backend-фреймворків (Django) та веб-серверів (Nginx, Node.js та ін.).

Важливою частиною backend є бази даних. Вони потрібні для зберігання масивів даних, які при запиті користувача виймаються та відображаються у веб-застосунку. На практиці можуть використовуватись різні бази даних, найбільш популярні з яких: PostgreSQL, MySQL, MongoDB.

1.1.2 Мережева взаємодія

Неможливо говорити про веб-додатки не розглянувши модель мережевих шарів - OSI, Open Systems Interconnection [3]. Адже щоб запит на будь-який ресурс, вписаний в адресний рядок браузера, повернув відповідь з сервера користувачу, використовується складна структура передачі мережевих даних. У моделі OSI описано 7 рівнів:

- фізичний (physical);
- канальний (data link);
- мережевий (network);
- транспортний (transport);
- сеансовий (session);
- презентації (presentation);
- прикладний (application).

Фізичний рівень є найнижчим. Він працює безпосередньо із середовищем передачі, сигналами та двійковими даними. Наприклад, через оптоволоконний кабель передаються пучки світла, які інтерпретуються машиною як нулі чи одиниці. Також може використовуватись радіоканали,

USB, Registered Jack.

Канальний рівень відповідає за з'єднання пристроїв однієї, локальної мережі та відповідає за фізичну адресацію. Два найчастіше використовувані способи підключення до локальної мережі – це технологія Ethernet, коли комп'ютери з'єднуються кабелем, і бездротова технологія Wi-Fi. Щоб різні пристрої могли відрізнятися між собою в локальній мережі, вони мають MAC-адреси, які є унікальними ідентифікаторами пристрою. MAC-адреса виглядає як шість 16-річних однобайтних чисел, розділених двокрапкою або тире: 00-15-5D-DA-B0-94.

Мережевий рівень відповідає за визначення маршруту та логічну адресацію, надає можливість спілкування пристроям із різних мереж. Використання IP-протоколу дозволяє присвоювати мережевим пристроям певні адреси та визначати їхню приналежність до конкретної мережі. IP-адреса складається з чотирьох однобайтних чисел, які записуються в десятковій системі. Типовий приклад: 127.0.0.1. На підставі адрес пристроїв мережі комп'ютер може взаємодіяти та підключатися до сайтів, серверів. Протоколи мережного рівня: IP/IPv4/IPv6 (Internet Protocol), IPX (Internetwork Packet Exchange, протокол між мережевого обміну), IPsec (Internet Protocol Security) та ін. Найбільш популярні протоколи 4-ої та 6-ої версій.

Транспортний рівень дозволяє ідентифікувати адресатів і змішувати потоки даних. Відповідає за Прямий зв'язок між кінцевими пунктами та надійність. Щоб трафіку не змішувалися, кожна програма використовує для мережного спілкування певний порт. Наприклад, браузер для відкриття сайту використовує порт 80, а для з'єднання з базою даних може відкриватися порт 8080. Найбільш популярні на транспортному рівні 2 протоколи: TCP та UDP. TCP гарантує отримання всієї інформації, що передається, наприклад, при завантаженні документа весь документ скачається повністю, щоб не було втрат. UDP протокол такої гарантії отримання цілісних даних не дає. Ніхто не помітить якщо у прямому ефірі новин, кілька кадрів загубиться.

Сеансовий рівень відповідає за підтримання сеансу зв'язку, дозволяючи

програмам взаємодіяти між собою тривалий час. Рівень керує створенням/завершенням сеансу, обміном інформацією, синхронізацією завдань, визначенням права передачі даних і підтримкою сеансу у періоди не активності додатків. Синхронізація передачі забезпечується приміщенням потоку даних контрольних точок, починаючи з яких відновлюється процес при порушенні взаємодії. Важливим застосуванням є передачі в прямому ефірі, в яких необхідно без різких переходів накладати звуковий та відео потоки і переходити від одного потоку до іншого, щоб уникнути перерв в ефірі.

Рівень презентації відповідає за перетворення протоколів та кодування/декодування даних. Запити програм, отримані з рівня програм, він перетворює на формат передачі по мережі, а отримані з мережі дані перетворює на формат, зрозумілий програмам. На цьому важливому рівні може здійснюватися стиснення/розпакування або кодування/декодування даних, а також переправлення запитів іншому мережному ресурсу, якщо вони не можуть бути локально оброблені. Прикладом є переклад різних кодувань символів, наприклад, ASCII в EBCDIC.

Прикладний рівень – це те, що забезпечує взаємодію мережі та користувача. Головне завдання даного шару – надати клієнту зручний інтерфейс для взаємодії з мережею та пристроями. На цьому рівні використовується величезна кількість різноманітних протоколів, серед яких:

- HTTP – застосовується браузером для отримання даних з Інтернету;
- RDP – для забезпечення віддаленої роботи користувача з сервером;
- FTP – передачі з файлового сервера на ПК кінцевого користувача;
- BitTorrent – піринговий протокол передачі файлів, вони розбиваються на невеликі частини й у вигляді передаються через мережу;
- SMTP – вид протоколу, який може встановлювати правила передачі віртуальної пошти. Він відповідальний за передачу та верифікацію доставки, а також оповіщення про можливі помилки.

1.1.3 Механізм роботи веб-сервера

В основі функціонування веб-застосунків лежить таке поняття як веб-сервер. Веб-сервер – це програма, яка приймає вхідні HTTP-запити, обробляє ці запити, генерує HTTP-відповідь та надсилає його клієнту. Після того, як користувач звернувся до певного ресурсу протоколу HTTP, браузер формує HTTP-запит до веб-сервера. Браузер попередньо перетворює це ім'я на IP-адресу за допомогою сервісів DNS. Завдання веб-сервера – прослуховувати певний TCP-порт, зазвичай це порт 80 і приймати всі HTTP-повідомлення. Якщо вхідні дані не відповідають формату повідомлення HTTP, такий запит ігнорується, а клієнту повертається повідомлення про помилку.

Однією з найважливіших завдань, які вирішуються при побудові веб-сервера є завдання забезпечення масштабованості – можливості збільшення кількості клієнтів, що обслуговуються, і захищеності від різних атак. Оскільки веб-сервер працює у відкритому середовищі – глобальній мережі Інтернет – то найчастіше доступ до нього може здійснюватися звідки завгодно. Це робить сервер схильним до великих навантажень і потенційних атак. Найбільш поширеними атаками на сервер є звернення до сервера з великою кількістю запитів та їх високою частотою. У цьому випадку сервер не зможе швидко обробляти всі запити, а це може вплинути на продуктивність сервера. Такі атаки називають DDOS-атаками. Зазвичай для боротьби з DDOS атаками блокуються всі запити, які приходять з певної IP-адреси, або встановивши на сервер готове рішення захисту від DDOS атак від таких великих компаній як Cisco або Microsoft та ін. У разі атак слід подбати про оптимізацію програмного коду , наприклад, використовувати кешування – у разі при обробці кожного запиту навантаження на центральний процесор буде менше, що може значно ускладнити завдання атакуючим.

Часто на тому самому веб-сервері розташовується безліч незалежних

веб-сайтів. І всі ці веб-сайти використовують ту саму IP-адресу. А саме веб-сервер, що має тільки одну IP-адресу, може розміщувати в собі кілька сайтів і при цьому кожен такий сайт буде асоційований з власною адресою. Таке явище називається віртуальним хостингом. Роздільна здатність IP-адреси відбувається за допомогою служб DNS. Однак, незважаючи на те, що запит надсилається, використовуючи отриману IP-адресу, клієнт вказує додатковий заголовок HTTP «Host», в якому визначається оригінальне ім'я сайту. Завдяки цій інформації сервер може розмежувати доступ до кількох сайтів і при цьому використовувати ту саму IP-адресу. Це дуже важливо, оскільки якби для кожного доменного імені доводилося б реєструвати окрему IP-адресу, то адресний простір протоколів IP дуже швидко закінчився б, а вартість розміщення сайту в глобальній мережі Інтернет була б набагато вищою. Адресний простір протоколу IPv4 останнього доступного реєстратора AFRINIC закінчився у 2020 році. Перехід на IPv6 неминучий.

Схему віртуального хостингу використовує більшість компаній, що займаються розміщенням сайтів в Інтернеті, адже цей спосіб один із найдешевших. В рамках віртуального хостингу зазвичай заборонено запускати різні служби та послуги, а також існує обмеження за ступенем використання центрального процесора. Оскільки іноді від сервера потрібна велика кількість ресурсів або в рамках цього сервера необхідно запускати додаткові програми або служби, віртуальний хостинг можна використовувати не завжди. У цьому випадку зазвичай орендують виділений сервер - фізичний або віртуальний, але це дорожчий вид розміщення веб-програм в мережі Інтернет.

Програмний код, що обробляє HTTP-запити та генерує HTTP-відповіді, можна умовно розділити на дві частини:

- у програмний код, що реалізує службові функції взаємодії через протокол HTTP;
- програмний код, що реалізує логіку конкретного веб-додатку, це бізнес-логіка, звернення до СУБД тощо.

Оскільки програмний код web-програми зазвичай пакується в окремі модулі і поставляється незалежно, то потрібні механізми взаємодії цих двох частин - інтерфейс взаємодії. CGI, ISAPI, ICAPI і NSAPI – існуючі інтерфейси взаємодії, під якими розуміється набір правил, якими сервер і додаток взаємодіятимуть друг з одним.

CGI (Common Gateway Interface) – найбільш ранній спосіб взаємодії сервера та веб-програми. Основна ідея, яка лежить в основі CGI полягає в тому, що при надходженні чергового HTTP-запиту сервер ініціює створення нового процесу і передає йому всі необхідні дані HTTP-запиту. Оскільки сервер і додаток – це різні процеси з погляду операційної системи, то обміну інформації з-поміж них використовуються засоби між процесної взаємодії (IPC) – найчастіше це змінні оточення, іменовані канали тощо. Основною перевагою CGI є те, що процес сервера та програми ізольовані один від одного і у разі неполадок у веб-додатку, завершиться помилково саме процес програми, при цьому процес самого сервера продовжуватиме функціонувати. З іншого боку, необхідність створення кожного разу нового процесу спричиняє додаткові накладні витрати на створення процесу. Цей факт є серйозним недоліком і істотно впливає на масштабованість веб-програми.

ISAPI (Internet Server API) – альтернативний спосіб взаємодії сервера та веб-додатки, що дозволяє здійснювати доступ до функцій та служб сервера Internet Information Server (IIS) фірми Microsoft. NSAPI – сервера фірми Netscape, а ICAPI – сервера Internet Connection Server фірми IBM. На відміну від CGI, при взаємодії в рамках інтерфейсу ISAPI, при надходженні чергового запиту сервер ініціює створення нового потоку в рамках основного процесу, в якому працює сервер. Оскільки з погляду операційної системи створення потоку – це менш дорога операція, ніж створення процесу, такі додатки практично виявляються масштабованими. Однак, у разі серйозних неполадок у веб-застосунку, що взаємодіє з сервером в рамках ISAPI, сервер також потенційно наражається на ризик бути завершеним. Оскільки сервер і веб-додаток працюють в тому самому процесі, це дійсно так. Тому

розробникам програмного коду сервера, що підтримує ISAPI, слід приділити цьому питанню особливу увагу.

На сьогоднішній день найбільш поширеним способом взаємодії сервера та веб-додатки є інтерфейс ISAPI, оскільки забезпечує найбільш оптимальні показники за накладними витратами та масштабованістю. Але зараз популярно використовувати суміщений підхід – для кожної програми може створюватися пул програми (application pool), який є окремим процесом, в якому функціонують потоки для обробки вхідних HTTP-запитів від користувачів. У цьому випадку, якщо якийсь із програм буде містити код, який завершує роботу процесу з помилкою, то завершуватиметься процес лише цієї програми. Більш того, кожен пул програми містить набір заздалегідь створених та підготовлених потоків. Це необхідно для того, щоб не витрачати час на створення потоку в момент надходження запиту. Такий набір заздалегідь створених потоків називається пулом потоків. Як правило, сервер стежить за кожним пулом програми і якщо він завершує свою роботу з помилкою, сервер перезапускає його процес.

Раніше згадувався мережевий сервіс DNS. Він та інші мережеві сервіси призначені спростити взаємодію комп'ютерів у мережі з погляду користувача. DNS розшифрується як Domain Name System і призначений для трансформації буквеної адреси сайту на його IP-адресу. Так, коли користувач вводить google.com, DNS перетворює його на 216.58.205.206.

Коли ви запитуєте адресу якоїсь сторінки в Інтернеті, то крім даних для веб-сервера, ділитеся ще й IP-адресою. Цим легко можуть скористатися зловмисники. Для захисту даних було створено мережевий сервіс NAT-Network Address Translation, перетворення мережевих адрес. З його допомогою запит до віддаленого сервера йде не від імені вашого ПК, а реальна адреса комп'ютера маскується роутером або файрвол і передається в прихованому вигляді. Запит відправлятиметься не від окремого ПК, а від загального роутера, тому дізнатися адресу конкретної машини практично неможливо.

Будь-який комп'ютер у мережі повинен мати унікальну IP-адресу. Але вручну дати унікальну адресу ПК дуже складно. Тому компанія Microsoft розробила технологію DHCP (Dynamic Host Configuration Protocol, протокол динамічного налаштування вузла), яка дозволяє динамічно привласнювати IP-адресу з доступних діапазону. Завдання значно спрощується: адміністратору потрібно лише один раз задати діапазон значень, які самі будуть підставлятися у міру звернення комп'ютера до мережі.

1.1.4 HTTP-протокол

HTTP (HyperText Transfer Protocol) – один із найважливіших протоколів, який забезпечує передачу даних через інтернет. Протокол HTTP розташований на сьомому, прикладному рівні моделі OSI і працює на основі протоколу TCP.

Програмне забезпечення для роботи з HTTP-протоколом ділять на 3 категорії:

- сервери, що обробляють запити;
- клієнти, надсилають запити, споживають інформацію;
- проксі, транспортні служби.

HTTP-протокол має кілька версій. Найбільш використовуваною нині вважається HTTP/1.1 (у цій редакції протоколу введено режим постійного з'єднання TCP). HTTP/2 введений у вжиток у 2015 році і дозволяє ущільнювати канал (передавати більшу кількість запитів за наявних потужностей). У 2019 році з'явилася третя версія протоколу, яка передбачає заміну TCP на UDP як основний транспортний протокол.

