Міністерство освіти і науки України
Харківський національний університет радіоелектроніки

Факультет комп'ютерних наук
(повна назва)

Кафедра програмної інженерії
(повна назва)

КВАЛІФІКАЦІЙНА РОБОТА
Пояснювальна записка

рівень вищої освіти другий (магістерський)

Дослідження методів та інструментів забезпечення безпеки в мікросервісних архітектурах з використанням контейнеризації та оркестрації
(тема)

Виконав:
студент (ка) 2 курсу, групи ІПЗм-22-5

Коломойцев П.А.
(прізвище, ініціали)

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

Керівник д.т.н. проф. Єрохін А.Л.
(посада, прізвище, ініціали)

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

2024 р.

Харківський національний університет радіоелектроніки

Факультет комп'ютерних наук
Кафедра програмної інженерії
Рівень вищої освіти другий (магістерський)
Спеціальність 121 – Інженерія програмного забезпечення
Тип програми освітньо-наукова програма
Освітня програма Інженерія програмного забезпечення
(шифр і назва)

ЗАТВЕРДЖУЮ:
Зав. кафедри _________
(підпис)
«____» _________ 2024 р.

ЗАВДАННЯ
НА КВАЛІФІКАЦІЙНУ РОБОТУ

студентові Коломойцеву Павлу Андрійовичу
(прізвище, ім'я, по батькові)

1. Тема роботи «Дослідження методів та інструментів забезпечення безпеки в мікросервісних архітектурах з використанням контейнеризації та оркестрації»

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

2. Термін подання студентом роботи до екзаменаційної комісії 25.06.2024

3. Вихідні дані до роботи Перелік методів забезпечення безпеки, програмні системи для адресування даних, техніки контейнеризації, техніки оркестрації, техніки забезпечення безпеки у контейнеризованих середовищах, Docker, Kubernetes, RabbitMQ, Kafka

4. Перелік питань, що потрібно опрацювати в роботі: сучасний стан мікросервісної архітектури, контейнеризація та оркестрація, безпека в мікросервісних архітектурах, методи забезпечення безпеки при використанні контейнерів, забезпечення безпеки в оркестраційних , аналіз технологічних рішень для створення мікросекрвісних архітектур

КАЛЕНДАРНИЙ ПЛАН

Номер	Назва етапів роботи	Термін виконання етапів роботи	Примітка
1	Видача завдання	30.03.2024	виконано
2	Аналіз предметної галузі	01.04.2024	виконано
3	Постановка задачі	01.04.2024	виконано
4	Експериментальні дослідження	02.04 – 20.04.24	*виконано*
5	Аналіз результатів експериментальних досліджень та розробка рекомендацій	20.04 – 23.04.24	*виконано*
6	Написання та оформлення статті та тез доповіді	17.04 – 26.04.24	*виконано*
7	Підготовка пояснювальної записки	01.04 – 26.04.24	*виконано*
8	Підготовка презентації та доповіді	26.04 – 2.05.24	*виконано*
9	Нормоконтроль	14.05 – 17.06.24	*виконано*
10	Рецензування	18.05 – 20.06.24	*виконано*
11	Занесення диплома в електронний архів	22.06.2024	*виконано*
12	Попередній захист	22.06.202	*виконано*
13	Допуск до захисту у зав. кафедри	23.06.2024	*виконано*

Дата видачі завдання «30» березня 2024 р.

Студент ____________________________ Коломойцев П.А.

(підпис)

Керівник роботи____________________ д.т.н. проф. Єрохін А. Л.

(підпис) (посада, прізвище, ініціали)

РЕФЕРАТ / ABSTRACT

Пояснювальна записка містить 82 ст., 13 рис., 3 табл., 21 джерел.

БЕЗПЕКА ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ, КОНТЕЙНЕР, МЕРЕЖЕВА БЕЗПЕКА, МІКРОСЕРВІСНА АРХІТЕКТУРА, ОРКЕСТРАЦІЙНІ СИСТЕМИ, DOCKER, KUBERNETES, RBAC, TLS.

Об'єктом дослідження є методи та інструменти забезпечення безпеки в мікросервісних архітектурах, які використовують контейнеризацію та оркестрацію.

Метою роботи є ретельний аналіз та дослідження різноманітних методів та інструментів, які використовуються для забезпечення безпеки в мікросервісних архітектурах, з особливим акцентом на використанні контейнеризації та оркестрації. Робота спрямована на вивчення сучасних підходів до забезпечення безпеки програмного забезпечення у розподілених системах, а також оцінку ефективності використаних інструментів у конкретному контексті мікросервісної архітектури.

В результаті проведеного дослідження буде отримана глибока інсайтова інформація щодо методів та інструментів забезпечення безпеки в мікросервісних архітектурах, зокрема при використанні контейнеризації та оркестрації. Робота визначить ключові аспекти безпеки програмного забезпечення в контексті мікросервісів і розгляне практичні рекомендації для ефективного застосування цих методів у реальних проектах.

CONTAINER, DOCKER, KUBERNETES, MICROSERVICE ARCHITECTURE, NETWORK SECURITY, ORCHESTRATION SYSTEMS, RBAC, SOFTWARE SECURITY, TLS.

The object of research are methods and tools for ensuring security in microservice architectures that use containerization and orchestration.

The aim of the work is to thoroughly analyze and explore the various methods and tools used to ensure security in microservice architectures, with a special emphasis on the use of containerization and orchestration. The work is aimed at studying modern approaches to ensuring software security in distributed systems, as well as evaluating the effectiveness of the tools used in the specific context of microservice architecture.

As a result of the conducted research, deep insight information will be obtained regarding methods and tools for ensuring security in microservice architectures, in particular when using containerization and orchestration. The work will identify the key aspects of software security in the context of microservices and will consider practical recommendations for the effective application of these methods in real projects.

Я, Коломойцев Павло Андрійович, студент(ка) гр. ІПЗм-22-5, здобувач вищої освіти на другому (магістерському) рівні кафедри «Програмна інженерія», заявляю: моя кваліфікаційна робота на тему «Дослідження методів та інструментів забезпечення безпеки в мікросервісних архітектурах з використанням контейнерізації та оркестрації», що буде представлена в екзаменаційну комісію для публічного захисту, виконана самостійно, в ній не містяться елементи плагіату і вона може бути опублікована в електронному архіві відкритого доступу ElArKhNURE. Всі запозичення з друкованих та електронних джерел мають відповідні посилання.

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

ПЕРЕЛІК СКОРОЧЕНЬ

API – Application Programming Interface;

OPA– Open Policy Agent;

REST– Representational State Transfer;

HTTP – HyperText Transfer Protocol;

gRPC – gRPC Remote Procedure Calls ;

DevSecOps – Development, Security, and Operations;

RBAC– Role-Based Access Control;

TLS – Transport Layer Security;

mTLS – Mutual Transport Layer Security;

JSON – JavaScript Object Notation;

IAM – Identity and Access Management;

MFA – Multi-Factor Authentication;

DDoS – Distributed Denial of Service;

SAML – Security Assertion Markup Language;

OAuth2 – Open Authorization;

DDC – Docker Datacenter;

DTR – Docker Trusted Registry;

STOMP – Simple (or Streaming) Text Oriented Messaging Protocol;

MQTT – Message Queuing Telemetry Transport.

ЗМІСТ

Вступ 10
1 Аналіз предметної галузі 11
  1.1 Визначення мікросервісної архітектури 11
  1.2 Переваги та недоліки використання мікросервісів 14
  1.3 Контейнеризація в розподілених системах 15
    1.3.1 Шаблони управління одним контейнером 15
    1.3.2 Шаблони одновузлових та багатоконтейнерних додатків 16
    1.3.3 Шаблони багатовузлових додатків 19
  1.4 Оркестрація мікросервісів 21
  1.5 Постановка задач дослідження 26
2 Методи вирішення проблеми 28
  2.1 Безпека в мікросервісних архітектурах 28
  2.2 Методи забезпечення безпеки при використанні контейнерів 31
  2.3 Засоби забезпечення безпеки в оркестраційних системах 33
3 Аналіз технологічних рішень та інструментів для створення мікросервісних структур 35
  3.1 Мікросервісні комунікації REST API та черги повідомлень 36
  3.2 Фреймворки передачі даних через використання черги повідомлень 42
  3.3 Організація мікросервісів за допомогою інструментів для розгортання 49
    3.3.1 Ідея та переваги використання контейнеризації 50
    3.3.2 Опис використання контейнерів, їх переваги та основні принципи функціонування 57
  3.4 Вивчення пропускної здатності шин передачі даних 63
Висновки 66
Перелік джерел посилання 67
Перелік джерел посилання за науковими напрямами керівника та науковців кафедри програмної інженерії 69
Додадок А 70
Додаток Б 71

Додаток В 74
Додаток Г... 82

ПЕРЕЛІК СКОРОЧЕНЬ

API – Application Programming Interface;

OPA– Open Policy Agent;

REST– Representational State Transfer;

HTTP – HyperText Transfer Protocol;

gRPC – gRPC Remote Procedure Calls ;

DevSecOps – Development, Security, and Operations;

RBAC– Role-Based Access Control;

TLS – Transport Layer Security;

mTLS – Mutual Transport Layer Security;

JSON – JavaScript Object Notation;

IAM – Identity and Access Management;

MFA – Multi-Factor Authentication;

DDoS – Distributed Denial of Service;

SAML – Security Assertion Markup Language;

OAuth2 – Open Authorization;

DDC – Docker Datacenter;

DTR – Docker Trusted Registry;

STOMP – Simple (or Streaming) Text Oriented Messaging Protocol;

MQTT – Message Queuing Telemetry Transport.

ВСТУП

У сучасному цифровому ландшафті, де динамічність та масштабованість визначають рух розробки програмного забезпечення, мікросервісна архітектура виступає ключовим елементом для досягнення гнучкості та ефективності в розробці та управлінні розподіленими системами. За останні роки виникла нова ера в області розгортання та управління мікросервісами завдяки технологіям контейнеризації та оркестрації, таким як Docker та Kubernetes.

Проте, разом зі зростанням популярності цих технологій, виникають нові виклики та загрози в сфері безпеки інформаційних систем. Забезпечення безпеки в мікросервісних архітектурах, які базуються на контейнеризації та оркестрації, стає важливим завданням для індустрії та підприємств, оскільки це визначає успішність та стабільність їхніх інформаційних систем.

Ця наукова робота присвячена глибокому дослідженню методів та інструментів забезпечення безпеки в мікросервісних архітектурах, з фокусом на використанні технологій контейнеризації та оркестрації. Ми розглянемо сучасні підходи до вирішення проблем безпеки в контексті розподілених систем, а також розглянемо ефективність та переваги застосування різних засобів безпеки. Наша мета – визначити оптимальні стратегії та рішення для забезпечення найвищого рівня безпеки в мікросервісних архітектурах, що використовують контейнеризацію та оркестрацію, у відповідь на сучасні виклики та загрози інформаційної безпеки.

1 АНАЛІЗ ПРЕДМЕТНОЇ ГАЛУЗІ

1.1 Визначення мікросервісної архітектури

Термін “Мікросервісна архітектура”, також відомий як мікросервіси, з’явився в середині 2010-х, щоб описати особливий архітектурний стиль розробки програмних додатків. Цей стиль розробки отримав розповсюдження в зв’язку з розвитком практик гнучкої розробки та DevOps. На даний момент мікросервісній архітектурі приділяється багато уваги: статті, блоги, дискусії в соціальних мережах і презентації на конференціях. Коли йдеться про забезпечення гнучкої розробки і доставки складних корпоративних додатків – такий спосіб розробки має значні переваги (див. рис. 1.1).

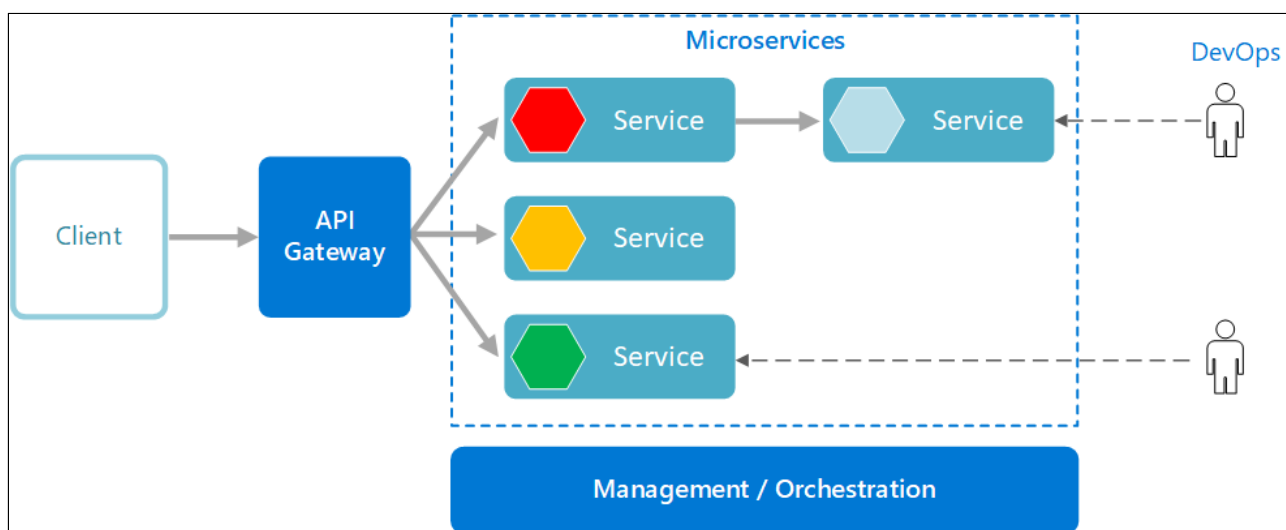

Рисунок 1.1 – Стиль архітектури мікросервісу (за даними [1])

Якщо коротко, то архітектурний стиль мікросервісів – це підхід, коли єдиний додаток будується як сукупність невеликих, самодостатніх, незалежних, не тісно зв’язаних сервісів, що спілкуються між собою за допомогою легких механізмів як то HTTP, gRPC, AMQP. Ці сервіси побудовані навколо бізнес-потреб (кожен відповідальний за конкретний процес) та розгортаються незалежно з використанням повністю автоматизованого середовища. Існує абсолютний мінімум централізованого управління цими сервісами. Самі по собі сервіси можуть бути написані на різних мовах і використовувати різні технології зберігання даних.

Одна з причин використання мікросервісів полягає в тому, що компанії хочуть мати можливість швидко щось змінювати, щоб швидше реагувати на зміни бізнес-вимог, випереджати конкурентів. Мікросервіси допомагають розробникам доставляти зміни швидше, безпечніше і з більш високою якістю, тобто зберігати швидкість розвитку продукту, навіть коли той стає неосяжних розмірів. Адже не тісно зв'язані сервіси дають можливість проводити зміни з більшою частотою ітерацій мінімізуючи вплив змін на решту частин системи.