Щоб сервер зрозумів, чого хоче клієнт, використовуються HTTP-запроси (методи):

- GET для отримання необхідних даних від сервера;
- HEAD, аналогічний GET, але тіло відповіді залишається завжди

порожнім, дозволяє перевірити доступність запитуваного ресурсу та прочитати HTTP-заголовки відповіді;

- OPTIONS, визначає можливості сервера;
- POST, дозволяє завантажити інформацію на сервер, за змістом змінює ресурс на сервері, але часто використовується і для створення ресурсу на сервері, тіло запиту містить ресурс, що змінюється/створюється;
- PUT, аналогічний POST, але за змістом займається створенням ресурсу, а чи не його зміною, тіло запиту містить створюваний ресурс;
- PATCH, перезаписує лише частину ресурсу;
- DELETE для видалення даних;
- CONNECT, для створення зашифрованого тунелю між клієнтом та сервером з використанням проксі;
- TRACE, дозволяє відстежувати інформацію, що змінюється проміжними серверами у запитах;
- Найчастіше використовувані практично запити – GET і POST.

HTTP-відповідь містить статусний код або код повернення. Кожна група статусних кодів ідентифікує ситуацію, в якій запит. Група визначається першим розрядом статусного коду. Код групи 1xx є інформаційними кодами, 2xx каже про успішне виконання HTTP-запиту. 3xx означає переправлення, тобто. Щоб виконати запит, потрібні додаткові дії. 4xx означає, що проблема пов'язана не з сервером, а із запитом. Найчастіші 401 – немає доступу та 404 – ресурс не знайдено. 5xx означає помилки на сервері.

Незважаючи на простоту протоколу, існує проблема витоку інформації, що передається. Оскільки інформація передається у вигляді звичайного тексту, перехоплення такої інформації здійснюється досить просто. У деяких ситуаціях ця проблема не критична. Проте, для веб-додатків, що працюють з конфіденційною інформацією, це досить істотний недолік.

Тому існує модифікація цього протоколу – HTTPS, тобто. протокол HTTP із підтримкою шифрування.

Для шифрування і розшифрування даних використовується той самий ключ – якщо хтось знає ключ до зашифрованої інформації, він може розшифрувати її. Ключ - це нормальна послідовність біт певної довжини. Чим більша довжина ключа, тим складніше зламати алгоритм шифрування. Таким чином, щоб захистити свою інформацію, необхідно зберігати в секреті ключ шифрування. Також використовують додатково інший вид шифрування – асиметричний. Тут існує пара ключів – відкритий і закритий. За допомогою відкритого ключа можна лише зашифрувати інформацію, а за допомогою закритого – розшифрувати. Зазвичай, при такому підході закритий ключ зберігається в секреті, а відкритий ключ є загальнодоступним. Однак асиметричний алгоритм працює повільніше, ніж симетричний, тому його використовують для первинного обміну симетричними ключами.

При зверненні клієнта до сервера захищеного каналу сервер зберігає відкритий і закритий ключ. У початковий час сервер передає клієнту відкритий ключ асиметричного шифрування. Клієнт випадково генерує ключ симетричного шифрування і шифрує його за допомогою відкритого ключа, отриманого від сервера. Після цього клієнт надсилає зашифрований ключ на сервер і в цей момент клієнт і сервер мають однакові ключі для симетричного шифрування. Далі йде HTTP-взаємодія, яка шифрується за допомогою цього симетричного ключа. Симетричний ключ залишається в секреті і не може бути перехоплений, оскільки закритий ключ (яким можна розшифрувати перше повідомлення, що містить симетричний ключ), залишається в секреті на сервері. Таким чином, забезпечується конфіденційність і цілісність даних, що передаються за протоколом HTTP.

1.1.5 Типи web-додатків

Не існує загальноприйнятої класифікації веб-додатків. Ми часто згадували що веб-додаток складається з клієнтської та серверної частин,

дуже рідко зустрічається що додаток суто клієнтський або суто серверний, найчастіше це їх комбінація, розглянемо ці два типи додатків.

Клієнтські програми в чистому вигляді не вимагають серверної частини та обходяться можливостями JavaScript, використовуючи як оболонку браузер користувача. Вони не зберігають результату своєї роботи довше однієї сесії. Приклад чисто клієнтської програми – будь-яка онлайн гра, де після перезавантаження сторінки інформація про результат не зберігається.

Серверні веб-програми працюють на віддалених комп'ютерах. Для їх написання використовують такі мови програмування: Python, Java, Ruby, PHP, C# та ін. Вони практично не вимагають втручання користувача. Приклад чисто серверної програми - push-сповіщення.

Клієнтська та серверна сторони становлять архітектуру програмного забезпечення веб-програми. Можна виділити кілька типів архітектури веб-застосунків, залежно від того, як логіка програми розподіляється між клієнтською та серверною сторонами. Найпоширеніші архітектури веб-додатків:

- SPA, Single Page Application, односторінкові веб-програми;
- MPA, Multi Page Application, Багатосторінкові веб-програми;
- PWA, Progressive Web Application, прогресивні веб-програми;
- Serverless або безсерверна архітектура;
- Архітектура мікросервісів.

SPA реалізують складний функціонал у межах одного вікна браузера без перезавантаження. Динамічне оновлення вмісту сторінки досягається технологією AJAX (Asynchronous JavaScript and XML, асинхронний JavaScript та XML). У поєднанні з фреймворками Angular, React, Vue.js робота таких програм стає максимально плавною. Майже всі поштові послуги є SPA. Відомі SPA: Gmail, Facebook, Twitter, Slack. Переваги SPA шаблон це висока швидкість роботи, кешування даних, швидка розробка. Недоліком є навантаження на браузер, можливі витоки пам'яті та вразливість у безпеці.

MPA використовуються для побудови складних систем. У цьому випадку будь-які зміни даних призводять до повного перезавантаження сторінки. Незважаючи на те, що вони вимагають більших обсягів ресурсів для реалізації та суттєво дорожчі, інші види web-додатків їх замінити не можуть. Загальна частка MPA поступово знижується.

Стандартний приклад такої програми - інтернет магазини з великим масивом товарів – Amazon, eBay. Переваги шаблону MPA це звичність відвідувачів, легка оптимізація. Недоліком є складна розробка та підтримка, суттєві фінансові вкладення та дороге утримання.

PWA – новий спосіб «подачі» web-сервісів, який максимально зближує їх зі звичайним, звичним настільним додатком, але на якісно вищому рівні. Тепер і на ПК, і в телефоні ви зможете отримувати сповіщення, працювати без доступу до мережі, незалежно від моделі пристроїв та їх потужностей. Головна сфера застосування таких програм – мобільні пристрої. Користувачеві більше не потрібно входити на AppStore або PlayMarket, щоб завантажити програму – все зробить браузер автоматично. З появою нового контенту PWA надсилає користувачеві push-повідомлення. Слід визнати, що незабаром ці програми можуть замінити практично всі мобільні аналоги. Головний приклад програми PWA - AliExpress. Переваги PWA шаблону це Швидка розробка, висока швидкість завантаження, робота без доступу до мережі. Недоліком є відсутність підтримки у деяких бразуерів та платформ.

Безсерверна архітектура дозволяє створювати та запускати програми та послуги, не турбуючись про сервери[4]. При безсерверних обчисленнях додаток працює на серверах, але управління цими серверами хмарні технології повністю бере на себе. За допомогою хмарних технологій та їх безсерверної платформи можна створювати та розгортати програми, використовуючи економічні сервіси, які забезпечують вбудовану доступність додатків та гнучкі можливості масштабування. Це дозволяє зосередитися на розробці коду програми, не турбуючись про виділення, налаштування серверів та управління ними.

Щоб зрозуміти архітектуру мікросервісів, краще порівняти її з монолітною моделлю. Традиційна монолітна архітектура веб-програми складається з трьох частин - бази даних, клієнтської та серверної сторін. Це означає, що внутрішня та зовнішня логіка, як і інші фонові завдання, генеруються в одній кодовій базі. Щоб змінити або оновити компонент програми, розробники програмного забезпечення повинні переписати всі програми [5]. Що стосується мікросервісів, цей підхід дозволяє розробникам створювати веб-додаток із набору невеликих сервісів. Розробники створюють та розгортають кожен компонент окремо. Архітектура мікросервісів є вигідною для великих і складних проектів, оскільки кожен сервіс може бути змінений без шкоди для інших блоків. Тому, якщо вам потрібно оновити логіку оплати, вам не доведеться на якийсь час зупиняти роботу сайту. Відомі проекти: Netflix, Uber, Spotify, PayPal.

1.2 Критерії оцінки

Якісна архітектура робить процес розробки та супроводу програми більш простим та ефективним. Програму з гарною архітектурою легше розширювати та змінювати, а також тестувати, налагоджувати та розуміти. Тобто насправді можна сформулювати список цілком розумних та універсальних критеріїв:

- відмовостійкість;
- гнучкість;
- розширюваність;
- масштабованість розробки;
- безпека;
- спостереження;
- тестування;
- структурований та код що можливо прочитати.

Відмовостійкість це здатність системи продовжувати повноцінно працювати при виході з ладу окремих компонентів - серверів або каналів зв'язку, збоїв на рівні окремих модулів системи тощо.

Гнучкість - це здатність швидко та зручно внести зміни до існуючого функціоналу. Чим менше проблем і помилок це викличе — тим гнучкіша і конкурентоспроможна система.

Розширюваність – можливість нарощувати в системі нові сутності та функції, не порушуючи її основної структури. Архітектура повинна дозволяти легко додавати додатковий функціонал у міру потреби. Причому так, щоб внесення найімовірніших змін вимагало мало зусиль.

Масштабованість розробки – можливість прискорити термін розробки за рахунок додавання до проекту нових ресурсів. Іншими словами, розпаралелювання процесу розробки між ресурсами, щоб безліч команд могло працювати над одним проектом одночасно.

Безпека - це захисні заходи, у яких зловмисник зможе отримати доступ до конфіденційним даним як ззовні при спробі злому, і усередині компанії через нелегітимний доступ.

Спостереження – можливість зрозуміти поведінку програми та усунути неполадки.

Тестування – писати код в такий спосіб, щоб розробники самі могли його протестувати.

Структурований і код що читається - правильна архітектура повинна давати можливість легко і швидко розібратися в системі новим людям. Проект має бути добре структурований, не містити дублювання, мати добре оформлений код та документацію. Контролери – електронні пристрої, що контролюють роботу зчитувачів і керуючі пристроями виконавчими, бувають одне функціональними і багатофункціональними.

1.3 Постановка задачі

Метою даної дипломної роботи є розглянути загальну інформацію щодо функціонування веб-додатків, розглянути основні протоколи та механізми, які задіяні у процесі роботи веб-додатків. Скласти порівняльний аналіз монолітної та мікросервісної архітектури веб-додатків. Детально розглянути патерни проектування, розгортання та тестування мікросервісів, а також порівняння ефективності патернів між собою. Скласти рекомендації щодо проектування архітектури веб-додатків архітектури спираючись на переваги та недоліки.

Ця тема є актуальною, оскільки зараз триває активний перехід від настільних додатків до браузерних рішень. Мікросервісна архітектура дозволяє проводити горизонтальне масштабування програми, просте обслуговування та дозволяє використовувати різні технології для різних сервісів.

Для досягнення поставленої мети необхідно вирішити такі завдання:

- сформувати уявлення про функціонування Web-додатків та на основі отриманої інформації висунути критерії оцінки предметної області;
- розглянути існуючі архітектурні рішення веб-додатків та провести їх порівняльний аналіз;
- спроектувати математичну модель розподіленої мікросервісної архітектури веб-додатка;
- провести аналіз мікросервісної архітектури. Визначити особливості, тенденції, переваги та недоліки, а також скласти звіт про дослідження.

2 АРХІТЕКТУРА МОНОЛІТНОГО ТА МІКРОСЕРВІСНОГО ПІДХОДІВ

2.1 Визначення поняття масштабованої веб-архітектури та її основні характеристики

Основна частина розробки веб-програми – це його масштабованість. Незалежно від того, який проект ви збираєтеся запустити, ви повинні бути готові до припливу користувачів і очікувати, що система впорається з цим. Потрібно брати до уваги те, що можуть бути випадки, коли ваша система недостатньо гнучка і не може витримувати більшого навантаження. Щоб запобігти цьому, важливо розпочати масштабування програм до того, як почнеться етап розробки.

У цьому розділі йтиметься про масштабоване веб-додаток, який може обробляти величезний потік даних без несподіваних збоїв. Почнемо з визначення масштабованості веб-програми та визначимо деякі з ключових принципів їх побудови.

Масштабованість – це здатність веб-програми справлятися зі зростаючим числом користувачів, що одночасно взаємодіють з додатком. Отже, веб-програма, що масштабується, - це програма, яка однаково добре працює з одним або тисячею користувачів і витримує злети і падіння трафіку.

Якщо говорити про фактори, що впливають на масштабованість веб-додатку, важливо виділити наступне:

- архітектура веб-додатків та правильно написаний код відіграють ключову роль у створенні масштабованих веб-додатків;
- фреймворк впливає на продуктивність програми, необхідно виберіть найбільш підходящий для вирішення бізнес вимоги з урахуванням різних факторів;
- навантажувальне тестування допомагає знайти та подолати слабкі сторони програми, щоб гарантувати його безперебійну роботу;

- сторонні послуги також повинні бути ретельно відібрані, інакше вони можуть призвести до збою в роботі.

Однак недостатньо просто написати програмне забезпечення або запустити сервер. Обидва вони вимагають регулярного обслуговування та управління. Щоб забезпечити обслуговування як програмного забезпечення, так і сервера, ми можемо використовувати платформи та інструменти, які полегшують відстеження програми, дозволяють застосовувати оновлення за необхідності та забезпечують ефективну роботу програми. У побудові системи необхідно враховувати основні принципи масштабованості. Багато хто з існуючих типів перекриваються і навіть іноді конфліктують один з одним, але їхня присутність має вирішальне значення для процесу проектування, і вони завжди повинні співіснувати в балансі. Оскільки ці типи пов'язані з принципами проектування, ми розберемо їх докладно нижче:

Обирайте горизонтальне масштабування замість вертикального. Ви можете виявити, що система іноді обмежена горизонтальним масштабуванням, проте ці обмеження регулярно розширюють бази даних в одному напрямку. Крім того, в горизонтальному масштабуванні можна просто додати інший сервер, а не оновлювати існуючий. Це заощадить вам небагато ресурсів.

Не навантажуйте одне ядро. Згодом кількість клієнтів перевищить кількість доступних вам послуг. Таким чином, щоб уникнути будь-яких вузьких місць при масштабуванні програми, розподіліть якнайбільше роботи від ядра.

Насамперед API. Завжди розглядайте веб-програму як службу API. Програми для смартфонів, веб-сайти з JS та настільні програми сьогодні дуже популярні, тому що клієнти підключаються через API. Оскільки API не розрізняє клієнтів, він може продовжувати обслуговувати будь-кого.

Кешуваня має бути обов’язковим оскільки кеш зберігає дані, які дозволяють швидше обслуговувати майбутні запити, вони значно покращують масштабованість та продуктивність.

Автоматизуйте процеси постачання та обслуговування програми. Якщо ви хочете, щоб система працювала ефективно щодня, ви повинні стежити за нею та регулярно надавати оновлення. Витрата часу та зусиль на підтримку програми забезпечує безперебійну роботу.

Дотримуйтесь програми, яка не залежатиме від будь-якого стану. Такий підхід є актуальним для розподіленої архітектури, яка підтримує горизонтальну масштабованість. Поки компоненти залишаються без стану, вони можуть легко перерозподілятися у разі поломки та масштабуватись для адаптації до змін навантаження.

При проектуванні великої веб-системи необхідно провести ретельне планування, яке допоможе довгостроковій перспективі. Нижче представлені основні принципи, що впливають на розробку високонавантажених веб-систем, що масштабуються.

В першу чергу доступність, бо час безвідмовної роботи сайту дуже важливий для репутації та функціональності багатьох компаній. Для деяких великих сайтів недоступність навіть протягом декількох хвилин може призвести до величезної втрати доходів, тому проектування їх систем, що забезпечують постійну доступність та стійкість до збоїв, є як фундаментальним бізнесом, так і технологічною вимогою. Висока доступність у розподілених системах вимагає ретельного розгляду надмірності для ключових компонентів, швидкого відновлення у разі часткових збоїв системи та поступового зниження продуктивності у разі виникнення проблем.

По друге, продуктивність сайту є важливим показником більшості сайтів. Швидкість веб-сайту впливає на роботу та задоволеність користувачів, а також ранжування пошукових систем, все це впливає на утримання аудиторії та дохід. Результатом хорошої продуктивності є створення системи, яка оптимізована для швидких відповідей та низьких затримок.

Система повинна бути надійною, щоб запит даних постійно повертав ті

самі дані. У випадку, якщо дані змінюються або оновлюються, той самий запит повинен повернути нові дані. Користувачі повинні знати, що якщо якісь дані записані в систему, то вони збережуться і їх можна буде використовувати надалі.

Коли йдеться про будь-яку велику розподілену систему, розмір системи є лише одним фактором, який необхідно враховувати. Не менш важливим є зусилля, необхідне збільшення пропускної спроможності системи, необхідної для обробки великих обсягів навантаження, зазвичай званої масштабованістю системи. Масштабованість може стосуватися багатьох різних параметрів системи: скільки додаткового трафіку вона може обробити, наскільки легко додати більше місця для зберігання та обробки даних.

Розробка системи, яка є простою в експлуатації, є ще одним важливим міркуванням у проектуванні розподіленої системи. Керованість системи прирівнюється до масштабованості операцій: обслуговування та оновлення. Щодо керованості, слід враховувати простоту діагностики проблем, що виникають у ході роботи, простоту внесення змін та оновлень, а також простоту експлуатації системи.

При розробці будь-якої веб-програми важливо враховувати наведені вище ключові принципи. Деякі їх можуть впливати більше, інші менше, але важливо врахувати кожен аспект, щоб уникнути дорогих проблем у майбутньому. Хоча інвестування ресурсів у повне масштабування системи, на початковому етапі, є нерозумним, важливо правильно вибрати архітектурний підхід у проектуванні системи з огляду на потенціал високої масштабованості в майбутньому. Все це допоможе заощадити ресурси та час пізніше. Існують основні фактори побудови веб-архітектури, що масштабується, що полегшують масштабованість при необхідності, які будуть досліджені в наступному розділі.

2.2 Основні типи архітектури сучасного програмного забезпечення

Розуміння найкращого способу розробки та розгортання програм сьогодні є важливим фактором для будь-якої організації, керованої даними. Такі опції, як сервіс-орієнтована архітектура (SOA) та мікросервіси, пропонують цінну гнучкість для створення та запуску додатків, які не підходять для традиційних монолітних підходів [6]. Однак варто розуміти різницю між ними, щоб визначити, що найкраще підходить для програми. У цьому розділі ми розглянемо різні типи архітектури, їх характеристики, і як вони використовуються для розробки програмного забезпечення.

Монолітна програма побудована як єдине ціле, в якому інтерфейс користувача та доступу до даних об'єднані в одну програму на одній платформі. Корпоративні монолітні докладання складаються з трьох частин: База даних, що складається з безлічі таблиць, зазвичай, у системі управління реляційними базами даних. Клієнтський інтерфейс користувача, що складається з HTML і JavaScript, запускається в браузері. Серверні програми, які працюють як посередник між інтерфейсом користувача і базою даних, будуть обробляти HTTP запити, виконувати деяку специфічну для домену логіку, витягувати і оновлювати дані з бази даних і заповнювати HTML-уявлення для відправки в браузер.

Труднощі при використанні монолітної архітектури виникають при масштабуванні програми. Щоразу, коли ви будуєте, тестуєте та впроваджуєте новий функціонал, ви повинні змінити весь моноліт, тому що модулі сильно залежать один від одного. Монолітна архітектура найбільш ефективна в невеликих проектах з чітко визначеною областю, де ви навряд чи підтримуватимете або розвиватимете кодову базу на постійній основі.

Сервісно-орієнтована архітектура в основному була створена як відповідь на традиційні монолітні підходи до створення програм. SOA розбиває компоненти, необхідні для додатків, на окремі сервісні модулі, які взаємодіють один з одним для досягнення конкретних бізнес-цілей. Кожен

модуль значно менший, ніж монолітний додаток, і його можна розгорнути для різних цілей на підприємстві [7]. Крім того, SOA поставляється через хмару і може включати послуги для інфраструктури, платформ і додатків.

Дві основні ролі SOA – це постачальник послуг і споживач послуг. Його рівень постачальника послуг включає різні послуги, залучені до SOA, тоді як рівень споживача працює як інтерфейс користувача.

SOA надає чотири різні види послуг:

- задля ведення бізнесу використовуються функціональні послуги;
- сервіси підприємства реалізують функціональність;
- служби прикладних програм призначені для розробки та розгортання програм;
- служби інфраструктури призначені для нефункціональних процесів, таких як безпека та аутентифікація.

Очевидний недолік полягає в тому, що, незважаючи на простоту окремих сервісів, архітектура може стати занадто складною, щоб відповідати вимогам бізнесу, що зростають. Тому, якщо не потрібно розділяти окремі функції вашої системи, то краще утриматися, оскільки це вимагатиме великих ресурсів.

Мікросервісна архітектура поділяє додаток на дрібніші, повністю незалежні компоненти, що забезпечує їм більшу гнучкість і масштабованість. Це логічна еволюція SOA, яка відповідає сучасним вимогам.

Мікросервіси вирішують проблеми застарілих монолітних систем. Цей тип архітектури складається з великої кількості невеликих сервісів, кожен з яких виконує власні функції і може бути незалежно розгорнутий, такий сервіс легше розуміти, розробляти і тестувати для забезпечення безперервної поставки і поліпшення. Архітектура мікросервісів підходить для хмарної платформи, тому якщо буде у цьому потреба це не викликає незручностей. Розгортання мікросервісів здійснюється в окремих контейнерах програмного забезпечення, якими керують оркестратори контейнерів.

Проведемо порівняльний аналіз монолітної та мікросервісної

архітектури за критеріями оцінки: проектування, розгортання, надійність, масштабованість, гнучкість, розробка, оновлення, тестування, постачання і безпека.

Таблиця 2.1 – порівняння основних типів архітектур зі характеристиками

	Монолітна	Сервісно-орієнтована архітектура	Мікросервісна
Проектуваннння	Монолітні програми розростаються до великих розмірів, і в таких умовах складно зрозуміти всю повноту програми	Набор сервісів і процесів, які можна комбінувати, а також змінювати з часом відповідно до змін вимог	Мікросервіси побудовані у вигляді невеликих додатків, що взаємодіють один з одним
Розгортання	Можливо розгортати та відкатувати кожен мікросервіс незалежно	Менша гнучкість у розгортанні	Виконуєтеться одноразове розгортання всієї системи один раз і за потреби вносите зміни
Відмовостійкість	Вся система може вийти з ладу через один збій	Кожна служба має забезпечити своєчасну доставку повідомлень.	Збій однієї служби не впливає на інші служби, тому додаток не завершується аварійно [8].

Продовження таблиці 2.1

Масштабованість	Масштабування монолітних додатків часто може бути важким	Можуть виникати складнощі з масштабуванням через залежності між сервісами та спільно використовувані модулі	Мікросервіси існують як незалежні одиниці розгортання і можуть масштабуватися незалежно від інших сервісів
Гнучкість	Важко досягти гнучкості при повторному розгортанні монолітних артефактів додатків. Не можна впроваджувати нові технології, мови програмування тощо	Велика кількість залежностей, що виникають через полегшення доступу до компонентів та ускладнюють управління	Для кожного мікросервісу можуть бути використовані різні технології та різні мови для різних бізнес-потреб і цілей
Розробка	Всі команди залучені до процесу розробки одночасно	Сервіс-орієнтована архітектура складається з шарів, тому вона виступає за паралельність у процесі розробки	Всі команди є окремими підрозділами, здатними розробляти різні елементи програми

Продовження таблиці 2.1

Оновлення	Повільні оновлення через внутрішні залежності й інші команди та розробники, які працюють одночасно	Оскільки кожна служба програмного забезпечення є незалежною одиницею, її легко оновлювати та підтримувати, не завдаючи шкоди іншим службам	Швидкі оновлення, оскільки кожен модуль/сервіс є автономним
Тестування	Можливе наскріз-не тестування	Загальне тестування постачальника сервісу, споживача сервісу та реєстру сервісів	Потрібне комплексне тестування кожного компонента окрема і взаємодію компонентів
Постачання	Через внутрішні залежності та розмір архі-тектури моноліти набагато важче розгорнути, що призводить до неоптимального часу постачання	Скороченний час реалізації про-єктів і виходу на ринок	Час постачання суттєво опти-мізован завдяки безперебійному розгортанню, безперервному тестуванню та мінімальним внутрішнім залежностям

Продовження таблиці 2.1

Безпека	Передача та обробка даних простіше завдяки загальному засобу безпеки	Деякі проблеми виникають на етапі захисту шини	Деякі проблеми з безпекою можуть виникнути через зв’язок між службами

На основі цих даних можна зробити висновок, що, при порівнянні з монолітною архітектурою, здається, що використання мікросервісів є кроком вперед. Головна перевага, у порівнянні з іншими архітектурами, в тому, що мікросервіси розбита на кілька окремих сервісів, кожен з яких може бути розгорнутий, а потім повторно розгорнутий незалежно. Але це може бути надто складним для того, що потрібно вашому бізнесу, тому що більш проста платформа може бути простішою у реалізації та економічнішою. Необхідно чітко розуміти ситуацію, в якій використання мікросервісів буде корисним для бізнесу.

2.3 Розрахунок математичної моделі порогового значення масштабування розподіленої мікросервісної системи

Архітектура додатків мікросервісів є лідуючим рішенням бо набагато зручніше розподіляти навантаження, створювати високодоступні розгортання та керувати оновленнями, одночасно полегшуючи розробку та керування командою розробки і тестування.

Питання оркестрації контейнерів вирішує автоматичне масштабування. Контейнери розготаються за розрахуванням на поточне навантаження тому не потребує постійної уваги розробників.

Важливі деталі такі як мінімальна і максимальна кількість реплік,

пороги збільшення та пониження, періоди синхронізації, затримки охолодження – усі ці налаштування дуже пов’язані між собою. Змінення одного, швидше за все, вплине на інше, але все одно доведеться організувати збалансовану комбінацію, яка відповідатиме як програмі і розгортанню, так і інфраструктурі. Для цього потрібно робити розрахунок параметрів оркестрації контейнера.

Коли мають на увазі автоматичне масштабування мікросервісів для програми, насправді розглядається покращення двох основних моментів:

- переконання що розгортання може швидко розширюватися в разі швидкого збільшення навантаження, щоб користувачі не стикалися з затримками або HTTP помилками 500;
- зниження вартості інфраструктури, тобто запобігання недовантажених екземплярів.

Це в основному означає оптимізацію порогових значень програмного забезпечення контейнера для збільшення та зменшення масштабу. Як приклад, алгоритм Kubernetes має один параметр для двох.

Щоб цей метод працював, потрібно переконатися, що веб-додоаток відповідає наступним вимогам:

- навантаження має бути рівномірно розподілене між кожним екземпляром додатку;
- час запиту має бути коротшим за інтервал перевірки завантаження кластера контейнерів.

Основна причина цих умов полягає в тому, що алгоритм розраховує навантаження не на одного користувача, а як розподіл.

Спочатку повинна бути сформулюване визначення швидкого збільшення навантаження або, іншими словами, найгіршого сценарію. Це хороший спосіб перекласти це: наявність великої кількості користувачів, які виконують потребуючі ресурси дії протягом короткого періоду часу – і завжди є ймовірність, що це станеться, коли інша група користувачів або служб виконує інші завдання.

Введення змінних:

- N_u – максимальна кількість користувачів;
- $L_u(t)$ – навантаження, що створюється одним користувачем, який виконує «ресурсоємну операцію», t = 0 вказує на момент, коли користувач починає операцію;
- $L_{tot}(t)$ – загальне навантаження (генерується всіма користувачами);
- T_{tot} – короткий період часу.

У математичному світі, якщо говорити про велику кількість користувачів, які виконують одну і ту ж роботу в один і той же час, розподіл користувачів у часі відповідає гауссовому (або нормальному) розподілу, формула якого така:

$$G(t) = \frac{1}{\sigma\sqrt{2\pi}} e^{\frac{-(t-\mu)^2}{2\sigma^2}} \quad (1.1)$$

Де:

- μ – очікуване значення;
- σ – стандартне відхилення.

На рисунку 2.1 бачимо що значення за межами інтервалу $[-3\sigma; 3\sigma]$ дуже близькі до нуля і не сильно змінюються, тобто їх вплив дійсно незначний і його можна відкласти. До того ж, у інтервалі $[-3\sigma; 3\sigma]$ містить 99,7 відсотка користувачів, це досить близько до загальної кількості.