При всьому цьому не потрібно забувати що такий підхід додає додаткову складності проекту в цілому. Нам потрібні DevOps'и для моніторингу та управління, при цьому між ними і розробниками повинні бути тісні відносини і хороша взаємодія. При роботі з мікросервісами нам доводиться більше розгортати, ускладнюється система моніторингу, сильно розростається кількість можливих збоїв. Тому в компанії дуже важлива сильна DevOps-культура.

Мікросервіси vs моноліт (див. рис. 1.2).

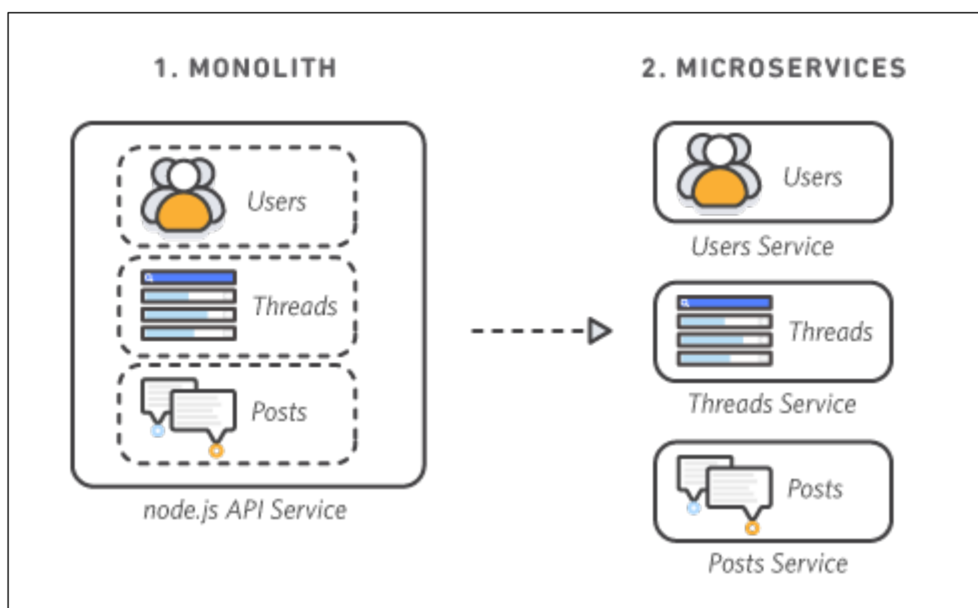

Рисунок 1.2 – Розбиття монолітної програми на мікросервіси (за даними [2])

Моноліт будуються як єдине ціле. Будь які зміни, навіть самі невеликі, потребують перебудови та розгортання всього додатку. З часом стає складніше зберігати хорошу модульну структуру, зміни логіки одного модуля мають тенденцію впливати на код інших модулів. Монолітні програми також може бути

важко масштабувати, коли різні модулі мають конфліктні вимоги до ресурсів. Наприклад, один модуль може реалізовувати логіку обробки зображень з інтенсивним використанням процесору. Інший модуль може бути вимогливим до використання оперативної пам'яті. Однак, оскільки ці модулі розгортаються разом, вам доведеться йти на компроміс з вибором апаратного забезпечення. Масштабувати доводиться всей додаток, навіть якщо це потрібно тільки для одного модуля цього додатку.

Мікросервіси на противагу – кожен мікросервіс розгортається окремо. Тож якщо ви змінюєте щось в одному з них, ви можете розгорнути ці зміни, не чіпаючи інших мікросервісів, які можуть продовжувати працювати.

Тому веб проекти все частіше розробляються як окремі сервіси що можна розгортати та масштабувати окремо.

Але мікросервісна архітектура постійно піддається критиці с самого моменту її формування, серед нових проблем, котрі виникають при її впровадженні відзначаються:

- мережеві затримки: якщо в модулях, що виконують кілька функцій, взаємодія локально, то мікросервісная архітектура накладає вимогу атомізації модулів і взаємодії їх по мережі;
- формати повідомлень: відсутність стандартизації та необхідність узгодження форматів обміну фактично для кожної пари взаємодіючих мікросервісів призводить як до потенційних помилок, так і складнощів налагодження;
- баланс навантаження і відмовостійкості;
- складність операційної підтримки – потрібні грамотні DevOps-інженери, безперервне розгортання і автоматичний моніторинг. Без всього цього мікросервіси використовувати не слід;
- тестування мікросервісів може бути громіздко. Використовуючи моноліт, нам потрібно тільки запустити додаток на сервері і переконатися в зв'язку з базою даних. А в мікросервісах, кожен окремий сервіс повинен бути запущений перед тим, як почати тестування.

Багато відомих компаній вирішили проблему моноліту, прийнявши архітектуру мікросервісів, замість того, щоб будувати єдиний монструозний моноліт. Серед них маємо Amazon, eBay, Walmart, Netflix, SoundCloud, Spotify, Twitter, Stripe, PayPal, Uber та Medium.

1.2 Переваги та недоліки використання мікросервісів

До переваг можемо віднести.

Сервіси запускається швидше, що робить розробників більш продуктивними та прискорює розгортання:

– кожен сервіс може бути розгорнутий незалежно від інших, тож якщо ви змінюєте щось в одному з них, ви можете розгорнути ці зміни, не чіпаючи інших мікросервісів, які можуть продовжувати працювати;
– кожен сервіс може бути масштабований окремо;
– баг в одному мікросервісі не підірве роботу системи, неполадки у мікросервісі не повинні зламати весь додаток, швидше за все, вони не вплинуть суттєво на роботу додатку, особливо великого;
– усуваються будь-які довгострокові зобов’язання щодо технології при розробці нового сервісу ви можете вибрати новий стек технологій, будь-який сервіс у системі можна замінити, його можна переписати з нуля в межах прийнятного часу та бюджету без необхідності перебудовувати всю систему;
– мікросервіси, як правило, краще організовані, оскільки кожен мікросервіс має дуже специфічну роботу і не займається роботою інших сервісів, тож потенційно легші для розуміння, підтримки і тестування;
– також відокремлені сервіси легше перекомпонувати і переналаштувати, щоб виконувати задачі різних додатків (наприклад, обслуговувати веб-клієнтів і публічний API).

Недоліки мікросервісів:

– розробники повинні мати справу з додатковою складністю створення розподіленої системи;

- складність розгортання – у виробництві існує також оперативна складність розгортання та управління системою, що складається з багатьох різних типів послуг;
- під час проєктування мікросервісної архітектури, ймовірно, виникне багато неочікуваних проблем, яких ви не очікували під час розробки.

1.3 Контейнеризація в розподілених системах

В кінці 1980-х та на початку 1990-х років об'єктно-орієнтоване програмування викликало революцію в розробці програмного забезпечення, популяризуючи підхід до створення додатків у вигляді наборів модульних компонентів. Сьогодні ми спостерігаємо аналогічну революцію в розробці розподілених систем зі зростанням популярності мікросервісних архітектур, побудованих на основі контейнерних програмних компонентів.

Контейнери особливо добре підходять як фундаментальний "об'єкт" у розподілених системах завдяки стінам, які вони воздвигають на межі контейнера.

Існує три типи шаблонів проектування, поява яких ми спостерігаємо в розподілених системах на основі контейнерів: шаблони з одним контейнером для управління контейнерами, шаблони з одним вузлом для тісно взаємодіючих контейнерів і шаблони з кількома вузлами для розподілених алгоритмів.

На щастя, за останні кілька років спостерігається стрімкий ріст впровадження контейнерних технологій Linux, таких як Docker і інші. Контейнер та його образ - це саме ті абстракції, які необхідні для розробки шаблонів розподілених систем. Герметично запечатані контейнери, що несуть з собою необхідні залежності та забезпечують атомарне розгортання, помітно полегшують процес розгортання програмного забезпечення в центрі обробки даних чи хмарі.

1.3.1 Шаблони управління одним контейнером

Контейнер надає природну межу для визначення інтерфейсу. Контейнери можуть викривати не лише функціональність, специфічну для додатка, але і засоби для систем управління через цей інтерфейс.

Традиційний інтерфейс управління контейнером є дуже обмеженим. Контейнер ефективно експортує три дії: run(), pause() та stop(). Хоча цей інтерфейс є корисним, багатший інтерфейс може забезпечити ще більше можливостей для розробників та операторів систем. Оскільки HTTP-сервери підтримуються майже у кожній сучасній мові програмування, існуюча підтримка для форматів даних, таких як JSON, дозволяє легко визначити API управління на основі HTTP, яке може бути "реалізоване" за допомогою встановлення в контейнері веб-сервера за конкретними точками доступу, додатково до його основної функціональності.

У "вгору" напрямку контейнер може викривати розширений набір інформації про додаток, включаючи метри моніторингу, специфічні для додатка (QPS, стан додатка і т. д.), інформацію про профілювання, яка цікавить розробників (потоки, стек, конфлікти замків, статистика мережевих повідомлень і т. д.), конфігураційну інформацію компонента та логи компонента. Наприклад, система управління контейнерами Kubernetes дозволяє користувачам визначати перевірки стану через визначені HTTP-точки доступу.

У "вниз" напрямку інтерфейс контейнера забезпечує природне місце для визначення життєвого циклу, що полегшує написання компонентів програмного забезпечення, які контролюються системою управління. Наприклад, система управління кластером зазвичай призначає "пріоритети" завданням, де завдання з високим пріоритетом гарантовано виконується навіть при перевищенні кластера. Цю гарантію забезпечує видалення вже запущених завдань з низьким пріоритетом, які потім повинні чекати, доки ресурси не стануть доступними.

1.3.2 Шаблони одновузлових та багатоконтейнерних додатків

Самим поширеним шаблоном для використання кількох контейнерів є "Шаблон Sidecar" (див. рис. 1.3). Цей шаблон розширює та удосконалює основний контейнер. Наприклад, в основі може бути веб-сервер, який може бути доповнений додатковим контейнером "logsaver", що збирає журнали веб-сервера з локального диска і передає їх до Datadog або Splunk Forwarder.

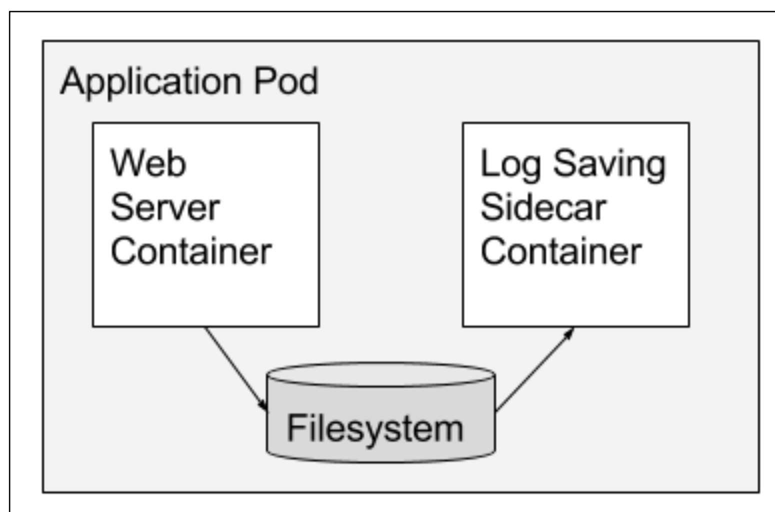

Рисунок 1.3 – Sidecar Pattern (за даними [3])

Хоча вбудовання функціональності додаткового контейнера в основний завжди є можливим, існують переваги використання окремих контейнерів. Контейнер є одиницею обліку та розподілу ресурсів, що дозволяє налаштовувати веб-сервер так, щоб він надавав стабільні відповіді з низькою затримкою на запити, які до нього надходять, тоді як "боковий" контейнер "logsaver" може використовувати вільні цикли процесора, коли веб-сервер не зайнятий. Крім того, контейнер є одиницею упаковки, що полегшує розділення відповідальностей за розробку цих окремих контейнерів між різними командами. Третій контейнер - це одиниця повторного використання, дозволяючи з'єднувати "бокові" контейнери з численними різними "основними" контейнерами. Четвертий, контейнер створює межу для стримування ситуацій збоїв, дозволяючи системі плавно деградувати (наприклад, веб-сервер може продовжувати обслуговування навіть при відмові "logsaver"). Зрештою, контейнер - це одиниця розгортання, що дає можливість оновлювати кожну частину функціональності та, за необхідності, незалежно відкочувати її.

Контейнери Ambassador виступають у ролі посередника для забезпечення зв'язку між основним контейнером та із нього (див. рис. 1.4). Наприклад, розробник може інтегрувати програму, яка взаємодіє за допомогою протоколу memcache, з амбасадором twemproxy.

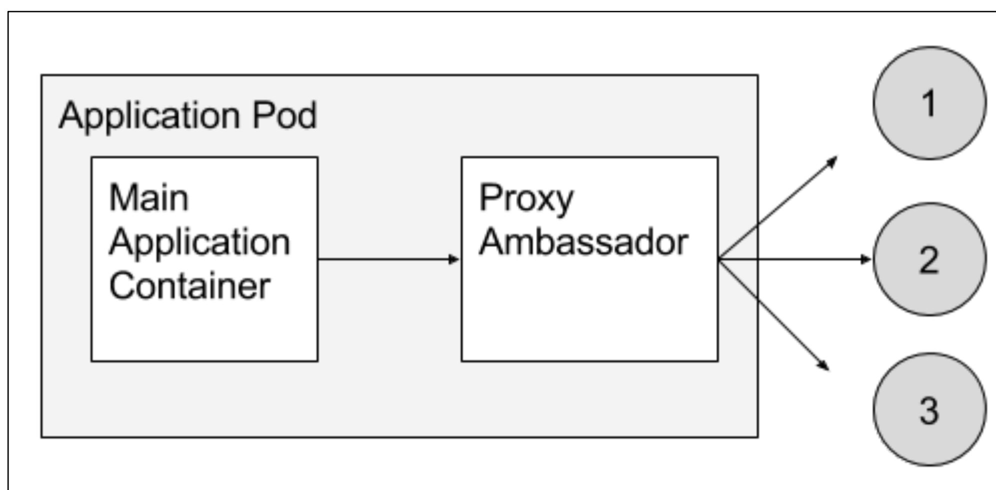

Рисунок 1.4 – Ambassador Pattern (за даними [3])

Додаток вважає, що він спілкується лише з однією кеш-пам'яттю на локальному хості, але насправді twemproxy розподіляє запити через розподілені вузли кеш-пам'яті в інших частинах кластера.

Цей шаблон контейнера спрощує роботу розробника по три способи: вони мають лише думати та програмувати з точки зору підключення своєї програми до одного сервера на локальному хості, можуть тестувати свою програму автономно, запустивши реальний екземпляр memcache на своїй локальній машині замість амбасадора, і можуть повторно використовувати twemproxy ambassador з іншими програмами, які навіть можуть бути написані на різних мовах. Амбасадори можливі завдяки тому, що контейнери на одній машині мають спільний мережевий інтерфейс локального хоста.

На відміну від шаблону Ambassador, який відображає програму зі спрощеним виглядом для зовнішнього світу, адаптери виступають у ролі інтерфейсу, що стандартизує зовнішній світ за допомогою простого та однорідного вигляду для програми. Це досягається шляхом уніфікації виводу та інтерфейсів у кількох контейнерах. Конкретним прикладом шаблону адаптера є адаптери, які забезпечують однаковий інтерфейс моніторингу для всіх контейнерів у системі (див. рис. 1.5).

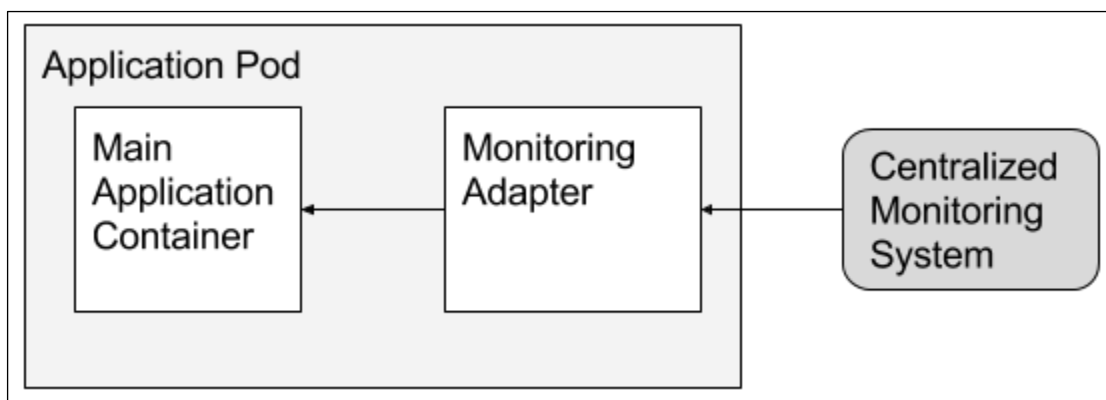

Рисунок 1.5 – Adapter Pattern (за даними [3])

Сучасні програми використовують різноманітні методи для експорту своїх показників (наприклад, JMX, statsd тощо). Однак усе стає простіше для єдиного інструменту моніторингу, коли всі програми мають узгоджений інтерфейс моніторингу. Адаптери допомагають забезпечити цю узгодженість, гарантуючи, що незалежно від того, які методи експорту використовують різні програми, їхні метрики можуть бути легко та єдинообразно зібрані, агреговані та представлені єдиною системою моніторингу.

1.3.3 Шаблони багатовузлових додатків

Переходимо від взаємодії контейнерів на одній машині до координації розподіленої архітектури з кількома вузлами.

Шаблон виборів лідера є важливим елементом у розподілених системах, де необхідно визначити лідера серед багатьох кандидатів. Реплікація, яка зазвичай використовується для розподілу навантаження між ідентичними екземплярами компонента, може бути використана і для реалізації виборів лідера, де інші репліки готові швидко замінити лідера у випадку відмови. Існує багато бібліотек, які надають механізми вибору лідера з боку програми.

Альтернативним підходом є використання контейнера для вибору лідера. Набір контейнерів для вибору лідера може спільно взаємодіяти з екземплярами програм, які потребують визначення лідера. Кожен контейнер у цьому наборі може конкурувати за роль лідера, і вони можуть надавати простий API HTTP через

локальний хост для кожного контейнера програми, який має вибрати лідера. Експерти можуть створювати контейнери вибору лідера один раз, а розробники додатків можуть легко використовувати спрощений інтерфейс незалежно від вибору мови реалізації.

В парадигмі Scatter Gather зовнішній клієнт ініціює початковий запит до "кореневого" або "батьківського" вузла в системі. Цей кореневий вузол розподіляє запит на широкий спектр серверів для паралельного виконання обчислень. Кожен шард або сервер повертає часткові дані, які потім збираються кореневим вузлом в єдину відповідь на вхідний запит (див.рис.1.6).

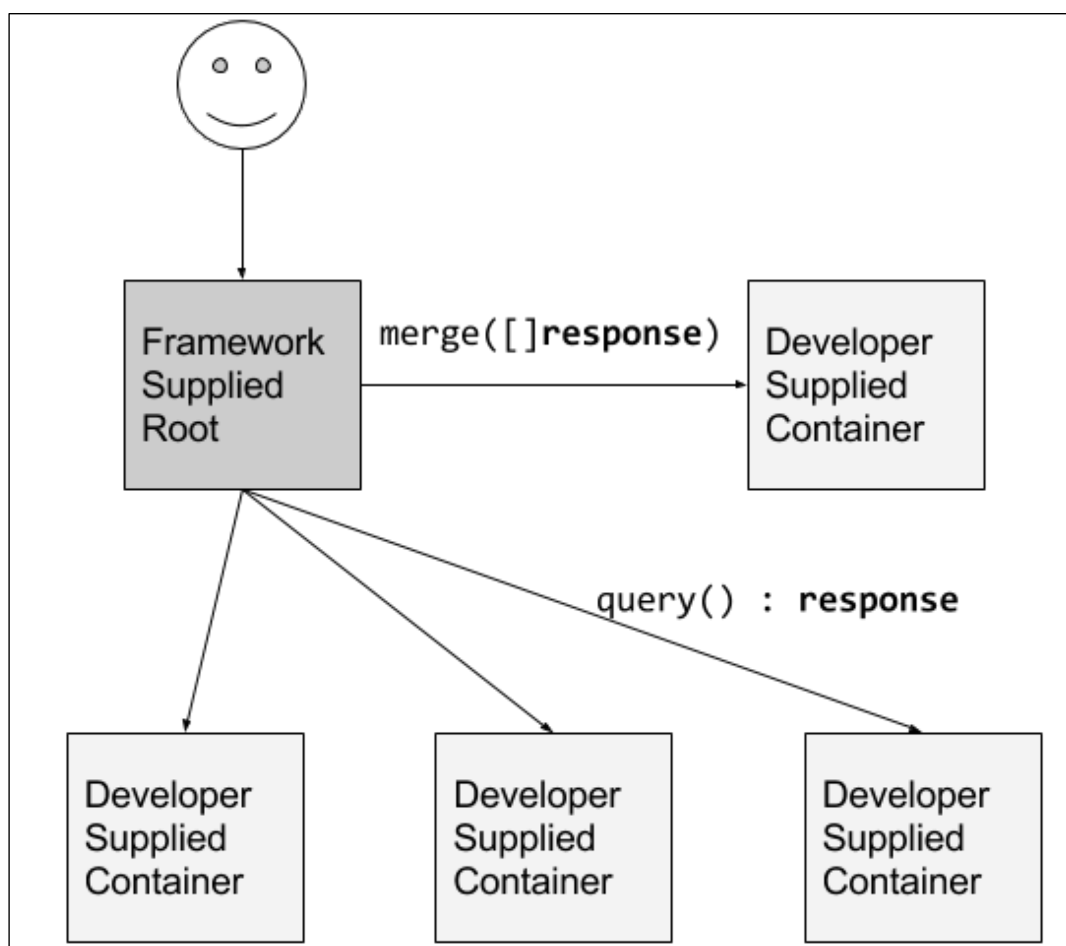

Рисунок 1.6 – Scatter Gather Pattern

Цей паттерн часто застосовується в пошукових системах. Розробка такої розподіленої системи вимагає значної кількості шаблонного коду, такого як розподіл запитів, збір відповідей, взаємодія з клієнтом та інші аспекти.

1.4 Оркестрація мікросервісів

Оркестровка мікросервісів – це процес керування та координації окремих мікросервісів для спільної роботи для надання більшої програми чи послуги в межах розподіленої архітектури. Це включає в себе керування розгортанням мікросервісів, зв'язок і масштабування, щоб забезпечити їх правильне й ефективне функціонування.

Оркестрація мікросервісів не лише забезпечує ефективний обмін даними, але також має інші переваги, такі як поліпшена масштабованість, вища стійкість до відмов, більша гнучкість, поліпшена співпраця та спрощене керування. Оркестрація дозволяє вирішити завдання викликів та взаємодії між мікросервісами, що сприяє високоякісному функціонуванню і покращує користувацький досвід.

Оркестровка мікросервісів може бути складним і складним завданням, особливо коли кількість мікросервісів і складність архітектури зростають. Хоча існує кілька інструментів і підходів для оркестровки, корисно дотримуватися певних передових практик, щоб забезпечити плавне розгортання мікросервісів. Давайте розглянемо кілька рекомендованих найкращих практик, яких слід дотримуватися:

- використовуйте контейнери для пакування та розгортання;
- впровадьте службу моніторингу та журналювання для своїх мікросервісів;
- використовуйте асинхронний зв'язок;
- відокремте сховище даних мікросервісу;
- впровадити пошук служб;
- використовуйте шлюз API для маршрутизації та автентифікації;
- використовуйте систему керування конфігурацією для узгодженості;
- розробка для неспроможності забезпечити відмовостійкість і стійкість;
- використовуйте принцип єдиної відповідальності (SRP).

Використання контейнерів для упаковування та розгортання є очевидною та настійно рекомендованою практикою. Контейнери представляють собою популярний метод для упаковування та розгортання мікросервісів, оскільки вони

забезпечують ізоляцію, портативність, високу доступність, масштабованість і покращену безпеку завдяки таким функціям, як сегментація мережі, керування секретами та ізоляція контейнерів.

Кожен мікросервіс може бути запакований у свій власний контейнер з усіма необхідними залежностями, конфігурацією та кодом. Потім їх можна розгорнути в будь-якому середовищі, яке підтримує контейнеризацію, що полегшує переміщення мікросервісів між різними середовищами, такими як розробка, постановка та виробництво.

Інструменти оркестровки контейнерів, такі як Kubernetes або Docker Swarm, оптимізують використання ресурсів, покращують продуктивність і автоматизують такі завдання, як створення, конфігурація та масштабування контейнерів, тим самим заощаджуючи час і знижуючи ризик помилок.

Впровадження служби моніторингу та журналювання для мікросервісів є важливим етапом, оскільки мікросервіси генерують велику кількість даних та подій, що може ускладнювати швидке виявлення можливих проблем.

Системи моніторингу та журналювання мікросервісів дозволяють вам відстежувати продуктивність та ефективність мікросервісів, що має вирішальне значення для підтримки стабільності та надійності вашої системи. При відстеженні продуктивності мікросервісів ви можете ідентифікувати потенційні проблеми та оптимізувати розподіл ресурсів. Також ви можете виявляти тенденції та вчасно реагувати на них, оптимізуючи масштабування, коли це необхідно.

Рекомендується використовувати такі інструменти, як Prometheus, Grafana, ELK Stack або інші агрегатори журналів та збирачі метрик, щоб забезпечити повноцінні можливості моніторингу та журналювання для ваших мікросервісів.

Для ефективного функціонування хмарної програми необхідний ефективний зв'язок між мікросервісами. Одним з найкращих підходів для досягнення цього є використання асинхронного зв'язку (див. рис. 1.7). Цей тип зв'язку дозволяє мікросервісам незалежно надсилати та отримувати інформацію, не чекаючи негайної відповіді від інших мікросервісів.

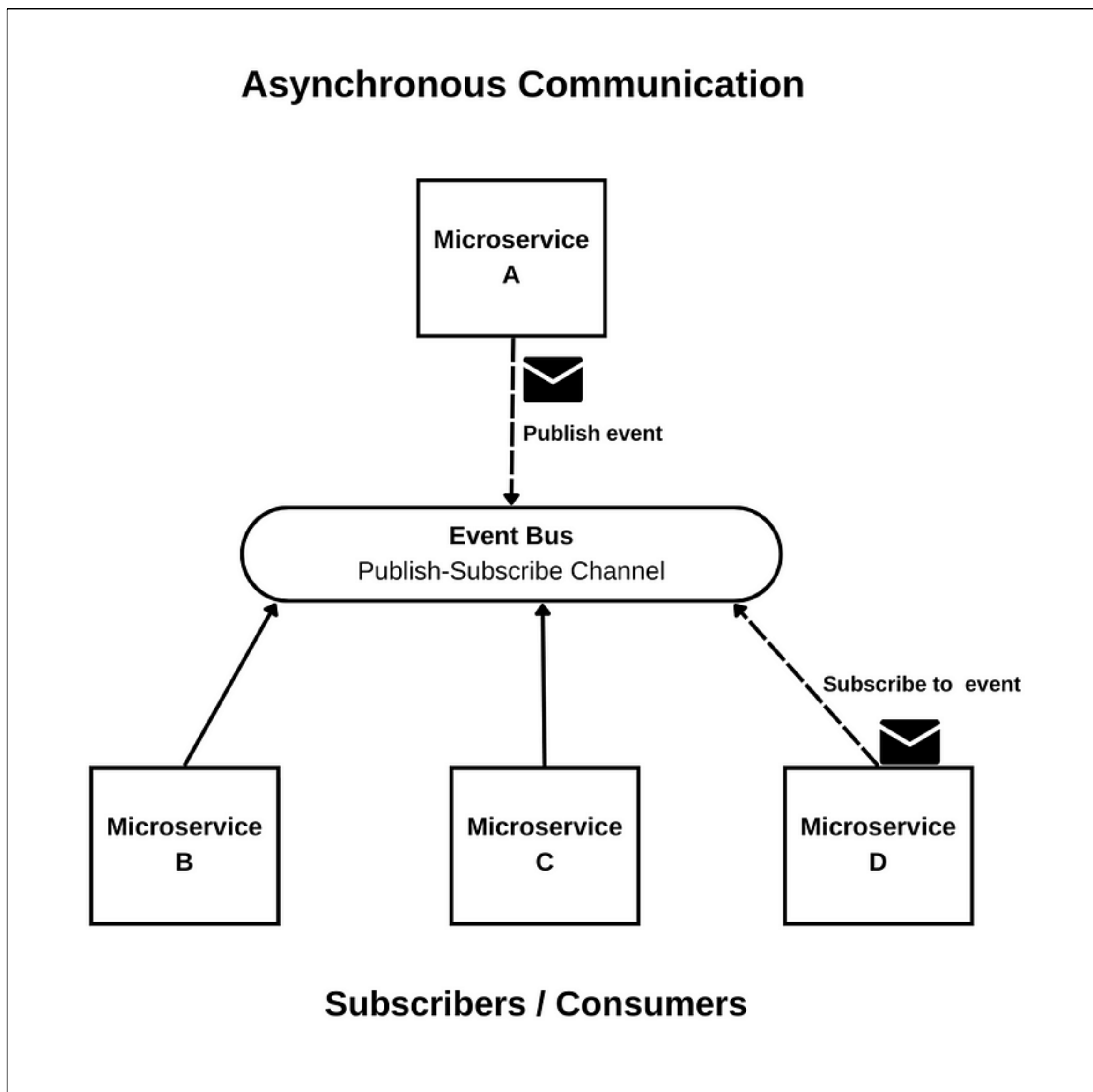

Рисунок 1.7 – Асинхронний зв’язок (за даними [4])

Шаблон публікації-підписка є одним з простих, але ефективних підходів до асинхронної комунікації. За цим підходом, коли відбувається цікава подія, виробник (мікросервіс) публікує запис цієї події в службі черги повідомлень. Інші мікросервіси, які зацікавлені в такій події, можуть підписатися на службу черги повідомлень як споживачі цієї події.