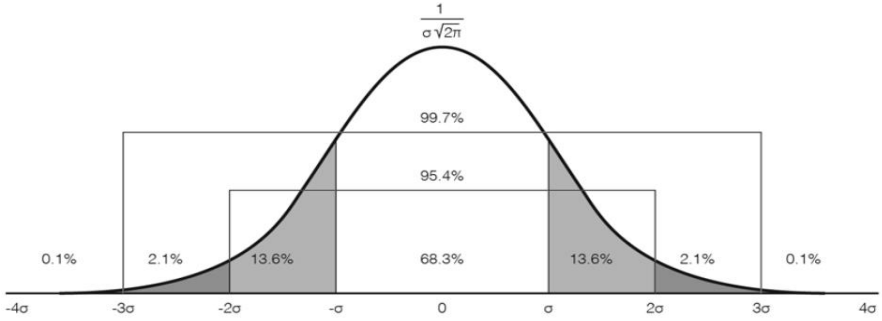

Рисунок 2.1 – Розподіл Гауса при μ = 0

Щодо листування до T_{tot}, дати йому значення 6σ ($[-3\sigma;\ 3\sigma]$) не буде гарним наближенням, оскільки 95,4 відсотка користувачів знаходяться в інтервалі $[-2\sigma;\ 2\sigma]$ яке є 4σ. Тож вибираючи T_{tot} дорівнювати 6σ додасть половину часу лише для 4,3% користувачів, що насправді не є репрезентативним. Тому було обрано взяти $T_{tot} = 4\sigma$, і ми можемо зробити висновок, що:

$$\sigma = \frac{T_{tot}}{4} \tag{1.2}$$

$$\mu = \frac{3}{4} \times T_{tot} \tag{1.3}$$

Після визначення цих констант, найгірший сценарій можно перекласти так – генероване навантаження на 99,7% N_u, виконує поглинаючу операцію $L_u(t)$ і де 95,4% з них роблять це протягом тривалості T_{tot} .

Вставляючи попередні результати в функцію розподілу користувача (Гаусса), ми можемо спростити рівняння наступним чином:

$$G(t) = \frac{4N_u}{T_{tot}\sqrt{2\pi}} e^{\frac{-(4t-3T_{tot})^2}{T_{tot}^2}} \tag{1.4}$$

Відтепер маючи σ і μ визначеними, ми будемо працювати в інтервалі $t \in \left[0, \frac{3}{2}T_{tot}\right]$, впродовж 6σ.

3 АНАЛІЗ ТЕХНОЛОГІЙ ТА ІНСТРУМЕНТІВ ДЛЯ СТВОРЕННЯ МІКРОСЕРВІСНОЇ АРХІТЕКТУРИ

Під час даної роботи було застосовано теоретичні та експериментальні дослідження. Теоретичні дослідження:

- аналіз;
- порівняльний аналіз;
- аналітичне дослідження.
- Експериментальні дослідження:
- пошукові;
- виробничі.

Додаток на основі мікросервісів є розподіленою системою, що працює на декількох процесах або сервісах, іноді навіть не кількох серверах або вузлах. Зазвичай кожен екземпляр служби це процес. Таким чином, служби повинні взаємодіяти за протоколом внутрішньопроцесної взаємодії, наприклад HTTP, AMQP або бінарного протоколу, такому як TCP, залежно від характеру кожної служби.

Клієнт та послуги можуть взаємодіяти через різні типи зв'язку в залежності від сценарію та цілей. Ці типи зв'язків можна поділити на два напрямки.

Перша група визначає, чи є протокол синхронним або асинхронним:

- синхронний протокол HTTP – це синхронний протокол. Клієнт відправляє запит і чекає на відповідь від служби. Це не залежить від виконання коду клієнта, яке може бути синхронним (потік заблокований) або асинхронним (потік не заблокований, відповідь зрештою буде надіслана). Тут важливо, що протокол (HTTP/HTTPS) є синхронним і код клієнта зможе продовжити виконання завдання лише після отримання відповіді від сервера HTTP;
- асинхронний протокол Інші протоколи, наприклад AMQP (протокол,

підтримуваний багатьма операційними системами та хмарними середовищами), використовують асинхронні повідомлення. Код клієнта або відправник повідомлення не очікує відповіді. Він просто надсилає повідомлення, як при надсиланні повідомлення в чергу RabbitMQ або іншого брокера повідомлень.

Друга група визначає, чи має запит одного або декількох одержувачів:

- один одержувач. Кожен запит повинен оброблятися лише одним отримувачем чи службою. Наприклад, шаблон Command;
- декілька одержувачів. Кожен запит може оброблятися різною кількістю одержувачів – від нуля до кількох. Такий тип взаємодії має бути асинхронним. Наприклад, механізм publish/subscribe, який використовується у таких шаблонах, як архітектура, керована подіями [9]. Він ґрунтується на інтерфейсі шини подій або брокері повідомлень, коли події оновлюють дані у кількох мікрослужбах. Зазвичай це реалізується через службову шину або подібний об'єкт, наприклад, службову шину Azure, за допомогою тем і підписок.

Додаток з урахуванням мікрослужб часто використовує комбінацію цих стилів взаємодії. Найпоширеніший тип — взаємодія з одним одержувачем за синхронним протоколом, наприклад HTTP або HTTPS, під час виклику звичайної служби веб-API HTTP. Для асинхронної взаємодії між мікрослужбами зазвичай використовуються протоколи повідомлень.

3.1 Мікросервісні комунікації REST API та черги повідомлень

Працюючи з мікросервісами досить важливий момент проектування як сервіси повинні спілкуватися один з одним. Багато людей зазвичай вибирають RESTful HTTP API, що надається кожною службою, а потім інші служби викликають його за допомогою звичайного HTTP-клієнта.

Це має деякі переваги, в першу чергу, полегшуючи виявлення служб за

допомогою дозволу DNS і шлюзів API, але також має багато недоліків. Наприклад, якщо викликаний сервіс вийшов з ладу і не може відповісти? Ваша клієнтська служба повинна реалізувати якусь логіку перепідключення або відновлення після відмови, інакше ви ризикуєте втратити запити та фрагменти інформації. Хмарна архітектура має бути стійкою та коректно відновлюватися після збоїв. Також HTTP-запит є блокуючим API і робить його асинхронним.

Використання черг повідомлень є якісним рішенням при роботі з декількома службами зв'язку. Архітектура черги повідомлень вимагає додаткової служби, яка називається брокером повідомлень, яка займається збором, маршрутизацією та розповсюдженням повідомлень від відправників до потрібних отримувачів.

Архітектуру черги повідомлень можна подати у вигляді поштової служби. Ви (producer) відправляєте лист (message) комусь (consumer), і ви робите це, вказуючи адресу (логіку маршрутизації для повідомлення, наприклад, тему, за якою воно опубліковане) та даючи лист до місцевого поштового відділення (брокер повідомлень). Після того, як ви залишите листа, вам більше не потрібно думати, чи дійде лист, про це подбає поштова служба. У мікросервісному середовищі це справді дуже надійне рішення, оскільки воно додає рівень абстракції (сам брокер повідомлень) між слабко зв'язаними сервісами та забезпечує повністю асинхронний зв'язок.

Передача репрезентативного стану (REST) була визначена Роєм Філдінгом у його докторській дисертації 2000 року, під назвою «Архітектурні стилі та проектування мережевих програмних архітектур». Доктор Філдінг працював у команді, яка працювала з протоколом HTTP. Працюючи над у цій сфері він перетворив свою модель на основний набір принципів, властивостей та обмежень, яка тепер називається REST [10].

Взаємодія RESTful стала життєво важливою для корпоративних обчислень, оскільки сьогодні вони підтримують безліч API в мережі. З урахуванням сказаного давайте визначимо, які проблеми найкраще вирішує

REST:

- синхронний запит/відповідь - HTTP (мережевий протокол, яким транспортується REST) сам по собі є протоколом запиту/відповіді, тому REST відмінно підходить для взаємодії запит/відповідь;
- загальнодоступні інтерфейси API. Оскільки HTTP є дефакто-транспортним стандартом завдяки роботі IETF, транспортний рівень API-інтерфейсів, створених з використанням REST, сумісний з будь-якою мовою програмування. Крім того, корисне навантаження повідомлення може бути легко задокументовано за допомогою таких інструментів як Swagger. І через широкий спектр загроз безпеки, присутніх в Інтернеті, екосистема безпеки для REST є надійною, від брандмауерів до OAUTH.

Через популярність сервісів RESTful сьогодні багато компаній потрапляють у пастку використання REST як інструменту «все в одному». Звичайно, деяка частина цієї популярності обумовлена міццю, яку REST надає на основі власних переваг. Розробники також звикли проектувати програми з синхронним запитом/відповіддю, оскільки API та бази даних навчили розробників викликати метод та очікувати негайної відповіді.

Ця надмірна залежність від використання REST та синхронних шаблонів має негативні наслідки, які в першу чергу відносяться до обміну даними між мікросервісами в рамках підприємства та в деяких випадках суперечать принципам правильної архітектури мікросервісів:

Тісний зв'язок – завжди буде деякий зв'язок сервісів навколо інтерфейсу (особливо навколо даних), але при виклику сервісу RESTful розробник припускає, що повідомлення будь-коли потрібно буде доставити лише одне місце. З'явиться проблема, коли в майбутньому з'явиться послуга або компонент, і вони знадобляться. Звичайно, ви можете оновити код, щоб додати нову кінцеву точку, але з'явиться зайве з'єднання. Незабаром ваш простий мікросервіс стане оркестратором, який ігнорує головний атрибут мікросервісу – єдиний у призначенні.

Блокування – якщо виклик REST сервісу, ваш сервіс блокується в очікуванні відповіді. Це знижує продуктивність програми, оскільки цей потік може обробляти інші запити. Подумайте про це так: що якщо бармен прийняв замовлення на напій, зробив коктейль і терпляче чекав, поки клієнт закінчить пити свій напій, перш ніж перейти до наступного клієнта? У цього клієнта буде відмінний досвід, але решта клієнтів відчуватиме спрагу і бути абсолютно нещасними. Бар може додати додаткових барменів, але потрібно один для кожного клієнта. Очевидно, що це буде дорого для бару, і його неможливо буде збільшувати та зменшувати, оскільки відвідувачі приходять та йдуть. Багато в чому такі ж проблеми виникають, коли потоки блокуються в додатках, що очікують відповіді служб RESTful.

Обробка помилок - HTTP було створено для роботи в Інтернеті і часто трапляється ситуація, коли браузер застрягне, намагаючись отримати доступ до веб-сторінки. Зазвичай ми натискаємо кнопку оновлення і сторінка відображається. Але що, якщо він знову зазнає невдачі? Спробувати поновити ще раз? Чи можна почати реалізовувати експоненційну людську форму, випивши чашку кави та повторивши спробу за кілька хвилин? Ми не знаємо, що робити, оскільки кожна веб-сторінка відрізняється і має унікальну поведінку. Така ж проблема виникає під час прямого виклику служби RESTful. Чи ця складна логіка повторення повинна знаходитися в коді сервісу? Якщо це станеться, сервіс буде ще більш тісно пов'язаний з іншими сервісами, що порушує ключовий принцип збереження архітектури мікросервісів єдиним призначенням і невеликим за розміром.

Рішенням багатьох недоліків, пов'язаних з RESTful/синхронними взаємодіями, є поєднання принципів керованої подіями архітектури з мікросервісами. Керовані подіями мікросервіси за своєю суттю асинхронні і повідомляються, коли час виконувати роботу. У багатьох випадках асинхронне спілкування – це кількість щоденних взаємодій. Розглянемо приклад роботи Facebook, було б неймовірно неефективно переходити до кожного друга і перевіряти, чи є у нього оновлення статусу. Натомість ми

отримуємо повідомлення, коли друг оновив свій статус або додав нову інформацію, що дає велику зручність у користуванні. Розглянемо переваги обміну повідомленнями для керованих подіями мікросервісів численні та різноманітні:

Слабкий зв'язок – якщо сервіси використовують обмін повідомленнями, зокрема функції публікації/підписки, служби не мають відомостей про інші служби. Вони повідомляються про нові події, обробляють цю інформацію та виробляють/публікують нову інформацію. Ця нова інформація може бути використана будь-якою кількістю служб завдяки публікації/підписці. Слабкий зв'язок дозволяє мікросервісам бути готовими до нескінченних змін, що відбуваються на підприємствах.

Неблокованність – мікросервіси повинні працювати максимально ефективно, без порожньої витрати ресурсів, коли багато потоків заблоковані та очікують відповіді. За допомогою асинхронного обміну повідомленнями програми можуть надсилати запит та обробляти інший запит замість очікування відповіді. Це стає зрозумілим під час повторного розгляду аналогії офіціанта у барі. Офіціанти – це люди, які можуть обслуговувати кількох клієнтів та чергувати виконання кількох завдань одночасно. Вони переходять від клієнта до клієнта та обробляють кілька замовлень, не блокуючи і не чекаючи жодного клієнта, тому кожен клієнт залишається задоволеним.

Простота масштабування. У міру зростання додатків та підприємств здатність динамічно масштабуватися стає однією з найважливіших переваг мікросервісної архітектури. Оскільки кожна служба невелика і виконує лише одне завдання, кожна служба повинна мати можливість збільшуватися або зменшуватися за необхідності. Управління подіями та обмін повідомленнями спрощують масштабування мікросервісів, оскільки вони не пов'язані та не блокуються. Це також дозволяє легко визначити, який сервіс є вузьким місцем, і додавати додаткові екземпляри тільки цього сервісу, замість того, щоб наосліп масштабувати всі сервіси, що може бути у випадку, коли

мікросервіси пов'язані між собою синхронним зв'язком. Можливість масштабування з використанням обміну повідомленнями була доведена такими компаніями, як Linkedin та Netflix.

Таблиця 3.1 – Порівняння черги повідомлень (MQ) та REST API за основними критеріями

	REST API	Message Queue
Використання	Може взаємодіяти з будь-якою мовою програмування та платформою.	Доступний майже будь-якою мовою програмування і підтримується всіма платформами
Масштабованність	Процес масштабування вимагає додаткового впровадження балансувальника навантаження, для розкидання тяг серед діючих екземплярів системи	Спрощує масштабування мікросервісів, оскільки може працювати з необмеженою кількістю екземплярів без будь-яких додаткових доопрацювань. Черга може працювати як з одним одержувачем, так і з безліччю.
Продуктивність	Працює синхронно, тобто ваш REST сервіс блокується в очікуванні відповіді. Це знижує продуктивність програми, оскільки цей потік може обробляти інші запити на REST сервісі.	Працює за допомогою асинхронного обміну повідомленнями, програми надсилають запит та обробляють інший запит замість очікування відповіді, це дає додатку високу продуктивність

Продовження таблиці 3.1

Відмовостійкість	Має меншу відмовостійкість, тому що вимагає правильної конфігурації балансувальника навантаження, який буде відстежувати працездатність кожного екземпляра. Якщо балансувальник вийде з експлуатації, система може повністю відвалитися.	Надає високу відмовостійкість. Якщо одна частина системи недоступна, інша може продовжувати взаємодіяти з чергою. Черги роблять ваші дані незмінними і зменшують кількість помилок, що виникають при відключенні різних частин вашої системи.
З’єднання	Має тісний зв'язок, оскільки існує певна залежність сервісів від інтерфейсу. При виклик REST сервіс має поняття хто його викликає і кому потрібно відправити дані, щоб відправити всі дані правильно. Будь-які кардинальні зміни в сторонніх сервісах зазвичай вимагають доопрацювання	Має слабкий зв'язок, який дозволяє мікросервісам бути готовими до нескінченних змін. Черга повідомляє про нові повідомлення, чекає на відповідь і повертає її. Вона нічого не знає про вміст і їй не важливо від кого і кому було надіслано повідомлення.