Застосовуючи асинхронний зв'язок та шаблон публікації-підписка, кожен мікросервіс може працювати незалежно та асинхронно, не чекаючи відповідей від інших мікросервісів. Це сприяє зменшенню ризику збою зв'язку, що може призвести до відмов в системі. Крім того, використання стандартизованих протоколів і форматів зв'язку є ще однією кращою практикою, оскільки це сприяє

сумісності між різними мікросервісами та спрощує інтеграцію нових мікросервісів в архітектуру.

Розділення сховища даних – це ще одна краща практика, яка може суттєво позитивно вплинути на оркестрацію мікросервісів. Цей принцип визначає, що кожен мікросервіс повинен мати власне відокремлене сховище даних і уникати використання загального сховища даних з іншими мікросервісами.

Розділяючи зберігання даних таким чином, ви отримуєте можливість легше розробляти, розгортати та управляти незалежними екземплярами управління даними для кожного мікросервісу. Це сприяє більшій гнучкості у масштабуванні окремих мікросервісів, оскільки кожен може мати свою власну базу даних. Крім того, це допомагає уникнути залежностей між сервісами та підвищує рівень безпеки, обмежуючи доступ до даних.

Впровадження системи пошуку служб стає критичним зростанням кількості мікросервісів у додатку, оскільки може виникнути складність у відстеженні активних служб та їх місцеположення. У цьому випадку інструменти виявлення служб, такі як шлюз API (наприклад, Ambassador Edge Stack), мережева сітка служб (наприклад, Consul) або реєстр служб (наприклад, Eureka), дозволяють знайти та взаємодіяти з різними службами.

Використання інструменту виявлення служб дозволяє уникнути необхідності розробки логіки виявлення для кожної мови програмування та фреймворку, які використовують клієнти служби. Такі інструменти спрощують управління та взаємодію з різними мікросервісами, забезпечуючи зручний механізм для виявлення та спілкування з активними службами в розподіленому середовищі.

Використання шлюзу API є ключовим для керування вхідним та вихідним трафіком до мікросервісів. Цей спеціальний рівень надає ряд функцій, таких як маршрутизація, балансування навантаження та автентифікація. Завдяки шлюзу API оркестровка мікросервісів стає простішою, забезпечуючи єдину точку входу для вхідного та вихідного трафіку.

Однією з ключових переваг використання шлюзу API є його здатність до балансування навантаження, що забезпечує високу доступність мікросервісів та

ефективно обробляє значний обсяг трафіку. Прикладом шлюзу API може служити Ambassador Edge Stack, який вміло розподіляє трафік між мікросервісами, забезпечуючи таким чином високу доступність та масштабованість системи.

Для забезпечення узгодженості та ефективного керування конфігурацією мікросервісів важливо використовувати систему керування конфігурацією. Однією з ключових найкращих практик є відокремлення конфігурацій від коду програми та їх зберігання як змінні середовища, що полегшує розгортання та керування сервісами в різних середовищах.

Системи керування конфігурацією є інструментами, які сприяють ефективному управлінню конфігурацією інфраструктури та додатків. Вони дозволяють зберігати конфігурації в актуальному стані, полегшують впровадження змін, спрощують відкат до попередніх станів, поліпшують безпеку та зберігають конфіденційні дані, такі як ключі API чи облікові дані бази даних, в безпечному вигляді поза кодовою базою для уникнення випадкового розкриття.

Мікросервіси повинні бути відмовостійкими та стійкими, щоб забезпечити неперервне функціонування, навіть у випадку відмови одного чи кількох сервісів. Для досягнення цієї мети рекомендується впроваджувати такі функції, як автоматичні вимикачі, повторні спроби, тайм-аути, гнучка деградація та використання перегородок. Давайте розглянемо кожну з цих функцій більш детально:

- автоматичні вимикачі – використовуються для захисту від каскадних відмов, при збої мікросервісу автоматичний вимикач призупиняє потік трафіку до цього сервісу, дозволяючи системі відновитися та уникнути повного збою;
- повторні спроби – використовуються для обробки тимчасових збоїв, коли запит до мікросервісу не вдається, клієнт може повторити запит через короткий проміжок часу, що допомагає пом'якшити проблеми, спричинені тимчасовими помилками;
- тайм-аути – для уникнення блокування інших запитів, які займають надто багато часу;

– витончена деградація – зменшення функціональності у разі відмови, щоб уникнути повного виходу з ладу;
– перегородки – використовуються для ізоляції та локалізації збоїв в окремих секціях системи;
– надлишковість – реплікація мікросервісів для забезпечення доступності резервної копії в разі збою.

Ці практики спрямовані на те, щоб мікросервіси продовжували функціонувати, навіть у випадку збоїв, і забезпечували стійку та відмовостійку архітектуру.

Дотримання принципу єдиної відповідальності (SRP) у розробці програмного забезпечення передбачає, що кожен клас або модуль повинен мати лише одну причину для зміни. Хоча SRP безпосередньо не пов'язаний з оркестровкою мікросервісів, цей принцип залишається важливою практикою у розробці програмного забезпечення.

В контексті мікросервісів принцип SRP можна застосовувати до окремих мікросервісів. Кожен мікросервіс повинен мати чітку та конкретну відповідальність, уникаючи непотрібних залежностей від інших мікросервісів. Це сприяє створенню узгоджених та легко обслуговуваних мікросервісів, а також спрощує організацію розгортання та керування ними, оскільки кожен мікросервіс має чітко визначену відповідальність та набір залежностей.

1.5 Постановка задач дослідження

Метою роботи є дослідження методів та інструментів забезпечення безпеки в мікросервісних архітектурах, з фокусом на використанні технологій контейнеризації та оркестрації.

Основні завдання дослідження включають аналіз предметної галузі, де необхідно визначити основні концепції мікросервісної архітектури, а також розглянути роль контейнеризації в розподілених системах. Далі йде вивчення

сучасних технологій контейнеризації та оркестрації, що передбачає аналіз популярних інструментів контейнеризації.

Одним з ключових завдань є аналіз безпекових загроз та викликів у мікросервісних архітектурах, де слід визначити основні загрози та ризики, пов'язані з використанням мікросервісів, а також дослідити специфічні проблеми безпеки, що виникають при використанні контейнерів та оркестраційних систем. Наступним кроком є розробка методів забезпечення безпеки, що включає дослідження підходів до забезпечення безпеки на рівні контейнерів та мікросервісів, розробку рекомендацій щодо використання інструментів для моніторингу та управління безпекою, а також пропозицію стратегій захисту від потенційних загроз.

Експериментальне дослідження передбачає проведення експериментів з використанням обраних інструментів контейнеризації та оркестрації, випробування розроблених методів безпеки на практичних прикладах та оцінку ефективності запропонованих рішень в реальних умовах. Необхідно узагальнити результати дослідження, надати практичні рекомендації щодо впровадження розроблених методів у реальних проєктах та визначити напрямки подальших досліджень у цій галузі.

Основною метою роботи є дослідження методів забезпечення безпеки в мікросервісних архітектурах, що використовують технології контейнеризації та оркестрації, з урахуванням сучасних викликів та загроз інформаційної безпеки.

2 МЕТОДИ ВИРІШЕННЯ ПРОБЛЕМИ

2.1 Безпека в мікросервісних архітектурах

Навіть при стрімкому поширенні мікросервісів виявляється, що безпека програм не встигає за цим темпом. Відсутній універсальний метод захисту мікросервісів, і розробники повинні уважно розглядати кілька чинників при визначенні стратегії безпеки. Такі чинники включають ризики та терміни введення на ринок, і необхідно знайти баланс між високим рівнем безпеки та реальними потребами та ресурсами організації [5].

Безпека програм мікросервісів вирішує завдання з управління запитами на доступ від зовнішніх користувачів та інших служб. Усі запити на керування доступом можуть бути оброблені за допомогою механізму авторизації, такого як Open Policy Agent (OPA). Крім того, додатки можуть використовувати монолітний шлюз API, який охоплює весь рівень мікросервісу для обробки зовнішніх запитів від клієнтів.

Найкращі методи забезпечення безпеки архітектури мікросервісів. Поверхня атак мікросервісів більша, ніж у монолітних програм, і ця архітектура може мати більше вразливостей через кількість компонентів. Існує сім оптимальних підходів для забезпечення цілісності вашої програми.

Першим є підхід "shift-left" , використання якого передбачає, що аспекти безпеки програми враховуються на всіх етапах розробки. Тестування та оцінка продуктивності проводяться на ранніх етапах життєвого циклу. Практика DevSecOps сприяє розвитку навичок безпеки та обізнаності в командах і забезпечує, що фахівці з безпеки є невід'ємною частиною всього процесу розробки, а не консультуються лише наприкінці.

Використання підходу зсуву вліво означає, що безпека програми враховується на всіх етапах розробки. Тестування та оцінка продуктивності проводяться на початку життєвого циклу. Практика DevSecOps сприяє розвитку навичок безпеки та обізнаності в командах і гарантує, що спеціалісти з безпеки є частиною всього процесу розробки, а не консультуються лише наприкінці.

Для цієї мети можна використовувати тестування безпеки статичного аналізу (SAST) і тестування безпеки динамічного аналізу (DAST). Ось у чому різниця:

- SAST використовує сканер, сумісний із вашою мовою програмування, для виявлення вразливостей у вашому коді та бібліотеках, що використовуються;
- DAST імітує атаку ззовні і не обов’язково має бути на певній мові програмування.

Впровадження глибокого захисту, схоже на те, як в замках передбачено внутрішні стіни для випадку, якщо зловмисники подолають зовнішні бар'єри. Служби, які містять найбільш конфіденційні дані у програмі, також повинні мати декілька рівнів захисту. Виключення таких служб від зовнішнього доступу шляхом використання, наприклад, брандмауера, не є достатнім заходом безпеки. Слід також впроваджувати передові практики контролю доступу, такі як ідентифікація на основі маркерів та деталізована авторизація на основі політики для користувачів і служб. Додатковими рекомендованими заходами безпеки є шифрування даних і постійний моніторинг програми для виявлення підозрілої активності.

Ключовим елементом безпеки мікросервісів є захист від несанкціонованого доступу. Для здійснення аутентифікації користувачів слід використовувати стандартні засоби керування ідентифікацією та доступом (IAM), такі як SAML, WS-Fed, або стандарти OpenID Connect/OAuth2. Також рекомендується впровадження багатофакторної аутентифікації (MFA) для підвищення рівня безпеки.

У сфері авторизації в мікросервісних додатках можна використовувати Styra Declarative Authorization Service (DAS) для впровадження політик як коду та їх керування. В якості рішення для рівня управління політиками для OPA, Styra DAS дозволяє впроваджувати точне керування на великому масштабі, не втрачаючи швидкодії та продуктивності.

Розгортання шлюзів API. Шлюзи API є найбільш поширеним методом створення єдиної точки входу для всіх зовнішніх запитів доступу від систем і

клієнтів до всіх мікросервісів, розгорнутих у програмі. Ці компоненти також є основною точкою атак та вимагають особливої уваги до безпеки.

Багато шлюзів API, таких як Amazon API Gateway, мають вбудовані засоби управління та безпеки і часто розроблені з урахуванням масштабованості. Вони автоматизують завдання, такі як автентифікація та обмеження швидкості, а також захищають програму мікросервісу від несанкціонованого доступу та атак типу "відмова в обслуговуванні" (DDoS), які використовують великі обсяги трафіку.

Як і для більшості компонентів мікросервісів, API-шлюзи також вимагають надійної системи авторизації. Використання Styra DAS для управління Open Policy Agent (OPA) дозволяє відокремити логіку доступу від коду програми, забезпечуючи ефективне керування, моніторинг та аудит потоку трафіку.

Забезпечення безпеки контейнерів. Команди розробників часто використовують контейнери для мікросервісних додатків, спрощуючи роботу з багатьма компонентами та полегшуючи процеси розгортання. Ризики безпеки контейнерів включають скомпрометовані образи, помилки конфігурації та ізоляції, а також вразливості в операційній системі хоста, які можуть вплинути на всі контейнери, які працюють на цьому хості.

Для практики принципу безпеки з найменшими привілеями рекомендується вживати такі стратегії:

- застосовувати обмеження авторизації лише необхідними привілеями;
- уникати використання Sudo або привілейованих облікових записів для запуску чи виконання сервісів;
- обмежувати доступ і використання ресурсів;
- уникати зберігання секретів на контейнерних дисках;
- встановлювати правила ізоляції доступу до ресурсів.

Інструменти для оркестрації контейнерів, такі як Kubernetes, дозволяють автоматизувати процеси масштабування, розгортання та безпеки контейнерів. Відповідне дотримання найкращих практик з безпеки та авторизації в Kubernetes може значно полегшити процес розгортання.

Мікросервіси вимагають безпечних механізмів обміну запитами автентифікації та авторизації один з одним. Розглянемо кілька використовуваних методів:

– перевірені мережі: цей застарілий підхід базується на відсутності спеціальних заходів безпеки для зв'язку "сервіс-сервіс" і полагається лише на мережевий рівень безпеки, щоб запобігти зловмисникам взаємодіяти з мікросервісами ззовні;
– веб-токени JSON: це легкі автономні маркери, які використовуються для передачі інформації у вигляді криптографічно підписаного об'єкта JSON, забезпечуючи при цьому автентифікацію;
– взаємна безпека транспортного рівня (mTLS): кожен мікросервіс володіє відкритим/приватним ключем, який дозволяє двосторонню ідентифікацію під час взаємодії через mTLS;
– сервісна сітка: зі збільшенням кількості мікросервісів розробники можуть впровадити окремий рівень інфраструктури, відомий як сервісна сітка, для контролю за зв'язком між сервісами. Меш-сервіс додає проксі-сервіс до кожного мікросервісу і, як правило, використовує mTLS для комунікації з іншими проксі-серверами.

2.2 Методи забезпечення безпеки при використанні контейнерів

Забезпечення безпеки при використанні контейнерів є важливим етапом в розробці та управлінні мікросервісами [6]. Нижче подано деякі методи та стратегії для забезпечення безпеки контейнерів:

– використання принцип "найменших привілеїв" для контейнерів – кожен контейнер повинен працювати з якнайменшими можливостями та привілеями. Забезпечте, щоб контейнери мали обмежений доступ до ресурсів та прав для запуску операцій;
– запобігання використання привілейованих облікових записів для запуску чи виконання служб в контейнерах, що допомагає обмежити можливості зловмисників у випадку компрометації контейнера;

- використання механізмів автентифікації та авторизації для контролю доступу до контейнерів. Застосовуйте багатофакторну автентифікацію та обмеження доступу до необхідного мінімуму;
- обмеження ресурсів контейнерів, таких як пам'ять і обчислювальні ресурси, що дозволяє уникнути ситуацій, коли один контейнер може витіснути інші контейнери на спільному хості;
- у контейнерах, які містять конфіденційні дані, такі як паролі або ключі, має використовуватися має виконуватися шифрування даних як під час передачі, так і під час зберігання;
- важливим методом є інтеграція систем моніторингу та аудиту для виявлення аномальної активності або спроб несанкціонованого доступу;
- регулярне оновлення базових образів контейнерів використання тільки офіційно підтримуваних та безпечних образів;
- використання сервісних сіток та взаємної безпеки транспортного рівня (mTLS) для захисту міжсервісного зв'язку, що дозволяє кожному контейнеру взаємно ідентифікуватися під час спілкування;
- використання інструментів управління образами, таких як Docker Content Trust (DCT) та Notary, для забезпечення інтегрованості та безпеки образів контейнерів;
- управляння секретами (паролями, клчами), які використовуються у контейнерах та використовуйте інструменти для безпечного зберігання та передачі секретів, такі як Kubernetes Secrets або інші схожі рішення;
- застосування політики безпеки для контейнерів, які визначаються та виконуються на рівні оркестратора, такого як Kubernetes або Docker Swarm, що може включати обмеження доступу до мережі, ресурсів та ізоляцію контейнерів;
- контроль віддаленого доступу до контейнерів через API та використання шлюзи API для моніторингу, фільтрації та захисту трафіку, який надходить до контейнерів.

Забезпечення безпеки контейнерів – це комплексний процес, що вимагає поєднання технічних заходів та культури безпеки в організації. Захист відповідальності від самого початку розробки, регулярні аудити безпеки та відстеження нових вразливостей - це ключові елементи успішної стратегії безпеки контейнерів.

2.3 Засоби забезпечення безпеки в оркестраційних системах

Оркестраційні системи, такі як Kubernetes, Docker Swarm, або Apache Mesos, дозволяють ефективно керувати та масштабувати контейнеризовані додатки. Забезпечення безпеки в оркестраційних системах важливо для забезпечення інтегрованості, конфіденційності та доступності. Нижче подано деякі засоби та практики забезпечення безпеки в оркестраційних системах:

- RBAC (Role-Based Access Control) – контроль доступу на основі ролей дозволяє обмежувати права доступу користувачів та служб до різних ресурсів у кластері. Kubernetes, наприклад, має RBAC, який дозволяє адміністраторам визначати, хто може виконувати які дії;
- Network Policies – визначення політик мережі дозволяє контролювати комунікацію між різними компонентами у кластері. Це допомагає зменшити поверхню атак та забезпечити безпеку комунікації між контейнерами;
- TLS для комунікації – використання TLS для захисту мережевої комунікації між компонентами кластера є особливо важливо для захисту внутрішньокластерного обміну даними;
- Secrets Management – ефективне управління секретами, такими як паролі та ключі, в оркестраційних системах допомагає забезпечити безпеку конфіденційної інформації, яка використовується в додатках;
- аудит та логування – включення систем аудиту та ведення журналу подій усіх дій у кластері допомагає виявляти та реагувати на потенційні загрози безпеки;

- Image Security Scanning – використання інструментів для сканування безпеки образів контейнерів на вразливості перед їх розгортанням у кластері;
- оновлення та патчі – підтримання актуальних версій оркестраційної системи та компонентів, регулярне оновлення системи для усунення вразливостей;
- організаційна культура безпеки – вироблення культури безпеки в організації, що включає навчання персоналу, визначення політик безпеки та здатність ефективно реагувати на інциденти, є важливою частиною безпечного впровадження оркестраційних систем;
- декларативні конфігурації – використання декларативного підходу для конфігурації, який описує, що потрібно зробити, а не як це робити, що дозволяє уникнути конфігураційних помилок та підвищити безпеку;
- ізоляція сервісів – забезпечення ізоляції сервісів у кластері, задля запобігання поширенню атак та зменшення впливу можливих вразливостей.

Забезпечення безпеки в оркестраційних системах - це комплексний процес, що вимагає поєднання технічних та організаційних заходів для ефективного захисту контейнеризованих додатків.

3 АНАЛІЗ ТЕХНОЛОГІЧНИХ РІШЕНЬ ТА ІНСТРУМЕНТІВ ДЛЯ СТВОРЕННЯ МІКРОСЕРВІСНИХ СТРУКТУР

Під час цієї роботи проведено теоретичні та експериментальні дослідження. У рамках теоретичних досліджень було проведено аналіз, порівняльний аналіз та аналітичні дослідження. Щодо експериментальних досліджень, вони включали пошукові та виробничі дослідження.

Додаток, що базується на мікросервісах, представляє собою розподілену систему, що функціонує на декількох процесах або сервісах, іноді навіть на кількох серверах або вузлах. Зазвичай кожен екземпляр служби - це окремий процес. У зв'язку з цим служби повинні взаємодіяти за допомогою внутрішньопроцесного протоколу, такого як HTTP, AMQP або бінарний протокол, наприклад TCP, в залежності від характеру кожної служби.

Клієнти та послуги можуть взаємодіяти за допомогою різних типів зв'язку відповідно до сценарію та мети. Ці типи зв'язків можна розділити на дві основні категорії.

Група перша визначає, чи протокол є синхронним чи асинхронним. Синхронний протокол HTTP – це такий, де клієнт надсилає запит і чекає на відповідь від служби. Це не залежить від того, чи виконується код клієнта синхронно (потік блокується) чи асинхронно (потік не блокується, але відповідь врешті-решт буде надіслана). Важливо зазначити, що протокол (HTTP/HTTPS) є синхронним, і код клієнта може продовжити виконання завдання лише після отримання відповіді від сервера HTTP. Щодо асинхронних протоколів, таких як AMQP (протокол, який підтримується багатьма операційними системами та хмарними середовищами), вони використовують асинхронні повідомлення. Код клієнта або відправника повідомлення не очікує відповіді. Просто надсилається повідомлення, наприклад, при надсиланні повідомлення в чергу RabbitMQ або іншого брокера повідомлень.

Група одержувачів визначає, чи має запит одного або декількох отримувачів.

Для одного отримувача – кожен запит обробляється лише одним отримувачем або службою, наприклад, використовуючи шаблон Command.

Для декількох отримувачів – кожен запит може бути оброблений різною кількістю отримувачів, від нуля до декількох. Цей тип взаємодії має бути асинхронним.

Наприклад, механізм publish/subscribe, який застосовується в архітектурі, що керується подіями [7]. Він базується на інтерфейсі шини подій або брокері повідомлень, коли події оновлюють дані в декількох мікрослужбах. Зазвичай це реалізується через службову шину або аналогічний об'єкт, наприклад, службову шину Azure, за допомогою тем і підписок.

Часто в додатках, які базуються на мікрослужбах, використовують комбінацію різних стилів взаємодії. Одним з найпоширеніших є взаємодія з одним отримувачем за синхронним протоколом, таким як HTTP або HTTPS, що відбувається під час виклику стандартного веб-API HTTP. Для асинхронної взаємодії між мікрослужбами зазвичай використовують протоколи повідомлень.

3.1 Мікросервісні комунікації REST API та черги повідомлень

При роботі з мікросервісами ключовою аспектом проектування є спосіб взаємодії між сервісами. Зазвичай люди обирають RESTful HTTP API, що надається кожним сервісом, і потім інші сервіси викликають його через звичайний HTTP-клієнт. Це має свої переваги, такі як полегшення виявлення сервісів через DNS і API-шлюзи, але також приносить багато недоліків. Наприклад, якщо викликаний сервіс перестає відповідати, ваша клієнтська служба повинна мати механізм перепідключення або відновлення, інакше існує ризик втрати запитів та даних. Хмарна архітектура має бути надійною і здатною коректно відновлюватися після збоїв. Крім того, HTTP-запит є блокуючим API, що робить його синхронним.

Використання системи черги повідомлень є ефективним варіантом при взаємодії з декількома мережевими сервісами. Ця архітектура передбачає існування додаткового компонента, відомого як брокер повідомлень. Він відповідає за збір, маршрутизацію та доставку повідомлень від відправників до відповідних отримувачів.

Подібно до роботи поштової служби, архітектура черги повідомлень працює на тому принципі, що ви, як відправник (producer), надсилаєте листа (message) отримувачу (consumer), вказуючи адресу (логіку маршрутизації, наприклад, тему, за якою воно опубліковане) та передаючи лист місцевому поштовому відділенню (брокер повідомлень). Після цього ви можете забути про нього, оскільки далі про доставку листа піклується саме поштова служба. У контексті мікросервісної архітектури це є досить надійним рішенням, оскільки воно додає додатковий рівень абстракції (брокер повідомлень) між слабко зв'язаними сервісами та гарантує повноцінний асинхронний обмін даними.

У своїй докторській дисертації 2000 року під назвою "Архітектурні стилі та проектування мережевих програмних архітектур" Рой Філдінг [8] визначив поняття репрезентативного стану (REST). Працюючи у команді, яка займалася розробкою протоколу HTTP, він розробив основний набір принципів, властивостей та обмежень, що стали основою для концепції REST. У сучасному світі взаємодія з RESTful стала критично важливою для корпоративних обчислень, оскільки багато сервісів підтримують різноманітні API через мережу. З огляду на це, давайте розглянемо, які проблеми найкраще вирішує REST:

Сам процес синхронного запиту та відповіді у протоколі HTTP (який використовується для транспортування REST) можна розглядати як протокол обміну запитами та відповідями, що робить REST ідеальним для використання у сценаріях взаємодії за принципом запит-відповідь.

Оскільки HTTP є де-факто стандартом у сфері мережевого сполучення завдяки роботі IETF, API-інтерфейси, створені за допомогою REST, стають доступними для будь-якої мови програмування. Крім того, інструменти, такі як Swagger, сприяють легкій документації корисного навантаження повідомлення. Щодо безпеки, у середовищі Інтернету існує різноманітні загрози, але екосистема безпеки для REST включає надійні рішення, від брандмауерів до протоколу OAUTH.

Зараз багато компаній, завдяки популярності RESTful сервісів, увійшли в пастку використання REST як інструменту «все в одному». Ця популярність

частково обумовлена перевагами, які надає REST. Розробники звикли проектувати програми з синхронним запитом/відповіддю, через навчання API та баз даних розробників викликати метод та одразу очікувати відповіді. Проте ця перевелика залежність від REST та синхронних шаблонів має негативні наслідки, особливо щодо обміну даними між мікросервісами в рамках підприємства, і у деяких випадках суперечить принципам правильної архітектури мікросервісів.

Використання сервісів навколо інтерфейсу завжди передбачає тісний зв'язок, особливо коли мова йде про дані. Однак при використанні RESTful сервісу розробник припускає, що потрібно буде доставити повідомлення лише до одного місця. Проблема виникає, коли з'являється новий сервіс або компонент, які потребуються. Звичайно, можна оновити код, щоб додати нову кінцеву точку, але це призводить до зайвих з'єднань. Поступово ваш простий мікросервіс перетворюється на оркестратора, який втрачає головний атрибут мікросервісу – його відокремленість.

Блокування відбувається, коли ваш сервіс зупиняє свою діяльність, чекаючи на відповідь від REST-сервісу. Це призводить до зниження продуктивності програми, оскільки цей процес може обробляти інші запити. Можна уявити це так: якщо бармен приймає замовлення на напій, готує коктейль і потім терпляче чекає, доки клієнт закінчить пити, перш ніж обслуговувати наступного. Хоча це забезпечить гарний досвід для цього клієнта, інші будуть чекати і страждати від спраги. Щоб вирішити цю проблему, бар може найняти додаткових барменів, але потрібно мати одного бармена на кожного клієнта. Це важко для бару, особливо коли кількість відвідувачів коливається. Такі ж проблеми виникають, коли потоки блокуються в додатках, які очікують відповіді від служб RESTful.

Обробка помилок HTTP призначена для роботи в Інтернеті і часто виникає ситуація, коли браузер збивається, намагаючись отримати доступ до веб-сторінки. Зазвичай ми натискаємо кнопку оновлення, і сторінка з'являється знову. Однак якщо це не працює, що робити? Спробувати ще раз? Можливо, варто відпочити, випити чашку кави і повернутися через кілька хвилин? Нам важко визначити правильну стратегію, оскільки кожна веб-сторінка має своє унікальне споведження.

Чи не має сенсу заносити цю складну логіку повторення безпосередньо в код служби при виклику RESTful? При цьому сервіс стає ще більш залежним від інших сервісів, що суперечить ключовому принципу збереження архітектури мікросервісів, а саме - однозначній функціональності та невеликому обсязу.

Поєднання принципів керованої подіями з архітектурою мікросервісів дозволяє вирішити численні недоліки, що виникають при використанні RESTful/синхронних взаємодій. Мікросервіси, що працюють за принципами керованих подій, оповіщають про необхідність виконання дій асинхронно, коли це потрібно. Це особливо важливо в ситуаціях, коли взаємодія з великою кількістю об'єктів стає рутинною.

Переваги використання обміну повідомленнями для керованих подій мікросервісів різноманітні та численні. Спочатку, слабкий зв'язок. У випадку використання обміну повідомленнями, служби не мають прямого зв'язку одна з одною. Вони лише повідомляються про нові події, обробляють цю інформацію та створюють нові дані. Ця інформація може бути використана будь-якою кількістю служб, завдяки механізму публікації та підписки. Це дає можливість мікросервісам легко адаптуватися до постійних змін у підприємстві.

Неблокованість – мікросервіси мають працювати максимально ефективно, не витрачаючи зайвих ресурсів, навіть коли багато процесів заблоковані та очікують відповіді. За допомогою асинхронного обміну повідомленнями програми можуть відправляти запити та обробляти інші без очікування відповіді. Це можна зрозуміти через аналогію з офіціантом у барі. Офіціанти обслуговують кількох клієнтів та обробляють кілька замовлень одночасно, не блокуючи жодного клієнта та забезпечуючи задоволення кожному.

При зростанні додатків і підприємств потреба у динамічному розширенні стає однією з найважливіших переваг мікросервісної архітектури. Оскільки кожна послуга є невеликою і виконує лише одне завдання, вона повинна мати можливість збільшуватися або зменшуватися за необхідності. Управління подіями та обмін повідомленнями спрощують розширення мікросервісів, оскільки вони не пов'язані та не блокуються. Це також дозволяє легко визначити, яка послуга є обмежуючим

фактором, і додавати додаткові екземпляри лише цієї послуги, замість того, щоб сліпо розширювати всі послуги, що може трапитися, якщо мікросервіси пов'язані між собою синхронним зв'язком. Можливість розширення за допомогою обміну повідомленнями була доведена такими компаніями, як Linkedin та Netflix.

Таблиця 3.1 – Порівняння основних критеріїв між чергою повідомлень (MQ) та REST API.