Продовження таблиці 3.1

Безпека	Може бути використаний протокол TLS, але його інтеграція вимагає внесення змін до програм одержувача та відправника, що може викликати додаткові труднощі.	Використовує протокол захисту повідомлень Advanced Message Security, без будь-яких додаткових змін на стороні програми

Як видно з таблиці, системи черги повідомлень добре вписуються в мікросервісну конструкцію, і рекомендована до впровадження у веб-додатки.

3.2 Огляд фреймворків передачі даних за допомогою черги повідомлень

Apache Kafka і RabbitMQ – це дві системи обміну повідомленнями на основі підписок з відкритим вихідним кодом, які легко впроваджуються в проекти. RabbitMQ – старіший інструмент, випущений у 2007 році, який був основним компонентом у системах обміну повідомленнями та SOA [11]. Сьогодні вона також використовується для потокових сценаріїв використання. Kafka - це новий інструмент, випущений в 2011 році, який з самого початку створювався для потокових сценаріїв.

RabbitMQ – це брокер повідомлень загального призначення, що підтримує такі протоколи, як MQTT, AMQP та STOMP. Він може працювати з високопродуктивними сценаріями використання, такими як обробка онлайн-платежів. Він може обробляти фонові завдання або виступати посередником повідомлень між мікросервісами.

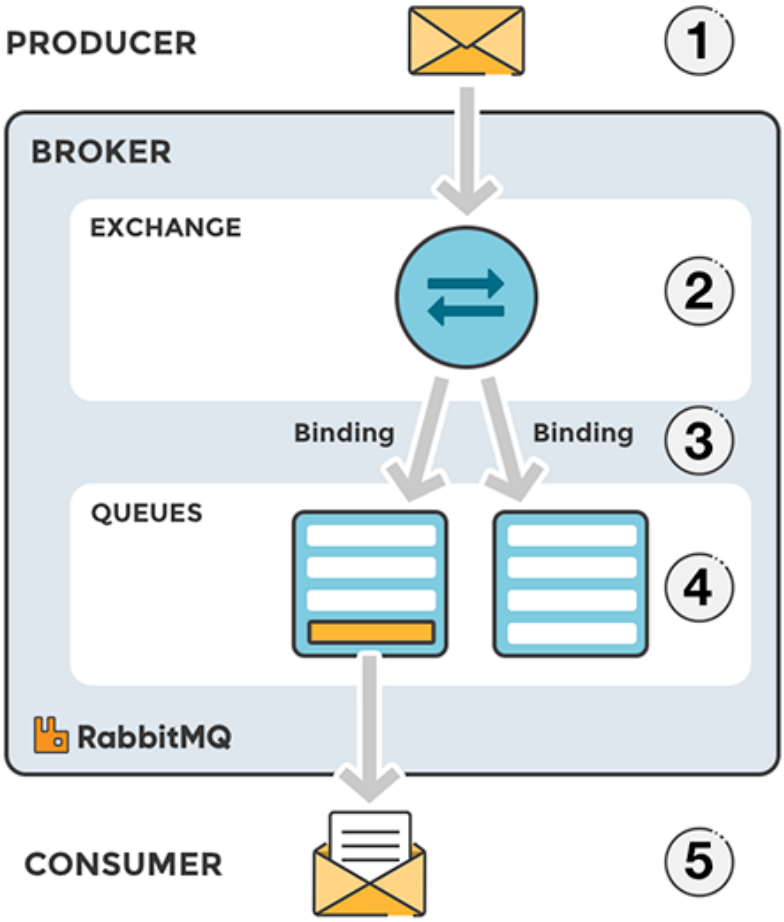

Рисунок 3.1 – Схема роботи брокеру повідомлень RabbitMQ

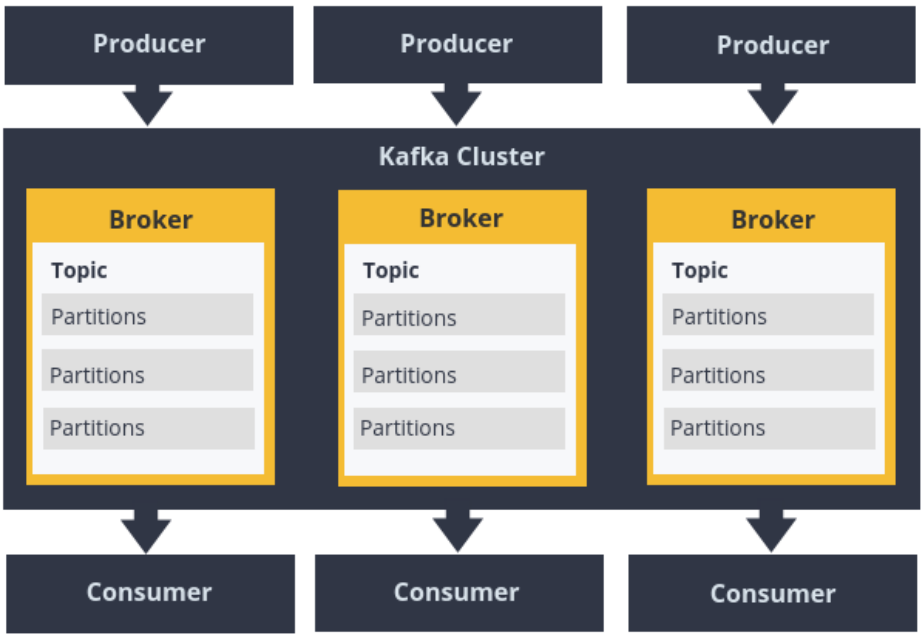

Рисунок 3.2 – Схема роботи шини повідомлень Kafka

Kafka – це шина повідомлень, розроблена для відтворення вхідних даних та потоків. Kafka – це надійний брокер повідомлень, який дозволяє програмам обробляти, зберігати та повторно обробляти потокові дані. Кафка має простий підхід до маршрутизації, який використовує ключ маршрутизації для надсилання повідомлень на топік [12].

Таблиця 3.2 – Порівняльна таблиця характеристик шин повідомлень

	RabbitMQ	Apache Kafka
Доступність	Надає кластеризацію та високо доступні черги	Надає сервер з відкритим вихідним кодом для управління станом кластера.
Розподіл та паралелізм	Можна масштабувати кількість одержувачів, це означає, що для кожного екземпляра черги ви матимете багато одержувачів, які конкурують за споживання повідомлення. У цій формі робота з обробки повідомлень поширюються всіма активними одержувачами, але все ж таки повідомлення може бути отримано лише один раз.	Спосіб розподілу споживачів здійснюється за тематичними розділами, і кожен одержувач із групи виділяється для одного розділу. Ви можете використовувати механізм розділів для надсилання кожному розділу різного набору повідомлень по бізнес-ключу

Продовження таблиці 3.2

Продуктивність	Також має високу продуктивність, яка дозволяє обробляти мільйон повідомлень за секунду, але потребує більше ресурсів.	Пропонує набагато більш високу продуктивність. Використовує послідовне дискове введення-виведення для підвищення продуктивності. Це може забезпечити високу пропускну здатність при обмежених ресурсах, що необхідне випадків використання великих даних.
Репликація	Черги не реплікуються автоматично, необхідно налаштувати. Якщо у вашому кластері є як мінімум три вузли, все, що вам потрібно зроби-ти, це оголосити свою чергу як черга кворуму, який подбає про резервне копіюванні	Брокер копіює, і якщо майстер-брокер не працює, автоматично вся робота передається іншому, який має повну копію померлого, повідомлення не втрачається.

Продовження таблиці 3.2

Приріст підписок	Повідомлення можуть бути надіслані в численні черги. У кожній черзі лише один одержувач із цієї черги може обробити повідомлення, але якщо повідомлення потрапляє до кількох черг, воно може бути оброблено кількома споживачами.	На повідомлення може бути підписано кілька одержувачів, тобто повідомлення може бути оброблено одночасно кількома одержувачами
Порядок повідомлень	Немає явного порядку повідомлень. Можна реалізувати за допомо-гою визначення множини черг і відправляючи кожне повідомлення в іншу чергу. Але, в масштабі, це рішення може бути скрутним.	Є розділи, де можна отримати порядок повідомлень у цьому модулі. Повідомлення надсилаються до топіків за ключом повідомлення, тому при виборі правильного ключа ви отримаєте один розділ (топік) для кожного ключа з впорядкованими повідомленнями.

Продовження таблиці 3.2

Протоколи повідомлень	Підтримує будь-які стандартні протоколи черги, такі як AMQP, STOMP (на основі тексту), MQTT (полегшений обмін повідомленнями публікації або підписки та HTTP	Підтримує примітиви (int8, int16, int32, int64, string, arrays) та двійкові повідомлення.
Час життя повідомлень	Представлений у вигляді черги, повідомлення видаляються після використання та надходить підтвердження. У RabbitMQ ви можете налаштувати постійні повідомлення, позначити чергу як постійну та повідомлення як постійні.	Представлений як журнал, повідомлення завжди знаходяться там, ви можете контролювати це, визначаючи політику зберігання повідомлень.
Гнучкість маршрутизації	Більше опцій, наприклад, за допомогою регулярних виразів та підстановочних знаків, документи перевіряються для отримання додаткової інформації.	Повідомлення відправляється в топік за ключом

Продовження таблиці 3.2

Пріоритет повідомлень	Можна визначити пріоритети повідомлень, а також використані повідомлення з високим пріоритетом.	Відсутнє. Може бути реалізовано за допомогою клавіш повідомлень, але в великих масштабах це може бути важко
Моніторинг	Є вбудований інтерфейс керування, доступний на вибраному порту. Також є вбудована інтеграція з популярними системами моніторингу Grafana та Prometeus	Можна використовувати сторонні інструменти (Confluent, Landoop, Kafka Tool)
Підтвердження	В обох платформах відправники отримують підтвердження того, що повідомлення надходить у чергу чи топік, а також одержувачі відправляє підтвердження, коли повідомлення успішно використовується, тому можна бути впевненим, що повідомлення не будуть втрачені по дорозі.	

Проналізувавши таблицю 3.2 висновки наступні, – треба використовувати Kafka у випадках, коли необхідна:

- багато одержувачів на те саме повідомлення;
- висока пропускна здатність (мільйони повідомлень за секунду);
- потокове оброблення;

- резервне копіювання черг;
- висока доступність;
- важливо порядок повідомлень.

Використовування RabbitMQ потрібно у випадках, коли необхідна:

- один одержувач на те саме повідомлення;
- гнучка маршрутизація;
- черга пріоритетів;
- використання повідомлень стандартного протоколу.

На закінчення можна сказати, що RabbitMQ досить хороший для простих випадків використання черги повідомлень, з невеликим трафіком даних. Є певні переваги, такі як пріоритетна черга та гнучкі параметри маршрутизації. Але для масивних даних та високої пропускної спроможності використовуйте Кафку без суперечок. Також, якщо вам потрібен журнал логів або кілька одержувачів для тих самих повідомлень, перейдіть в Kafka, тому що RabbitMQ не зможе допомогти з цим.

RabbitMQ – це безкоштовне рішення для організації черг повідомлень. Це брокер повідомлень, який розуміє AMQP (розширений протокол черги повідомлень), але може використовуватися з іншими популярними рішеннями обміну повідомленнями, такими як MQTT. Він високодоступний, стійкий до відмови і масштабуємо. RabbitMQ реалізований в Erlang OTP, технології, розробленої для побудови стабільних, надійних, відмовостійких і добре масштабованих систем, які мають вбудовані можливості обробки дуже великої кількості одночасних операцій.

При проектуванні веб-системи, ми можемо представити RabbitMQ як рівень проміжного програмного забезпечення, який дозволяє різним службам у вашому додатку взаємодіяти один з одним, не турбуючись про втрату повідомлень, забезпечуючи різні вимоги до якості обслуговування. Він також забезпечує тонку та ефективну маршрутизацію повідомлень, що забезпечує широке поділ додатків. RabbitMQ буде особливо корисним, коли потрібна архітектура, яка є більш гнучкою і адаптується до потреб багатьох

мікросервісів, що постійно змінюються.

Суть роботи RabbitMQ полягає в тому, що він приймає і пересилає повідомлення. Ви можете думати про це як про поштове відділення: коли ви поміщаєте пошту, яку ви хочете опублікувати, в поштову скриньку, ви можете бути впевнені, що листоноша в результаті доставлять пошту вашому одержувачу. У цій аналогії RabbitMQ - це поштова скринька, поштове відділення та листоноша.

Розглянемо основні компоненти, що використовуються при роботі:

- Producer - програма, яка надсилає повідомлення;
- Queue - це ім'я поштової скриньки, що знаходиться всередині RabbitMQ. Хоча повідомлення проходять через RabbitMQ та ваші програми, вони можуть зберігатися лише у черзі. Черга обмежена лише межами пам'яті хоста та диском, по суті це великий буфер повідомлень. Багато Producers можуть надсилати повідомлення, які йдуть в одну чергу, і багато Consumers можуть намагатися отримувати дані з однієї черги;
- Consumer – програма яка чекає на отримання повідомлення з черги.

3.3 Інструменти розгортання мікросервісів

Методи розгортання мають вирішальне значення для створення надійних і стабільних мікросервісів. На відміну від монолітної програми, де ви можете оптимізувати розгортання для одного випадку використання, методи розгортання мікросервісів мають масштабуватися до кількох служб, написаних різними мовами, кожна зі своїми залежностями. Ви повинні мати можливість довіряти своєму процесу розгортання, щоб випускати нові функції і нові послуги без шкоди для загальної доступності або внесення критичних дефектів.