Повідомлення	REST API	Message Queue
Використання	Можливість взаємодії з будь-якою програмною мовою та платформою	Можна використовувати майже будь-яку мову програмування і вона сумісна з усіма платформами
Масштабованість	Необхідно впровадити балансувальник навантаження, щоб забезпечити розподіл тяг між діючими екземплярами системи, що вимагається для процесу масштабування	Зменшує розмір мікросервісів, оскільки може опрацьовувати будь-яку кількість інстанцій без додаткових налаштувань. Черга може функціонувати як з одним отримувачем, так і з безліччю
Ефективність	Робота в синхронному режимі означає, що ваш REST сервіс зупиняється, чекаючи на відповідь. Це призводить до зниження	Програма досягає високої продуктивності завдяки використанню асинхронного обміну повідомленнями, при

Продовження таблиці 3.1

Повідомлення	REST API	Message Queue
	продуктивності програми, оскільки інші запити до	якому вона надсилає запити та обробляє інші

	REST сервісу не можуть бути оброблені протягом цього часу.	запити, не чекаючи на відповідь.
Стійкість до відмов	Має меншу стійкість до відмов, оскільки потребує правильної настройки балансувальника навантаження, який має відстежувати працездатність кожного екземпляра. Якщо балансувальник вийде з ладу, система може зазнати повної відмови.	Має високу стійкість до відмов. Якщо один елемент системи недоступний, інший може продовжувати взаємодіювати з чергою. Черги забезпечують сталість вашої інформації та зменшують кількість помилок, що можуть виникнути при відключенні різних частин вашої системи.
Зв'язок	Інтерфейс тісно пов'язаний з сервісами, оскільки існує певна залежність сервісів від нього. Коли REST-сервіс викликається, важливо мати уявлення, хто здійснює цей виклик та кому потрібно надіслати дані, щоб вони були оброблені правильно. Любі значущі зміни в	Існує слабке зв'язання, яке дозволяє мікросервісам легко адаптуватися до безлічі змін. Кожен мікросервіс повідомляє черзі про нові повідомлення, потім очікує відповіді і повертає її. Він абсолютно не знає про вміст повідомлення та

Кінець таблиці 3.1

Повідомлення	REST API	Message Queue

	сторонніх сервісах зазвичай потребують адаптації.	не цікавиться, від кого воно прийшло чи кому відправлено.
Захист	Протокол TLS може бути використаний, але для його впровадження потрібно вносити зміни до програми як відправника, так і одержувача, що може призвести до додаткових проблем.	Використовується протокол Advanced Message Security для захисту повідомлень, не потребуючи додаткових змін у програмі.

З таблиці зрозуміло, що системи черги повідомлень гарно інтегруються в мікросервісний підхід та рекомендуються для використання у веб-додатках.

3.2 Фреймворки передачі даних через використання черги повідомлень

Apache Kafka та RabbitMQ – це дві відкриті системи обміну повідомленнями на основі підписок, які легко інтегруються в проекти. RabbitMQ, що вийшов у 2007 році, є старішим інструментом і використовувався як основний компонент для систем обміну повідомленнями та SOA [9], а нині також активно застосовується у потокових сценаріях. Kafka, випущений у 2011 році, є новішим інструментом, спеціально розробленим для потокових сценаріїв з самого початку. RabbitMQ – це універсальний брокер повідомлень, який підтримує протоколи MQTT, AMQP та STOMP та здатен ефективно працювати в різноманітних сценаріях (рис. 3.1).

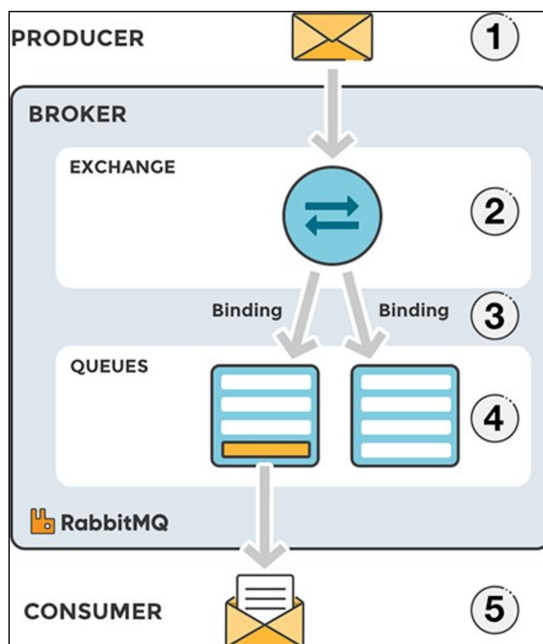

Рисунок 3.1 – Структура функціонування системи повідомлень брокера RabbitMQ

Kafka – це платформа для передачі повідомлень, призначена для відтворення вхідних даних та потоків. Цей надійний брокер повідомлень дозволяє програмам обробляти, зберігати та повторно обробляти дані у потоці. Використовуючи простий метод маршрутизації за допомогою ключа, Kafka надсилає повідомлення на відповідний топік [10] (рис. 3.2).

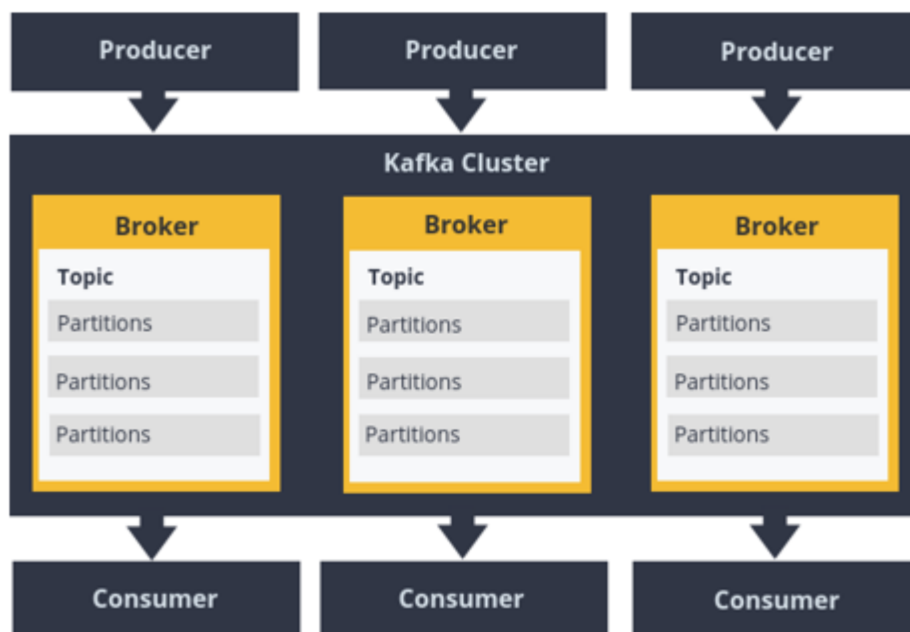

Рисунок 3.2 – Принцип дії системи повідомлень Kafka

Було здійснено порівняльний аналіз характеристик шин повідомлень між RabbiyMq та Apache Kafka, дивитись таблицю 3.2

Таблиця 3.2 – Порівняння характеристик шин повідомлень

Повідомлення	REST API	Message Queue
Доступ	Забезпечує кластеризацію та високу доступність черг	Передбачає сервер з відкритим вихідним кодом для керування станом кластера
Розподіл і паралельність	Можливість масштабування кількості отримувачів дозволяє мати багато отримувачів для кожного екземпляра черги, які конкурують за отримання повідомлення. У цій конфігурації обробка повідомлень розподіляється між усіма активними отримувачами, але кожне повідомлення може бути отримано лише один раз.	Розподіл споживачів відбувається за тематичними розділами, де кожен отримувач у групі призначений для одного з розділів. Ви можете використовувати механізм розділів для надсилання кожному розділу різного набору повідомлень за бізнес-ключем
Ефективність	Також, має високий рівень ефективності, що дозволяє обробляти мільйони повідомлень за секунду, але потребує більше ресурсів	Пропонує значно більшу ефективність, використовуючи послідовне дискове введення-виведення для підвищення продуктивності. Це може забезпечити

Продовження таблиці 3.2

Повідомлення	REST API	Message Queue
		високу пропускну здатність при обмежених ресурсах, що необхідно в сценаріях використання великих обсягів даних
Процес реплікації	Черги не автоматично реплікуються і потребують налаштувань. Якщо у вашому кластері є щонайменше три вузли, все, що вам потрібно зробити, це оголосити вашу чергу як чергу кворуму, яка буде забезпечувати резервне копіювання	Брокер автоматично копіює інформацію, і якщо головний брокер перестає працювати, функції автоматично передаються іншому брокеру, який має повну копію даних з недіючого брокера, тож повідомлення не втрачаються
Зростання кількості підписників	Повідомлення можуть чекати в черзі, де є багато споживачів. У кожній черзі тільки один одержувач, але якщо повідомлення потрапляє до кількох черг, його може обробити кілька споживачів	Декілька одержувачів можуть підписатися на повідомлення, що означає, що його можуть одночасно обробити декілька одержувачів
Послідовність повідомлень	Існує безпорядок у повідомленнях. Їх можна організувати, визначивши набір черг і кожне повідомлення	Можна отримати послідовність у розділах модуля за допомогою ключів

Продовження таблиці 3.2

Повідомлення	REST API	Message Queue
		повідомлень
	відправляти в іншу чергу. Проте на великому масштабі це може бути проблематичним	повідомлення. Обравши відповідний ключ, ви отримаєте розділ (топік) з впорядкованими повідомленнями
Протоколи передачі повідомлень	Підтримує всі стандартні протоколи черги, такі як AMQP, STOMP (на базі тексту), MQTT (спрощений обмін повідомленнями публікації або підписки) та HTTP	Підтримує основні типи даних (int8, int16, int32, int64, рядки, масиви) та двійкові повідомлення
Час життя повідомлень	У формі черги, повідомлення автоматично видаляються після використання та підтвердження. У RabbitMQ можна налаштувати постійне збереження повідомлень, позначивши чергу і повідомлення як постійні	У вигляді журналу, повідомлення завжди доступні там, і ви можете контролювати це, налаштовуючи політику зберігання повідомлень
Гнучкість у виборі маршрутування	Розширені можливості, такі як використання регулярних виразів та підстановочних знаків, дозволяють переглядати документи для отримання додаткової інформації	Повідомлення надсилається в топік з використанням ключа

Кінець таблиці 3.2

Повідомлення	REST API	Message Queue
Пріоритетність сповіщень	Здатність визначити пріоритети повідомлень, а також використані повідомлення з високою пріоритетністю	Відсутність. Можливо реалізувати за допомогою клавіш повідомлень, але в масштабах це може бути складним
Слідкування	Існує вбудований інтерфейс управління, доступний через обраний порт. Також передбачена вбудована інтеграція з популярними системами слідкування Grafana та Prometheus	Можна користуватися зовнішніми інструментами (Confluent, Landoop, Kafka Tool)
Підтвердження	На обох платформах відправники отримують підтвердження про те, що повідомлення було додано до черги або топіку, і отримувачі отримують підтвердження, коли повідомлення успішно отримано, тому можна бути впевненим, що повідомлення не пропадуть по дорозі	

Після аналізу таблиці 3.2 ми можемо зробити наступні висновки. Kafka варто використовувати у таких випадках:

- коли потрібно надіслати повідомлення багатьом одержувачам;
- коли необхідна висока швидкодія (мільйони повідомлень за секунду);
- для обробки потоків даних;
- для створення резервних копій черг;
- для забезпечення високої доступності;
- коли важливий порядок повідомлень.

RabbitMQ буде корисним у таких ситуаціях:

- коли потрібно надіслати повідомлення лише одному одержувачу;

– для гнучкої маршрутизації;

– для встановлення пріоритетів черги;

– для використання повідомлень у стандартному протоколі.

Загалом, можна сказати, що RabbitMQ підходить для простих сценаріїв використання черги повідомлень з невеликим обсягом даних. Він має свої переваги, такі як пріоритетна черга та гнучкі параметри маршрутизації. Проте для обробки великих обсягів даних та високої пропускної здатності рекомендується використовувати Kafka. Крім того, якщо вам потрібна підтримка журналів або декілька одержувачів для одних і тих самих повідомлень, краще звернутися до Kafka, оскільки RabbitMQ не може надати такої функціональності.

RabbitMQ – це безкоштовне рішення для створення черг повідомлень. Це брокер повідомлень, який підтримує розширений протокол черги повідомлень AMQP, а також може працювати з іншими популярними рішеннями обміну повідомленнями, такими як MQTT. Він має високу доступність, стійкість до відмов і можливість масштабування. RabbitMQ реалізований на базі Erlang OTP, технології, призначеної для створення стабільних, надійних, відмовостійких та добре масштабованих систем, які мають вбудовані засоби обробки великої кількості одночасних операцій.

При розробці веб-системи, ми можемо розглядати RabbitMQ як проміжний рівень програмного забезпечення. Він дозволяє різним сервісам у вашому додатку спілкуватися між собою, не хвилюючись про втрату повідомлень та забезпечуючи різні стандарти обслуговування. Також, він забезпечує ефективну маршрутизацію повідомлень, що дозволяє широке застосування додатків. RabbitMQ особливо корисний у випадках, коли потрібна гнучка архітектура, яка може адаптуватися до потреб багатьох мікросервісів, що постійно змінюються.

Основна мета роботи RabbitMQ полягає у прийманні та передачі повідомлень. Можна уявити його як систему доставки пошти: коли ви вкладаєте листа до поштової скриньки для відправлення, ви можете бути впевнені, що він буде доставлений адресатові завдяки поштовому відділенню та листоноші. У цьому

контексті RabbitMQ виступає як поштова скринька, поштове відділення та листоноша.

Давайте розглянемо головні складові, які використовуються під час роботи з такою системою:

- виробник (producer) – це програма, яка відправляє повідомлення.
- черга (queue) – це назва поштової скриньки всередині RabbitMQ. Хоча повідомлення проходять через RabbitMQ та ваші програми, вони можуть зберігатися лише у черзі. Черга обмежена лише обсягом пам'яті хоста та диском, по суті, це великий буфер повідомлень. Багато виробників можуть надсилати повідомлення, які потрапляють в одну чергу, і багато споживачів можуть намагатися отримувати дані з однієї черги.
- споживач (consumer) – це програма, яка очікує отримання повідомлення з черги.

3.3 Організація мікросервісів за допомогою інструментів для розгортання

Ефективні методи розгортання є ключовим чинником у створенні надійних та стабільних мікросервісів. У порівнянні з монолітними програмами, де можна оптимізувати розгортання лише для одного випадку використання, методи розгортання мікросервісів повинні масштабуватися для кількох служб, написаних різними мовами та мають власні залежності. Необхідно мати довіру до процесу розгортання, щоб випускати нові функції та послуги, не порушуючи загальної доступності або не внести критичні дефекти.