Оскільки програма мікросервісів розвивається на рівні підрозділів, які

можна розгортати, вартість розгортання нових служб повинна бути незначною, щоб інженери могли швидко впроваджувати інновації та надавати користь користувачам. Додаткова швидкість розробки, яку ви отримуєте від мікросервісів, буде втрачена даремно, якщо ви не зможете швидко та надійно запустити їх у виробництво. Автоматизоване розгортання має важливе значення для масштабної розробки мікросервісів.

3.3.1 Визначення контейнеризації та переваги її використання

Процес поділу частини програми від моноліту і додавання їх у окремий мікросервіс називається контейнеризацией. Це легковажна віртуалізація та ізоляція ресурсів на рівні операційної системи, яка дозволяє запускати програму та необхідний йому мінімум системних бібліотек у повністю стандартизованому контейнері.

Контейнеризація стала основною тенденцією в розробці програмного забезпечення як альтернатива або доповнення до віртуалізації. Він включає інкапсуляцію або упаковку програмного коду і всіх його залежностей, щоб він міг працювати рівномірно і узгоджено в будь-якій інфраструктурі. Технологія швидко розвивається, що дає відчутні переваги для розробників та робочих груп та для всієї інфраструктури програмного забезпечення.

Контейнеризація дозволяє розробникам створювати та розгортати програми швидше та безпечніше. За допомогою традиційних методів код розробляється в конкретному обчислювальному середовищі, яке при перенесенні на нове місце часто призводить до помилок та помилок. Наприклад, коли розробник переносить код з настільного комп'ютера на віртуальну машину або Linux в операційну систему Windows. Контейнери усувають цю проблему, поєднуючи код програми разом із пов'язаними файлами конфігурації, бібліотеками та залежностями, необхідними для його запуску. Цей єдиний пакет програмного забезпечення або «контейнер»

абстрагується від операційної системи хоста, і, отже, він сам по собі стає переносним – здатним працювати на будь-якій платформі або в хмарі без проблем.

Контейнери часто називають «легковажними», тобто спільно використовують ядро операційної системи комп'ютера і не вимагають додаткових витрат на асоціювання операційної системи в кожному додатку. Ємності по своїй суті менші, ніж у віртуальної машини, і вимагають менше часу запуску, що дозволяє набагато більшій кількості контейнерів працювати на тій самій обчислювальній потужності, що й у однієї віртуальної машини. Це підвищує ефективність роботи сервера та, у свою чергу, знижує витрати на сервер та ліцензування.

Простіше кажучи, контейнеризація дозволяє писати програми один раз і запускати будь-де. Ця мобільність важлива з погляду процесу розробки та сумісності з постачальником. Він також пропонує інші помітні переваги, такі як ізоляція відмов, простота управління та безпека, та багато інших.

Контейнери інкапсулюють додаток як єдиний пакет програмного забезпечення, що виконується, який пов'язує код програми разом з усіма відповідними файлами конфігурації, бібліотеками і залежностями, необхідними для його запуску. Контейнерні програми «ізольовані» у тому сенсі, що вони не об'єднуються в копію операційної системи. Натомість в операційну систему хоста встановлюється двигун з відкритим вихідним кодом, який стає каналом для контейнерів, щоб спільно використовувати операційну систему з іншими контейнерами у тій же обчислювальній системі.

Контейнеризація дозволяє розробникам створювати та розгортати програми швидше та безпечніше, незалежно від того, чи є програма традиційним монолітом (однорівнева програмна програма) або модульним мікросервісом (набором слабко пов'язаних служб). Нові хмарні програми можуть створюватися з нуля як контейнерні мікросервіси, розбиваючи складну програму на ряд невеликих спеціалізованих та керованих сервісів.

Існуючі програми можуть бути перепаковані в контейнери та контейнерні мікросервіси, які використовують обчислювальні ресурси більш ефективно.

Контейнеризація пропонує значні переваги для розробників та груп розробників. Розглянемо основні переваги:

перенесення. Контейнер створює пакет програмного забезпечення, який абстрагується, не прив'язаний до операційної системи хоста або не залежить від неї. Тому контейнер є переносним і здатний працювати рівномірно та узгоджено на будь-якій платформі або у хмарі.

Гнучкість. Docker Engine для запуску контейнерів розпочав промисловий стандарт для контейнерів з простими інструментами розробника та універсальним підходом до пакетування, який працює як у Linux, так і Windows. Контейнерна екосистема перейшла на механізми, керовані Ініціативою відкритих контейнерів (OCI). Розробники програмного забезпечення можуть продовжувати використовувати гнучкі інструменти або процеси DevOps для швидкої розробки та покращення програм.

Швидкість. Контейнери часто називають «полегшеними», тобто спільно використовують ядро операційної системи (ОС) комп'ютера і не перевантажені цими додатковими витратами. Це не тільки підвищує ефективність роботи серверів, а й знижує витрати на сервер та ліцензування, а також прискорює запуск, оскільки операційна система не завантажується.

Ізолювання помилок. Кожен контейнерний додаток ізольовано та працює незалежно від інших. Відмова одного контейнера не впливає на подальшу роботу будь-яких інших контейнерів. Групи розробників можуть виявляти та виправляти будь-які технічні проблеми в одному контейнері без будь-яких простоїв в інших контейнерах. Крім того, механізм контейнерів може використовувати будь-які методи ізоляції безпеки ОС, такі як управління доступом SELinux для ізоляції збоїв у контейнерах.

Ефективність. Програмне забезпечення, що працює в контейнерах, спільно використовує ядро ОС комп'ютера, а рівні програм у контейнері можуть спільно використовуватися контейнерами. Таким чином, контейнери

за своєю природою мають меншу ємність, ніж віртуальна машина, і вимагають менше часу запуску, що дозволяє набагато більшій кількості контейнерів працювати на тій самій обчислювальній потужності, що й одна віртуальна машина. Це підвищує ефективність роботи сервера, знижує витрати на сервер та ліцензування.

Простота керування. Платформа оркестрації контейнерів автоматизує встановлення, масштабування та управління контейнерними робочими навантаженнями та сервісами. Платформи оркестрування контейнерів можуть спростити завдання управління, такі як масштабування контейнерних програм, розгортання нових версій програм, а також забезпечення моніторингу, ведення журналу та налагодження, серед інших функцій. Kubernetes – найпопулярніша доступна система контейнерного оркестрування. Це технологія з відкритим вихідним кодом (спочатку з відкритим вихідним кодом від Google, заснована на їхньому внутрішньому проекті під назвою Borg), яка спочатку автоматизує функції контейнерів Linux. Kubernetes працює з багатьма механізмами контейнерів, такими як Docker, але також працює з будь-якою системою контейнерів, яка відповідає стандартам Open Container Initiative (OCI) для форматів зображень контейнерів і середовищ виконання.

Безпека. Ізоляція додатків як контейнерів за своєю суттю запобігає вторгненню шкідливого коду до інших контейнерів або хост-системи. Крім того, можуть бути визначені дозволи безпеки автоматичного блокування небажаних компонентів від входу в контейнери або обмеження зв'язку з непотрібними ресурсами.

Розробники програмного забезпечення розглядають мікросервіси як чудовий підхід до розробки додатків та управління ними в порівнянні з більш ранньою монолітною моделлю, в якій програмний додаток поєднувався з відповідним інтерфейсом користувача і базовою базою даних в одному пристрої на одній серверній платформі. За допомогою мікросервісів складний додаток розбивається на ряд дрібніших, більш спеціалізованих

сервісів, кожен зі своєю базою даних та власною бізнес-логікою. Мікросервіси потім взаємодіють один з одним через загальні інтерфейси (наприклад, API) та інтерфейси REST (наприклад, HTTP). Використовуючи мікросервіси, групи розробників можуть зосередитись на оновленні певних областей програми, не торкаючись його в цілому, що призводить до прискорення розробки, тестування та розгортання.

Концепції, що лежать в основі мікросервісів і контейнеризації, схожі, оскільки обидві вони є методами розробки програмного забезпечення, які по суті перетворюють додатки на набори невеликих сервісів або компонентів, які є переносними, масштабованими, ефективними та простими в управлінні. Більше того, мікросервіси та контейнеризація добре працюють разом. Контейнери забезпечують легку інкапсуляцію будь-якої програми, будь то традиційний моноліт або модульний мікросервіс. Мікросервіс, розроблений у контейнері, потім отримує всі властиві контейнеру переваги - мобільність з точки зору процесу розробки та сумісності з постачальником (без прив'язки до постачальника), а також гнучкість розробника, ізоляція збоїв, ефективність роботи серверів, автоматизація встановлення, масштабування та управління також рівні безпеки, серед іншого.

Контейнери, мікросервіси та хмарні обчислення працюють разом, щоб вивести розробку та доставку додатків на новий рівень, що неможливо за традиційних методологій та середовищ. Ці підходи наступного покоління додають гнучкості, ефективності, надійності та безпеки до життєвого циклу розробки програмного забезпечення – все це призводить до більш швидкої доставки додатків та удосконалень для кінцевих користувачів та ринку.

Концепція контейнеризації та ізоляції процесів давно існує, але поява в 2013 році Docker Engine з відкритим вихідним кодом, галузевого стандарту для контейнерів із простими інструментами розробника та універсальним підходом до упаковки, прискорила впровадження цієї технології. Останні дослідження прогнозує, що понад 50% компаній будуть використовувати контейнерні технології до 2020 [13].

Коли ми говоримо про контейнери, перш за все ми маємо на увазі Docker – open-source-технологію, що застосовується для розробки, тестування, доставки та запуску веб-застосунків у середовищах з підтримкою контейнеризації. Він необхідний для більш ефективного використання системи та ресурсів, швидкого розгортання готових програмних продуктів, а також для їх масштабування та перенесення в інші середовища з гарантованим збереженням стабільної роботи. Завдяки своїй популярності ця технологія стала синонімом слова «контейнер».

Запуск Docker робить всього кілька системних викликів і ядро створює для нового процесу окремий простір, окрему віртуальну мережу, окремий набір обмежень по ресурсах [14]. Процес «запущений у docker», насправді, знаходиться не в якійсь віртуальній машині (немає ніякого емулятора справжньої машини, ніякої віртуальної сутності), він запущений на тій же машині, тим же ядром.

Визначимо основні функції та властивості Docker контейнера:

- має вміст – щось знаходиться всередині контейнера або поза контейнером;
- портативний. Контейнер Docker можна використовувати на локальному комп'ютері, комп'ютері вашого колеги або серверах хмарного провайдера. Ніби та коробочка дитячих дрібничок, яку ти продовжуєш перевозити з собою з дому в будинок;
- має зрозумілі інтерфейси для доступу. Контейнер Docker має кілька механізмів взаємодії із зовнішнім світом. Він має порти, які можна відкрити для взаємодії через браузер. Ви можете настроїти його для взаємодії з даними через командний рядок;
- можна отримати з віддаленого доступу. Контейнер Docker зберігає образ, схоже на форму контейнера. Потім, коли вам потрібен контейнер, ви можете його зробити з образу. Образ містить Dockerfile з інструкціями нового образу, образ операційної системи, бібліотеки та кодову базу, необхідні для запуску вашої програми.

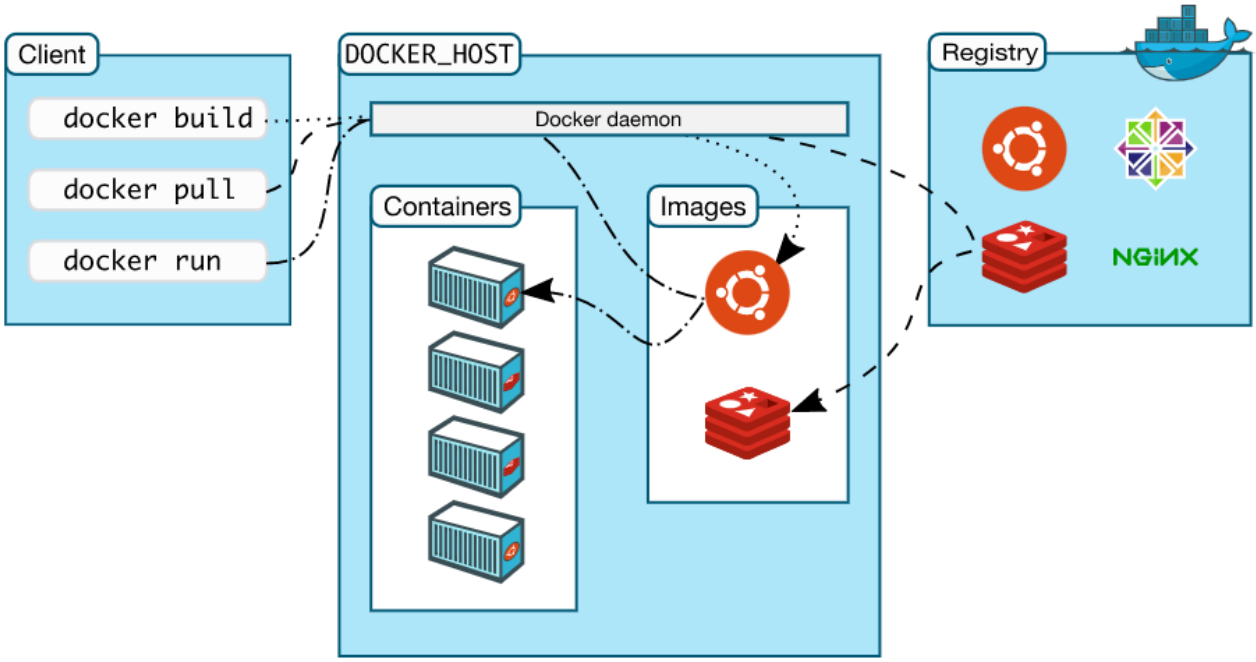

Рисунок 3.3 – Схема принципу работи Docker

Використання Docker контейнерів дозволяє здійснювати швидку та послідовну доставку потрібних програм. Docker оптимізує життєвий цикл розробки, дозволяючи розробникам працювати у стандартизованих середовищах, використовуючи локальні контейнери, які надають ваші програми та послуги. Контейнери відмінно підходять для безперервної інтеграції та робочих процесів з безперервною доставкою (CI/CD).

Розробники можуть писати код локально та ділитися своєю роботою зі своїми колегами, використовуючи контейнери Docker. Вони використовують Docker, щоб впровадити свої програми у тестове середовище та виконати автоматичні та ручні тести. Коли розробники знаходять помилки, вони можуть виправляти їх у середовищі розробки та повторно розгортати їх у тестовому середовищі для тестування та перевірки. Після завершення тестування отримати виправлення для клієнта так само просто, як надіслати оновлений образ до виробничого середовища.