Розвиток програми мікросервісів відбувається на рівні окремих підрозділів, які можна швидко розгортати. Вартість впровадження нових сервісів повинна бути невеликою, щоб забезпечити швидке впровадження інновацій та користь для користувачів. Отримана з мікросервісів додаткова швидкість у розробці буде марною, якщо не буде забезпечено швидке та надійне їх запуск у виробництво. Автоматизоване розгортання важливе для успішного масштабування розробки мікросервісів.

3.3.1 Ідея та переваги використання контейнеризації

Процес розділення програмного коду з монолітної структури та його включення до окремого мікросервісу відомий як контейнеризація. Це метод легкої віртуалізації та ізоляції ресурсів на рівні операційної системи, що дозволяє запускати програми з мінімальним набором системних бібліотек у повністю стандартизованому контейнері.

Контейнеризація стала основним напрямком у розробці програмного забезпечення, як альтернатива або доповнення до віртуалізації. Вона передбачає упаковку програмного коду разом з усіма необхідними залежностями для його роботи у будь-якій інфраструктурі з метою забезпечення єдності та стабільності. Ця технологія стрімко розвивається, що забезпечує значні переваги для розробників, робочих груп та всієї інфраструктури програмного забезпечення.

Контейнеризація дозволяє розробникам прискорити процес створення та розгортання програм, забезпечуючи їх безпеку. Зазвичай код розробляється для конкретного обчислювального середовища, і при його переміщенні можуть виникати помилки. Наприклад, коли код, створений для Linux, переносять на операційну систему Windows. Використання контейнерів вирішує цю проблему, оскільки вони об'єднують код програми разом з налаштуваннями, бібліотеками та іншими необхідними компонентами в єдиний пакет, який абстрагується від операційної системи хоста. Такий підхід робить контейнери переносними і здатними працювати на будь-якій платформі або в хмарі без проблем.

Контейнери часто описують як "легкі", оскільки вони використовують спільне ядро операційної системи комп'ютера і не потребують додаткових витрат на операційну систему для кожного додатка. Вони мають меншу ємність, ніж віртуальні машини, та запускаються швидше, що дозволяє працювати з більшою кількістю контейнерів на одній і тій же обчислювальній потужності, яка використовується однією віртуальною машиною. Це підвищує ефективність роботи сервера і, відповідно, зменшує витрати на сервер та ліцензування.

У спрощеному вигляді, контейнеризація дозволяє розробляти програми один раз і запускати їх де завгодно. Ця мобільність є важливою з точки зору процесу

розробки та сумісності з постачальником. Вона також має інші відомі переваги, такі як ізоляція відмов, простота управління та безпека, а також багато інших.

Контейнеризація упаковує додаток разом із усіма потрібними файли конфігурації, бібліотеками та залежностями, утворюючи єдиний пакет програмного забезпечення, що може виконуватися. Кожен контейнер працює відокремлено від операційної системи хоста, завдяки чому вони не впливають на неї. Це досягається за допомогою двигуна з відкритим вихідним кодом, який дозволяє контейнерам спільно використовувати операційну систему з іншими контейнерами на тій же обчислювальній системі.

Користуючись контейнеризацією, розробники можуть швидше та безпечніше створювати та розгортати програми, незалежно від їхньої архітектури – чи це традиційний монолітний додаток або модульний мікросервісний набір. Також нові хмарні додатки можуть бути створені відповідно до принципів контейнеризації, розбиваючи складні програми на невеликі, спеціалізовані та керовані сервіси.

Існуючі програми можна упакувати в контейнери та контейнерні мікросервіси, що дозволить ефективніше використовувати обчислювальні ресурси. Контейнеризація має чимало переваг для розробників і розробницьких команд. Розглянемо головні з них:

Переносність: Контейнер створює пакет програмного забезпечення, що ізольований від операційної системи хоста та може працювати однаково ефективно на будь-якій платформі або у хмарному середовищі.

Гнучкість: Docker Engine, який використовується для запуску контейнерів, став промисловим стандартом для їх управління, надаючи розробникам прості інструменти та універсальний підхід до пакування, який працює як у Linux, так і у Windows. Контейнерна екосистема перейшла на механізми, які регулюються Ініціативою відкритих контейнерів (OCI), що дозволяє розробникам програмного забезпечення продовжувати використовувати гнучкі інструменти або процеси DevOps для швидкої розробки та покращення програм.

Швидкість. Контейнери часто описують як "легші", оскільки вони використовують ядро операційної системи (ОС) комп'ютера спільно, не

перевантажуючи його додатковими витратами. Це не лише поліпшує ефективність роботи серверів, але й зменшує витрати на сервер та ліцензування, а також прискорює запуск, оскільки ОС не потрібно повністю перезавантажуватися.

Ізоляція помилок. Кожен контейнерний додаток працює ізольовано і не залежить від інших. Відмова одного контейнера не має впливу на роботу інших контейнерів. Розробники можуть виявляти та виправляти будь-які технічні проблеми в одному контейнері без перебоїв у роботі інших. Крім того, механізм контейнерів може використовувати будь-які методи ізоляції безпеки ОС, такі як управління доступом SELinux для запобігання впливу збоїв у контейнерах.

Ефективність. Програмне забезпечення, яке працює в контейнерах, спільно використовує ядро ОС комп'ютера, і рівні програм у контейнері можуть ділитися між контейнерами. Таким чином, контейнери за своєю природою мають менші витрати, ніж віртуальні машини, і потребують менше часу на запуск, дозволяючи працювати значно більшій кількості контейнерів на тій самій обчислювальній потужності, що й одна віртуальна машина. Це підвищує ефективність роботи сервера, зменшує витрати на сервер та ліцензування.

Керування контейнерами стає простішим завдяки платформі оркестрації, яка автоматизує процеси встановлення, масштабування та управління. Вона спрощує завдання, такі як масштабування програм, розгортання нових версій та забезпечення моніторингу, ведення журналу та налагодження. Кубернетес є найпопулярнішою системою оркестрації контейнерів, базується на відкритому вихідному коді від Google та працює з різними механізмами контейнерів, такими як Docker, а також з будь-якою системою, що відповідає стандартам OCI.

Щодо безпеки, ізоляція контейнерів захищає від вторгнень шкідливого коду до інших контейнерів або системи хоста. Крім того, можна налаштувати правила безпеки для автоматичного блокування небажаних компонентів та обмеження зв'язку з непотрібними ресурсами.

Розробники програмного забезпечення вважають мікросервіси відмінним підходом до створення та управління додатками порівняно з колишньою монолітною моделлю, де програмний продукт, його інтерфейс користувача та база

даних об'єднувалися на одному сервері. За допомогою мікросервісів складний додаток розбивається на набір менших, більш спеціалізованих сервісів, кожен з власною базою даних та бізнес-логікою. Ці сервіси взаємодіють через загальні інтерфейси, такі як API та REST (наприклад, HTTP). Використовуючи мікросервіси, групи розробників можуть зосередитись на оновленні конкретних частин програми, не втручаючись в її загальну структуру, що прискорює розробку, тестування та впровадження.

Основні ідеї за мікросервісами та контейнеризацією подібні, оскільки обидва підходи до розробки програмного забезпечення перетворюють додатки на невеликі, переносні, масштабовані, ефективні та легкі в управлінні сервіси чи компоненти. Більше того, ці дві концепції добре співпрацюють. Контейнери забезпечують просте ізолювання будь-якої програми, незалежно від того, чи є вона традиційним монолітом чи модульним мікросервісом. Мікросервіс, розроблений у контейнері, отримує всі переваги контейнерізації - зручність у розробці, незалежність від інфраструктури (постачальника), гнучкість, ізоляцію помилок, ефективне використання ресурсів сервера, автоматизацію установки, масштабування та управління, а також безпеку, серед іншого.

Контейнери, мікросервіси та хмарні обчислення працюють у співпраці, щоб підняти розробку та доставку додатків на новий рівень, що не досяжно за допомогою традиційних методів та середовищ. Ці передові підходи додають гнучкості, ефективності, надійності та безпеки до життєвого циклу розробки програмного забезпечення – це веде до швидшої доставки додатків та інновацій для кінцевих користувачів та ринку. Ідея контейнеризації та ізоляції процесів існує вже давно, але поява Docker Engine у 2013 році, який став індустріальним стандартом для контейнерів, завдяки простим інструментам розробника та універсальному підходу до упаковки, значно прискорила впровадження цієї технології. Останні дослідження прогнозують, що понад 50% компаній будуть використовувати контейнерні технології до 2020 року [11].

Коли мова йде про контейнери, перш за все ми маємо на увазі Docker – це відкрита технологія, яка використовується для створення, тестування, постачання

та запуску веб-застосунків у середовищах з підтримкою контейнеризації. Вона допомагає ефективніше використовувати систему та ресурси, швидко розгортати готові програмні продукти, а також масштабувати їх та переносити в інші середовища з гарантованим збереженням стабільної роботи. Через свою популярність ця технологія стала синонімом слова «контейнер».

Запуск Docker викликає лише декілька системних процедур, і ядро створює для нового процесу окремий простір, окрему віртуальну мережу, окремий набір обмежень по ресурсах [12]. Процес, який «запущений у docker», фактично працює не в якійсь віртуальній машині (немає ніякого емулятора справжньої машини, ніякої віртуальної сутності), а запущений на тій же машині, з тим же ядром.

Давайте розглянемо основні характеристики та можливості Docker контейнера (рис. 3.3):

- вміст – в контейнері Docker може міститися щось внутрішнє або зовнішнє;
- портативність – Docker контейнер можна використовувати на різних пристроях, будь то на вашому комп'ютері, комп'ютері колеги або серверах хмарного провайдера. Це схоже на коробку з іграшками, яку можна легко переносити з місця на місце;
- зручний інтерфейс доступу – Docker контейнер має чіткі інтерфейси для взаємодії з зовнішнім світом. Ви можете відкрити порти для доступу через веб-браузер або налаштувати командний рядок для роботи з даними;
- віддалений доступ – Docker контейнер зберігає образ, який визначає його структуру. Коли потрібно створити контейнер, ви можете це зробити з образу, який містить Dockerfile з інструкціями для створення нового контейнера, операційну систему, бібліотеки та кодову базу, необхідну для запуску програми.

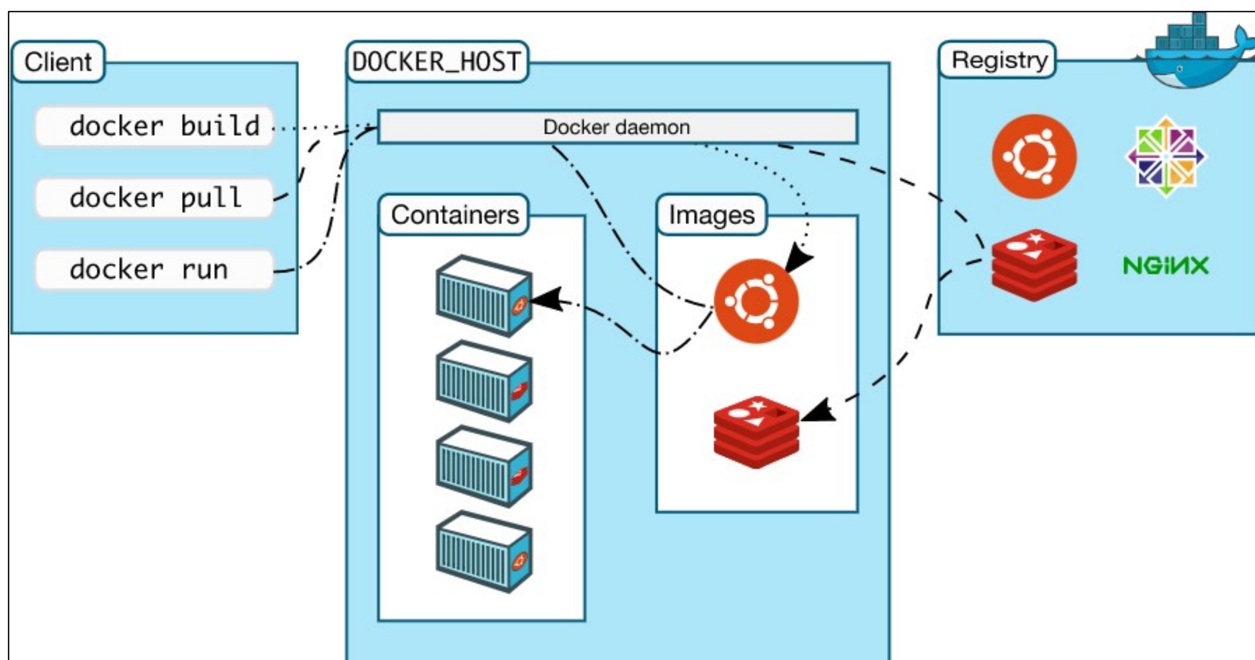

Рисунок 3.3 – Діаграма роботи Docker

Використання Docker контейнерів дозволяє ефективно та послідовно доставляти необхідні програми. Ця технологія оптимізує процес розробки, дозволяючи розробникам працювати в стандартизованих середовищах за допомогою локальних контейнерів, що містять їх програми та сервіси. Контейнери ідеально підходять для безперервної інтеграції та процесів з безперервною доставкою (CI/CD).

Розробники можуть кодувати локально та спільно працювати з колегами, використовуючи Docker контейнери. Вони використовують Docker для впровадження своїх програм в тестове середовище та проведення автоматичних та ручних тестів. Якщо виникають помилки, розробники виправляють їх у середовищі розробки та повторно розгортають їх у тестовому середовищі для перевірки. Після завершення тестування виправлення можна легко відправити клієнту, надіславши оновлений образ до виробничого середовища.

Платформа Docker забезпечує високу переносимість робочих навантажень. Контейнери Docker можуть бути використані на різних пристроях, таких як локальні комп'ютери розробників, фізичні або віртуальні машини у центрах обробки даних, у хмарних сервісах або в різних середовищах. Портативність та

простота використання Docker допомагають спрощувати динамічне керування робочими навантаженнями, масштабування та видалення додатків і служб відповідно до потреб бізнесу навіть у реальному часі.

Docker використовує архітектуру клієнт-сервер. Клієнт Docker взаємодіє з демоном Docker, який відповідає за збірку, запуск та розповсюдження ваших контейнерів Docker. Клієнт Docker та демон можуть працювати локально на одній системі, або ви можете підключити клієнт Docker до віддаленого демона Docker. Вони спілкуються за допомогою REST API, через UNIX-сокети або мережевий інтерфейс.

Докер-демон (Docker daemon) слухає запити API Docker та керує складовими Docker, такими як образи, контейнери, мережі та томи. Також може взаємодіяти з іншими демонами для управління сервісами Docker. Докер-клієнт - основний інструмент взаємодії багатьох користувачів Docker з самим Docker. Коли використовуються команди, такі як docker run, клієнт передає їх до демона Docker для виконання. Клієнт Docker може спілкуватися з декількома демонами через API Docker.