Контейнерна платформа Docker забезпечує високу перенесення робочих навантажень. Контейнери Docker можуть працювати на локальному ноутбуці розробника, на фізичних чи віртуальних машинах у центрі обробки

даних, у хмарних провайдерах або у різних середовищах. Портативність і легкість Docker також спрощують динамічне управління робочими навантаженнями, масштабування або руйнування додатків і служб відповідно до потреб бізнесу практично в реальному часі.

Docker використовує архітектуру клієнт-сервер. Клієнт Docker спілкується з демоном Docker, який виконує важку роботу зі збирання, запуску та розповсюдження ваших контейнерів Docker. Docker-клієнт та демон можуть працювати в одній системі або ви можете підключити Docker-клієнт до віддаленого Docker-демона. Клієнт Docker та демон взаємодіють за допомогою REST API, через сокети UNIX або мережевий інтерфейс.

Docker-демон (Docker daemon) прослуховує запити Docker API та керує об'єктами Docker, такими як зображення, контейнери, мережі та томи. Демон також може взаємодіяти з іншими демонами для керування сервісами Docker.

Docker-клієнт – це основний спосіб взаємодії багатьох користувачів Docker з Docker. Коли використовуються такі команди, як docker run, клієнт відправляє ці команди в Docker daemon що їх виконує. Клієнт Docker може спілкуватися з декількома демонами через Docker API.

У реєстрі Docker зберігають зображення Docker. Docker Hub – це загальнодоступний реєстр, який може використовувати кожен, і Docker за замовчуванням налаштований на пошук зображень у Docker Hub. Ви можете навіть запустити свій власний реєстр. Якщо ви використовуєте Docker Datacenter (DDC), він включає Docker Trusted Registry (DTR). Коли ви використовуєте команди Docker Pull або Docker Run, потрібні зображення витягуються з налаштованого реєстру. Коли ви використовуєте команду docker push, ваш образ міститься у вашому налаштованому реєстрі.

Docker легкий та швидкий. Він забезпечує життєздатну, економічно вигідну альтернативу віртуальним машинам, тому можливо використовувати більше обчислювальних потужностей для досягнення своїх бізнес-цілей. Docker ідеально підходить для середовищ з високою щільністю та для малих та середніх розгортань, де потрібно робити більше з меншими ресурсами.

3.3.2 Визначення оркестрації контейнерів, переваги та принцип роботи

Еволюція технологій змінила спосіб побудови архітектури додатків. Docker, хмарні сервіси та сервіси оркестрації контейнерів надали нам можливість розробляти розподілені, масштабованіші та надійніші рішення.

Оркестрація контейнерів автоматизує розгортання, керування, масштабування та створення мереж контейнерів. Підприємства, яким необхідно розгорнути та керувати сотнями або тисячами контейнерів та хостів Linux, можуть скористатися перевагами оркестрування контейнерів.

Оркестрація контейнерів може використовуватись у будь-якому середовищі, де ви використовуєте контейнери. Це може допомогти вам розгорнути один і той же додаток у різних середовищах без необхідності його перепроектування. А мікросервіси в контейнерах спрощують організацію сервісів, зокрема сховищ, мереж та безпеки.

Розглянемо завдання, які можна вирішити з допомогою оркестрації контейнерів:

- постачання та розгортання;
- конфігурація та керування програмою;
- розподіл ресурсів;
- масштабування або видалення контейнерів на основі балансування робочих навантажень в інфраструктурі програми;
- балансування навантаження та маршрутизація трафіку;
- моніторинг стану контейнера;
- налаштування програм на основі контейнера, в якому вони будуть працювати;
- підтримка взаємодії між контейнерами.

Коли ви використовуєте інструмент оркестрації контейнера, такий як Kubernetes або Docker Swarm, ви зазвичай описуєте конфігурацію своєї програми у файлі YAML або JSON, залежно від інструменту оркестрування. У цих файлах конфігурації (.yml) ви вказуєте інструмент оркестровки, де

збирати образи контейнерів, як встановлювати мережу між контейнерами, як монтувати томи зберігання і де зберігати журнали для цього контейнера. Як правило, групи розгалужують і керують версіями цих файлів конфігурації, щоб вони могли розгортати одні й самі програми в різних середовищах розробки і тестування, перш ніж розгортати їх у робочих кластерах.

Контейнери розгортаються на хостах, зазвичай у групах, що реплікуються. Коли настав час розгортати новий контейнер у кластері, інструмент управління контейнером планує розгортання та шукає найбільш підходящий хост для розміщення контейнера на основі визначених обмежень (доступність ЦП або пам'яті). Є можливість розміщувати контейнери відповідно до міток або метаданих або відповідно до їх близькості по відношенню до інших хостів - можна використовувати всі види обмежень.

Коли контейнер запущено на хості, інструмент оркестрування керує своїм життєвим циклом відповідно до специфікацій, викладених у файлі визначення контейнера. Основна зручність інструментів оркестрування контейнерів у тому, що ви можете використовувати їх у будь-якому середовищі, в якому ви можете запускати контейнери. В даний час контейнери підтримуються практично в будь-якому середовищі, від традиційних локальних серверів до екземплярів загальнодоступної хмари

Існують різні платформи для оркестрації контейнерів. Вони дозволяють реалізувати зручні та ефективні засоби розгортання контейнерних систем, побудови єдиної централізованої консолі для керування. Сьогодні таких платформ дуже багато, вони все більше набирають інтересу та розвитку у середовищі розробки високонавантажених систем. Найбільш відомі такі системи: Kubernetes, Docker Swarm та Apache Mesos.

Kubernetes – opensource-система для керування контейнерними кластерами. Завдяки тому, що Google відкрила код і зробила свою систему оркестрації відкритою, вона стала найпопулярнішою. Сьогодні багато хто розглядає її як найкраще та універсальне рішення для роботи з додатками у хмарному середовищі [15].

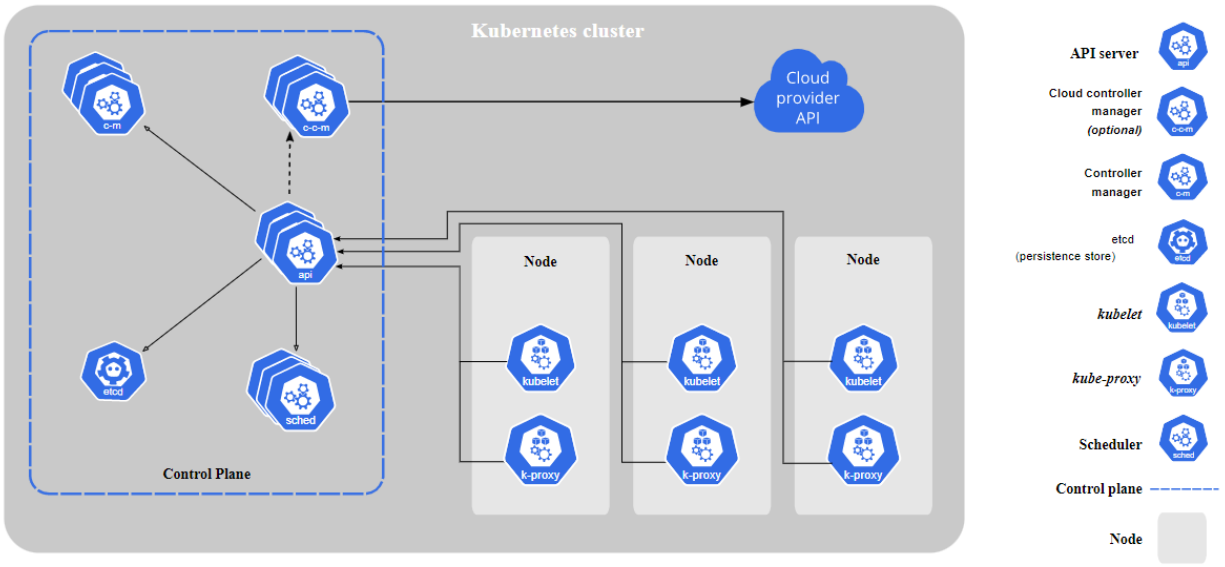

Рисунок 3.4 – Компоненти кластеру Kubernetes

Спочатку розроблений Google як відгалуження від його проекту Borg, Kubernetes зарекомендував себе як фактичний стандарт для оркестрування контейнерів. Це флагманський проект Cloud Native Computing Foundation, який підтримується такими ключовими гравцями як Google, Amazon Web Services (AWS), Microsoft, IBM, Intel, Cisco та RedHat. Розглянемо основні компоненти архітектури Kubernetes нижче.

Cluster або кластер – це набір вузлів з принаймні одним головним вузлом та декількома робочими вузлами (іноді званими міньйонами), які можуть бути віртуальними або фізичними машинами.

Kubernetes master. Майстер управляє плануванням та розгортанням екземплярів додатків на вузлах, а повний набір сервісів, які запускає майстер-вузол, називається площиною управління. Майстер зв'язується із вузлами через сервер API Kubernetes [16]. Планувальник призначає вузли модулів (одному або декільком контейнерам) залежно від обмежень ресурсу та політики, які ви визначили.

Kubelet. Кожен вузол Kubernetes запускає процес агента, званий kubelet, який відповідає за управління станом вузла: запуск, зупинка та обслуговування контейнерів додатків на основі інструкцій із площини

управління. Kublet отримує всю інформацію від сервера API Kubernetes.

Pods. Базовий блок планування, який складається з одного або кількох контейнерів, які гарантовано розміщуються на хост-машині та можуть спільно використовувати ресурси. Кожному модулю призначається унікальна IP-адреса в кластері, що дозволяє застосуванню використовувати порти без конфліктів. Ви визначаєте бажаний стан контейнерів у модулі через об'єкт YAML або JSON, який називається PodSpec. Ці об'єкти передаються в кубелі через сервер API

Розгортання, репліки та набори реплік. Розгортання – це об'єкт YAML, який визначає модулі та кількість екземплярів контейнера, які називаються репліками, для кожного модуля. Ви визначаєте кількість реплік, які ви хочете запустити в кластері за допомогою ReplicaSet, який є частиною об'єкта розгортання. Так, наприклад, якщо вузол, на якому запущено модуль, помирає, набір реплік гарантуватиме, що інший модуль запланований на іншому Docker Swarm - друга за популярністю система оркестрації.

Інструмент контейнерної кластеризації Docker Swarm з'явився трохи згодом і став частиною платформи Docker. Він дозволяє об'єднувати Dockerхости в загальний віртуальний хост. Docker Swarm цікава насамперед малим та середнім підприємствам.

Незважаючи на те, що Docker повністю прийняв Kubernetes як кращий механізм управління контейнером, компанія, як і раніше, пропонує Swarm, свій власний повністю інтегрований інструмент управління контейнером. Трохи менш розширюваний і складний, ніж Kubernetes, це гарний вибір для Docker ентузіастів, які хочуть більш простий і швидкий шлях до розгортання контейнерів. Фактично, Docker об'єднує Swarm і Kubernetes у своїй корпоративній версії, сподіваючись зробити їх додатковими інструментами.

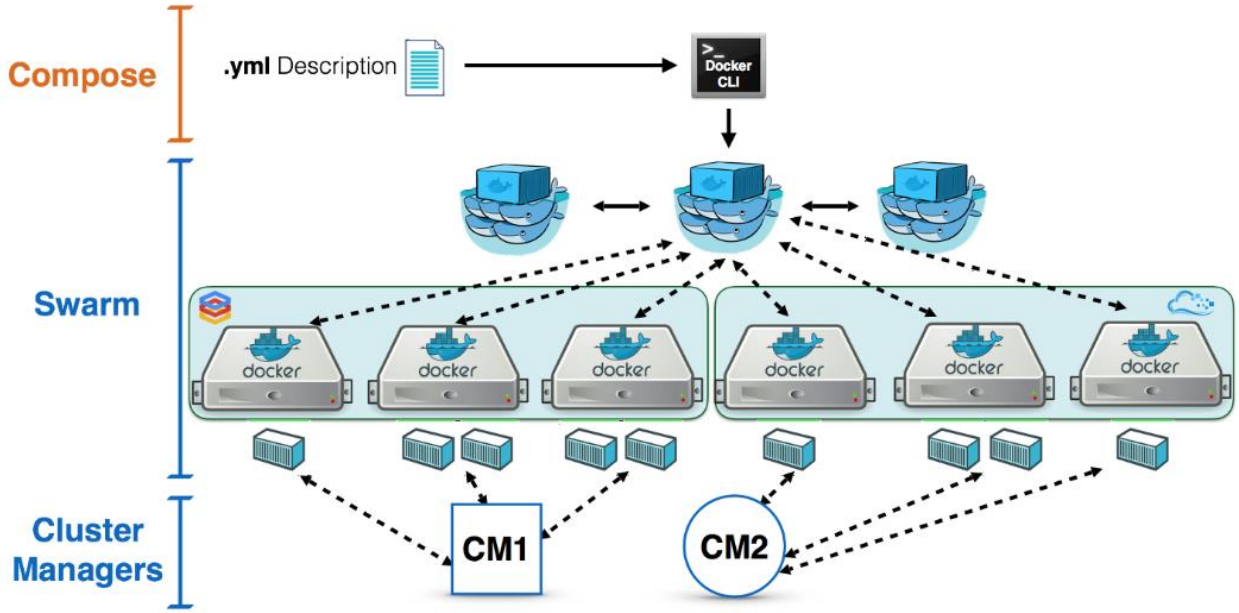

Рисунок 3.5 – Компоненти кластеру Docker Swarm

Розглянемо основні компоненти архітектури Docker Swarm нижче.

Swarm, як і кластер в Kubernetes, – це набір вузлів, принаймні, з одним головним вузлом та декількома робочими вузлами, які можуть бути віртуальними чи фізичними машинами [17].

Service – це завдання, які вузли менеджера або агента повинні виконувати в рої, як це визначено адміністратором рою. Служба визначає, які образи контейнерів повинен використовувати рій та які команди рій запускатиме в кожному контейнері. Служба у цьому контексті аналогічна мікросервісу; наприклад, саме тут ви визначаєте параметри конфігурації для веб-сервера Nginx, що працює у вашому рої. Ви також визначаєте параметри реплік у визначенні сервісу.

Менеджер вузла. При розгортанні програми у рої вузол менеджера надає кілька функцій: він доставляє роботу на робочі вузли, і навіть управляє станом роя, якого він належить. Вузол менеджера може запускати ті ж самі робочі вузли служб, але ви також можете налаштувати їх для запуску лише служб, пов'язаних з вузлом менеджера.

Робочі вузли. Ці вузли запускають завдання, розподілені вузлом менеджера у рої. На кожному робочому вузлі запускається агент, який

повідомляє головному вузлу про стан призначених йому завдань, тому вузол менеджера може відслідковувати служби та завдання, що виконуються у swarm.

Task – це контейнери Docker, які виконують команди, визначені вами у службі. Вузли диспетчера призначають завдання робочим вузлам, і після цього завдання не може бути переміщена іншому робітнику. Якщо завдання не виконується у наборі реплік, менеджер призначить нову версію цього завдання іншому доступному вузлу в рої.

На третьому місці за популярністю система Apache Mesos. Вона об'єднує існуючі об'єкти в єдиний віртуальний ресурс, формуючи великі кластери та ефективну систему управління серверною інфраструктурою, у якій кожному кластеру виділяється свій індивідуальний пул ресурсів.

Apache Mesos, трохи старший, ніж Kubernetes, – проект програмного забезпечення з відкритим вихідним кодом, спочатку розроблений у Каліфорнійському університеті в Берклі, але в даний час широко використовується в таких організаціях, як Twitter, Uber та Paypal. Полегшений інтерфейс Mesos дозволяє легко масштабувати до 10 000 вузлів (або більше) і дозволяє незалежно середовищам, що працюють на його основі, розвиватися.

Розглянемо основні компоненти архітектури Apache Mesos нижче.

Master daemon. Частина головного вузла, що керує демонами агентів. За допомогою Apache Zookeeper ви можете створити головний кворум Mesos, що складається щонайменше з трьох основних вузлів для цілей високої доступності [18].

Agent daemon. Інша частина головного вузла, яка виконує завдання, надіслані платформою.

Framework. Mesos не запускає робочі навантаження для оркестрування програм; натомість Marathon отримує ресурси від майстра Mesos (у формі пропозицій), а Marathon надсилає завдання на основі пропозицій ресурсів виконавцям, які запускають завдання на агентах.

Offer. Майстер Mesos збирає інформацію про ЦП та доступність пам'яті вузлів агента і надсилає цю інформацію до Marathon, щоб Marathon знав, які ресурси доступні.

Task. Це основні одиниці роботи, які Marathon планує з урахуванням пропозицій ресурсів від майстра Mesos. Завдання виконуються виконавцями на агентських вузлах.

У підсумок, оркестрація контейнерів є дуже важливим і корисним інструментом, який дозволяє створювати інформаційні системи з безлічі контейнерів, кожен з яких відповідає лише за певне завдання, а спілкування здійснюється через мережні порти та загальні каталоги. При необхідності кожен такий контейнер можна замінити іншим, що дозволяє, наприклад, швидко перейти на іншу версію бази даних за потреби.

3.4 Дослідження пропускної спроможності шин повідомлень

Враховуючі дані та порівняльні аналізи предоставлені у 3.1 розділі, вже відомо що Kafka — це платформа з відкритим вихідним кодом, що розповсюджується для потокової передачі подій. За своєю суттю Kafka розроблена як реплікований, розподілений, постійний журнал передачі, який використовується для живлення мікросервісів, керованих подіями, або великомасштабних програм обробки потоків [19]. Клієнти створюють або споживають події безпосередньо в/з проміжного кластера, який постійно зчитує/записує події в базову файлову систему, а також автоматично реплікує події синхронно або асинхронно в кластері для стійкості до помилок і високої доступності. RabbitMQ — прямий конкурент Kafka і є традиційним проміжним програмним забезпеченням для обміну повідомленнями з відкритим вихідним кодом, яке реалізує стандарт обміну повідомленнями AMQP, задовольняючи випадки використання черги. RabbitMQ складається з набору брокерських процесів, які розміщують «обмін» для розміщення

повідомлень і черг для отримання повідомлень. Доступність і довговічність є властивостями різних пропонованих типів очікування. Класичні ряди пропонують найменші гарантії доступності. Класичні дзеркальні черги реплікують повідомлення іншим брокерам і покращують доступність. Більш міцна стійкість забезпечується завдяки нещодавно введеним рядкам кворуму, але ціною продуктивності.

Існує багато способів порівняння систем у цьому просторі, але одна річ, про яку всіх хвилює, — це продуктивність. Відомо, що Кафка швидкий, але наскільки він швидкий сьогодні і як він поєднується з іншими системами? Ми вирішили перевірити продуктивність Kafka на новітньому хмарному обладнанні, щоб дізнатися.

Для порівняння був обраний традиційний агент обміну повідомленнями RabbitMQ та один із агентів обміну повідомленнями на основі Apache BookKeeper, Apache Pulsar. Ми зосередилися на критерію пропускній здатності системи та критерію системній затримці, оскільки це основні показники продуктивності для систем передачі виробничих подій. Зокрема, тест пропускної здатності визначає, наскільки ефективно кожна система використовує апаратне забезпечення, зокрема диски та центральні процесори. Тест затримки вимірює, наскільки близька кожна система до доставки повідомлень у режимі реального часу, включаючи затримку до p99 процентилю [20], це оцінка де 99% запитів мають бути швидшими за задану затримку або іншими словами, лише 1% запитів дозволено виконувати повільніше. Ця оцінка є ключовою вимогою для веб-додатків та архітектур мікросервісів.

Затримки RabbitMQ значно погіршуються при пропускній здатності більше 30 МБ/с. Крім того, вплив дзеркального відображення (реплікація вмісту черги) є значним при вищій пропускній здатності, а кращі затримки можна досягти, використовуючи лише класичну чергу без дзеркального відображення [21].

Таблиця 3.3 – Результати дослідження

	Kafka	RabbitMQ
Пропускна здатність	605 МБ/с	38 МБ/с
P99 затримка	5 мс	1 мс

Враховуючи постійно зростаючу популярність потокової обробки та архітектури, керованої подіями, ще одним ключовим аспектом систем обміну повідомленнями є наскрізна затримка для того, щоб повідомлення проходило по конвеєру від виробника через систему до споживача. Ми розробили експеримент, щоб порівняти це на всіх трьох системах із найвищою стабільною пропускною здатністю, яку кожна система могла витримати, не виявляючи жодних ознак надмірного використання.

Щоб оптимізувати затримку, ми змінили конфігурацію виробника в усіх системах на пакетні повідомлення лише до 1 мс (у порівнянні з 10 мс, які ми використовували для тестів пропускної здатності), а також залишили кожну систему у рекомендованій конфігурації за замовчуванням, забезпечуючи високу доступність. Kafka був налаштований на використання параметрів fsync за замовчуванням (тобто fsync вимкнено), а RabbitMQ був налаштований так, щоб не зберігати повідомлення під час дзеркального відображення черг. Ми помітили, що при пропускній здатності вище 30 тис. повідомлень/с RabbitMQ зіткнеться з вузькими місцями ЦП.

Було виявлено що Kafka забезпечує найкращу пропускну здатність, забезпечуючи найнижчі наскрізні затримки майже 100%, точніше p99.9 процентиль, як зазначено на графіку на рисунку 3.6. При меншій пропускній здатності RabbitMQ доставляє повідомлення з дуже малими затримками. Kafka забезпечує найвищу пропускну здатність з усіх систем, записуючи в 15 разів швидше, ніж RabbitMQ.

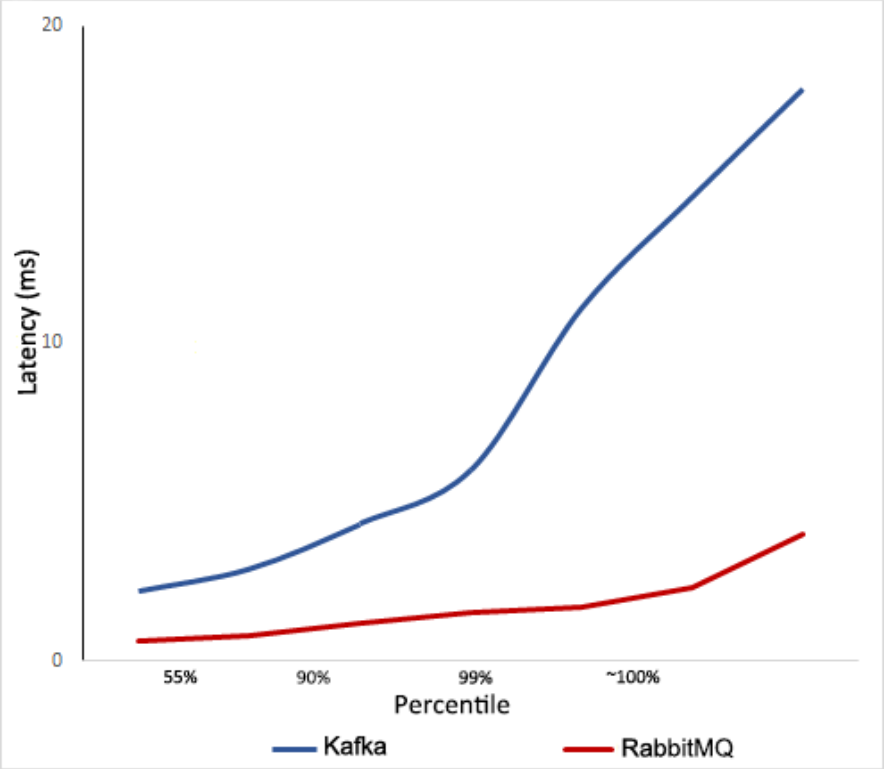

Рисунок 3.6 – Графік зміни пропускної здатності на персентиль Kafka та RabbitMQ

Kafka забезпечує найнижчу затримку при вищій пропускній здатності, а також забезпечує міцну довговічність і високу доступність. RabbitMQ може досягти меншої наскрізної затримки, ніж Kafka, але лише зі значно нижчою пропускною здатністю.

ВИСНОВКИ

У даній атестаційній роботі були розглянуті принципи роботи та структура web-додатків. Було розглянуто основні протоколи, рівні мережевої взаємодії та інші механізми, які задіяні у процесі роботи веб-програми. Було сформовано уявлення про функціонування протоколу HTTP. Розглянуто мережеві послуги DNS, NAT, DHCP.

Було вивчено поняття масштабованої розподіленої веб архітектури, її типи, принципи проектування та інфраструктурні вимоги.

У ході роботи були проведені такі роботи:

- порівняльний аналіз різних типів архітектури – моноліт, SOA та мікросервісну;
- порівняльний аналіз монолітної архітектури та мікросервісів з урахуванням основних критеріїв проектування системи;
- спроектована математичну модель розподіленої мікросервісної архітектури веб-додатка.

Були досліджені концепції побудови мікросервісів - контейнеризація, черга повідомлень та система оркестрації та відповідні інструменти. У ході роботи були проведені такі роботи:

- порівняльний аналіз черги повідомлень та REST API за основними критеріями оцінки ПЗ [22];
- порівняльний аналіз брокерів повідомлень Kafka і RabbitMQ за критеріями проектування системи
- дослідження пропускної спроможності брокерів повідомлень Kafka і RabbitMQ;
- аналіз систем оркестрації контейнерів Kubernetes, Docker Swarm та Apache Mesos.

ПЕРЕЛІК ДЖЕРЕЛ ПОСИЛАННЯ

1. Monolithic VS Microservices: Which App Development Architecture To Use [Електронний ресурс] / Режим доступу: https://superadmins.com/monolithic-vs-microservices-app-development-architecture/, Дата доступу: 23.03.2022

2. Аналіз веб-додатків на основі мікросервісної архітектури [Текст] / Мартовицький В. О., Осіпова Д. Ю. 2022 – 2с.

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

4. An introduction to Serverless Applications [Електронний ресурс] / Режим доступу: https://www.sqlshack.com/an-introduction-to-serverless-applications/, Дата доступу: 23.03.2022.

5. Madhuka Udantha Microservice Architecture and Design Patterns for Microservices [Електронний ресурс] / Режим доступу: https://medium.com/@madhukaudantha/microservice-architecture-and-designpatterns-for-microservices-e0e5013fd58a, Дата доступу: 24.03.2022

6. Service-Oriented Architecture (SOA) and Web Services: The Road to Enterprise Application Integration (EAI) [Електронний ресурс] / Режим доступу: https://www.oracle.com/technical-resources/articles/javase/soa.html, Дата доступу: 01.04.2022.

7. «From monolithic to microservices: an experience report from the banking domain» [Текст] / A. Bucchiarone, N. Dragoni, S. Dustdar, S. T. Larsen, and M. Mazzara IEEE Software, 2018 – vol. 35, no. 3, pp. 50–55.

8. Chris Richardson. From Design to Deployment [Текст] / Chris Richardson, Floyd Smith, 2016. – 74 p.

9. Pattern: Messaging [Електронний ресурс] / Режим доступу: https://microservices.io/patterns/communication-style/messaging.html, Дата

доступу: 20.04.2022

10. RESTful Web Services [Текст] / Leonard Richardson, Sam Ruby Released May O'Reilly Media, Inc., 2007 – 30 p.

11. What is RabbitMQ [Електронний ресурс] / Режим доступу: https://www.cloudamqp.com/blog/2015-05-18-part1-rabbitmq-forbeginners-what-is-rabbitmq.html, Дата доступу: 20.04.2022.

12. Optimizing Kafka Performance [Електронний ресурс] / Режим доступу: https://granulate.io/optimizing-kafka-performance/, Дата доступу: 07.05.2022.

13. Docker Wikipedia [Електронний ресурс] / Режим доступу: https://uk.wikipedia.org/wiki/Docker, Дата доступу: 10.04.2022.

14. Docker, Run swarm and kubernetes interchangeably. your choice of swarm or kubernetes for flexible and powerful orchestration options [Електронний ресурс] / Режим доступу: https://www.docker.com/products/orchestration, Дата доступу: 05.05.2022

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

16. Connecting Applications with Services [Електронний ресурс] / Режим доступу: https://kubernetes.io/docs/concepts/services-networking/connect-applications-service, Дата доступу: 05.05.2022

17. Swarm mode overview [Електронний ресурс] / Режим доступу: https://docs.docker.com/engine/swarm/, Дата доступу: 05.05.2022

18. A Platform for Fine-Grained Resource Sharing in the Data Center [Текст] / Benjamin Hindman, Andy Konwinski, Matei Zaharia, Ali Ghodsi, Anthony D. Joseph, Randy Katz, Scott Shenker, Ion Stoica., 2020 – 5 p.

19. Performance Analysis of Microservices Design Patterns. IEEE Internet Comput [Текст] / Akbulut, A.; Perros, 2019 – 19-27 pp.

20. Як обчислюється процентиль [Електронний ресурс] / Режим доступу: https://uk.economy-pedia.com/11039689-percentile, Дата доступу:

07.05.2022

21. Kubernetes Documentation learning environment [Електронний ресурс] / Режим доступу: https://kubernetes.io/docs/setup/, Дата доступу: 12.05.2022

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