У Docker-реєстрі зберігаються образи Docker. Docker Hub - це загальнодоступний реєстр, який може використовувати кожен, і Docker за замовчуванням налаштований на пошук образів у Docker Hub. Ви навіть можете запустити свій власний реєстр. Якщо ви використовуєте Docker Datacenter (DDC), воно включає Docker Trusted Registry (DTR). При використанні команд Docker Pull або Docker Run, необхідні образи витягуються з налаштованого реєстру. При використанні команди docker push ваш образ зберігається у вашому налаштованому реєстрі.

Docker – це легкий та швидкий інструмент. Він забезпечує життєздатну та економічну альтернативу віртуальним машинам, тому є можливість використовувати більше обчислювальних потужностей для досягнення бізнес-цілей. Docker ідеально підходить для високо щільних середовищ та для малих та середніх розгортань, де потрібно зробити більше за менше використання ресурсів.

3.3.2 Опис використання контейнерів, їх переваги та основні принципи функціонування

Розвиток технологій змінив підхід до створення архітектури програмних додатків. Docker, хмарні сервіси та системи оркестрації контейнерів дозволили нам створювати розподілені, більш масштабовані та надійні рішення.

Оркестрація контейнерів автоматизує процеси розгортання, управління, масштабування та налаштування мережі контейнерів. Підприємства, які мають потребу в розгортанні та керуванні сотнями або тисячами контейнерів та серверів на базі Linux, можуть скористатися перевагами оркестрування контейнерів.

Системи оркестрації контейнерів можна використовувати в будь-якому середовищі, де використовуються контейнери. Це може спростити розгортання одного й того ж додатку в різних середовищах, без необхідності повторного проектування. Мікросервіси в контейнерах допомагають організувати різні сервіси, включаючи сховища, мережі та забезпечення безпеки.

Розглянемо завдання, які можна вирішити за допомогою оркестрації контейнерів:

- постачання та розгортання програмного забезпечення;
- налаштування та управління програмами;
- ефективний розподіл ресурсів;
- автоматичне масштабування або видалення контейнерів на основі оптимізації робочих навантажень у інфраструктурі програм;
- забезпечення балансування навантаження та ефективної маршрутизації трафіку;
- постійний моніторинг стану контейнерів;
- налаштування програм на основі контейнера, у якому вони будуть працювати;
- забезпечення взаємодії між контейнерами.

При використанні інструментів оркестрації контейнерів, таких як Kubernetes або Docker Swarm, зазвичай описується конфігурація програми у файлі YAML або

JSON, залежно від вибраного інструменту. У цих файлах конфігурації вказується, де збирати образи контейнерів, як встановлювати мережу між ними, як монтувати томи зберігання та де зберігати журнали. Зазвичай, групи керують версіями цих конфігураційних файлів для розгортання програм у різних середовищах перед розгортанням у робочих кластерах.

Контейнери розгортаються на хостах у групах, які зазвичай реплікуються. При розгортанні нового контейнера у кластері, інструмент управління контейнерами планує розміщення та шукає найбільш підходящий хост на основі обмежень, таких як доступність ЦП або пам'яті. Можна також використовувати мітки або метадані для розміщення контейнерів або враховувати їх близькість до інших хостів - існують різноманітні можливості обмежень.

При запуску контейнера на хості, інструмент оркестрації керує його життєвим циклом відповідно до вказівок, описаних у файлі визначення контейнера. Основна перевага інструментів оркестрації контейнерів полягає в тому, що вони можуть працювати в будь-якому середовищі, де можна запускати контейнери. На сьогодні контейнери підтримуються майже у всіх середовищах, від традиційних локальних серверів до хмарних інстансів.

Існує низка платформ для оркестрації контейнерів, які дозволяють ефективно розгортати контейнерні системи та будувати єдину централізовану консоль для управління. Сьогодні такі платформи набувають все більшого інтересу та розвитку у високонавантажених середовищах розробки. Найвідоміші з них – Kubernetes, Docker Swarm та Apache Mesos.

Kubernetes – це відкрита система для управління контейнерними кластерами. Благодаря тому, що Google відкрила її код та зробила доступною, вона стала найбільш популярною (рис. 3.4). Сьогодні багато хто розглядає її як найкращий та універсальний засіб для роботи з додатками у хмарному середовищі [13].

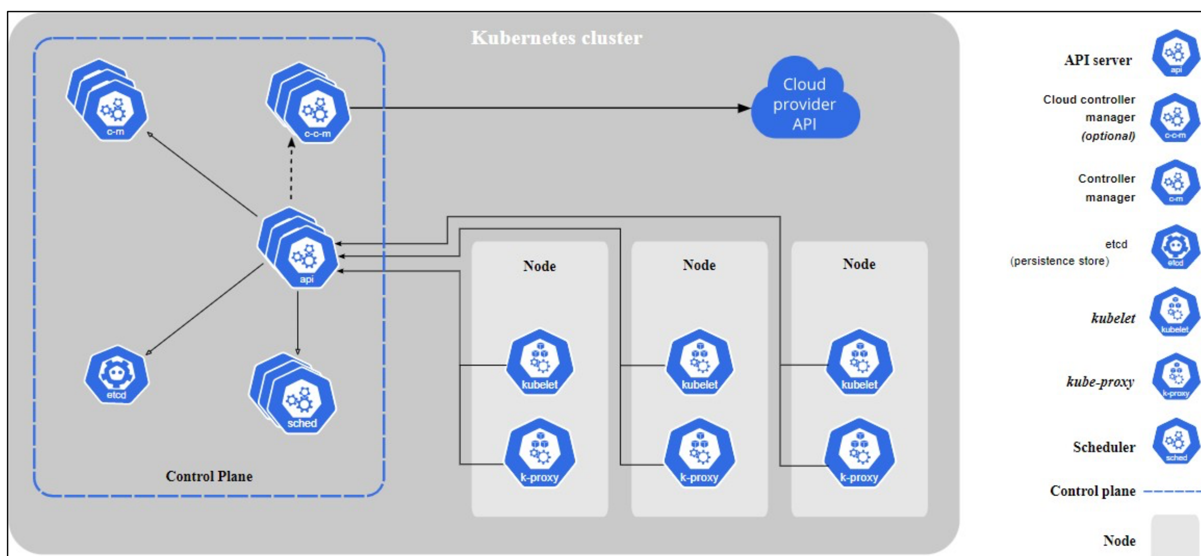

Рисунок 3.4 – Елементи кластера Kubernetes

Спочатку розроблений як відгалуження від проекту Borg компанії Google, Kubernetes визнаний фактичним стандартом для оркестрації контейнерів. Це ключовий проект фонду Cloud Native Computing Foundation, який отримав підтримку таких великих гравців, як Google, Amazon Web Services (AWS), Microsoft, IBM, Intel, Cisco та RedHat. Розглянемо основні компоненти архітектури Kubernetes нижче.

Кластер, або група, представляє собою набір вузлів, які складаються принаймні з одного головного вузла та декількох робочих вузлів (іноді називаються міньйонами), які можуть бути віртуальними або фізичними машинами.

Майстер Kubernetes відповідає за планування та розгортання екземплярів додатків на вузлах. Повний набір сервісів, що запускається майстром, називається площиною управління. Майстер взаємодіє з вузлами через сервер API Kubernetes [14]. Планувальник визначає розподіл модулів (одного або декількох контейнерів) на вузли відповідно до обмежень ресурсів та визначених політик.

Kubelet – це процесний агент, що управляє станом вузла Kubernetes, забезпечуючи запуск, зупинку та обслуговування контейнерів додатків відповідно до інструкцій з площини управління. Всю необхідну інформацію kubelet отримує від сервера API Kubernetes.

Поди – це основний елемент планування, який складається з одного або кількох контейнерів, розміщених на хост-машині та спільно використовують ресурси. Кожному модулю присвоюється унікальна IP-адреса в кластері, що дозволяє застосуванню використовувати порти без конфліктів. Бажаний стан контейнерів у модулі визначається через об'єкт YAML або JSON, відомий як PodSpec, який передається в kubelet через сервер API.

Розгортання, репліки та набори реплік - це об'єкти YAML, що визначають модулі та кількість екземплярів контейнера (реплік), для кожного модуля. Кількість реплік, які запускаються в кластері, визначається за допомогою ReplicaSet, який є частиною об'єкта розгортання. Якщо, наприклад, вузол, на якому запущено модуль, відмовляє, набір реплік забезпечить, що інший модуль запланований на іншому вузлі.

Інструмент контейнерної кластеризації Docker Swarm був представлений трохи пізніше і став частиною платформи Docker. Він дозволяє групувати хости Docker у спільний віртуальний хост. Docker Swarm особливо цікавий для малих та середніх підприємств (рис. 3.5).

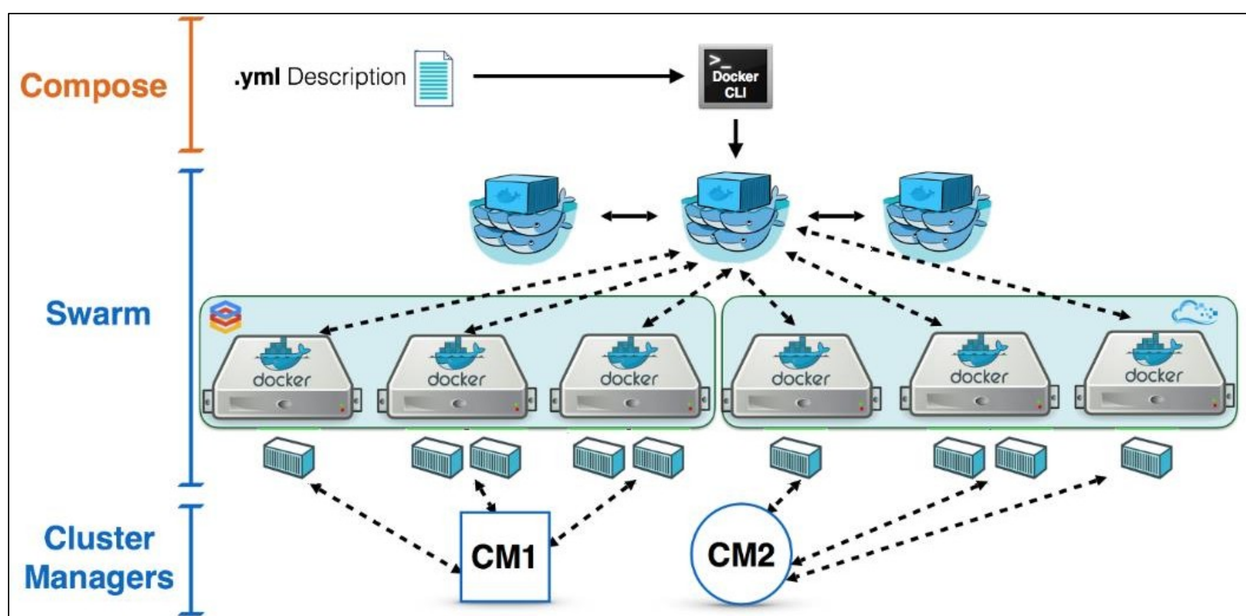

Рисунок 3.5 – Елементи кластера Docker Swarm

Навіть після того, як Docker повністю прийняв Kubernetes як найкращий механізм управління контейнерами, компанія все ще пропонує Swarm як свій

власний інтегрований інструмент управління контейнерами. Він менш масштабований і складний, ніж Kubernetes, що робить його привабливим вибором для ентузіастів Docker, які шукають простий і швидкий спосіб розгортання контейнерів. Фактично, Docker об'єднав Swarm і Kubernetes у своїй корпоративній версії, сподіваючись зробити їх додатковими інструментами.

Давайте розглянемо основні складові архітектури Docker Swarm нижче. Swarm, подібно до кластера в Kubernetes, складається з набору вузлів, які містять принаймні один головний вузол та кілька робочих вузлів, які можуть бути віртуальними або фізичними машинами [15].

Служба визначає завдання, які вузли менеджера або агента повинні виконувати в рої, згідно з визначенням адміністратора рою. Вона вказує, які образи контейнерів повинні використовуватися в рої та які команди потрібно запускати в кожному контейнері. Служба у цьому контексті схожа на мікросервіс; наприклад, тут ви визначаєте конфігураційні параметри для веб-сервера Nginx, який працює в вашому рої, а також визначаєте параметри реплікації у визначенні служби.

Під час розгортання програми у рої вузол менеджера виконує кілька функцій. Він посилає роботу на робочі вузли і керує станом роя. Крім того, вузол менеджера може запускати ті ж самі робочі вузли служб, але їх також можна налаштувати для запуску лише служб, що пов'язані з вузлом менеджера.

Робочі вузли запускають завдання, які розподілені вузлом менеджера у рої. На кожному робочому вузлі працює агент, який повідомляє головному вузлу про стан призначених йому завдань. Таким чином, вузол менеджера може відстежувати служби та завдання, що виконуються у рої.

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

Apache Mesos займає третє місце у списку популярних систем. Вона об'єднує існуючі ресурси в один віртуальний пул, формуючи значні кластери та ефективну

систему управління серверною інфраструктурою. Кожному кластеру надається власний набір ресурсів. Цей проект з відкритим вихідним кодом, трохи старший за Kubernetes, спочатку розроблявся у Каліфорнійському університеті в Берклі. Зараз він широко використовується такими компаніями, як Twitter, Uber та Paypal. Простий інтерфейс Mesos дозволяє легко масштабувати до 10 000 вузлів або більше, дозволяючи середовищам, що працюють на його основі, розвиватися незалежно.

Основні компоненти архітектури Apache Mesos описані нижче. Master daemon – це частина головного вузла, що керує демонами агентів. Використовуючи Apache Zookeeper, можна створити головний кворум Mesos, що складається щонайменше з трьох основних вузлів для забезпечення високої доступності [16].

Агент-демон – це інша складова основного вузла, яка виконує завдання, надіслані платформою. Фреймворк – Mesos не виконує робочі навантаження для програмного оркестрування; замість цього, Marathon отримує ресурси від майстра Mesos (у вигляді пропозицій), і Marathon відправляє завдання на виконавців на основі пропозицій ресурсів, які запускають завдання на агентах. Пропозиція – майстер Mesos збирає інформацію про ЦП та доступність пам'яті на вузлах агента і передає цю інформацію до Marathon, щоб Marathon знала, які ресурси доступні. Завдання – це основні одиниці роботи, які Marathon планує, враховуючи пропозиції ресурсів від майстра Mesos. Завдання виконуються виконавцями на агентських вузлах.

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

3.4 Вивчення пропускної здатності шин передачі даних

На основі наданих даних та порівняльного аналізу у розділі 3.1 вже відомо, що Kafka є відкритою платформою для потокової передачі подій. За своєю суттю, Kafka розроблена як реплікований, розподілений та постійний журнал передачі, який використовується для живлення мікросервісів, керованих подіями, або великомасштабних програм обробки потоків [18]. Клієнти можуть створювати або споживати події безпосередньо в/з проміжного кластера, який постійно зчитує/записує події в базову файлову систему та автоматично реплікує події синхронно або асинхронно в кластері для забезпечення стійкості до помилок та високої доступності. RabbitMQ, як прямий конкурент Kafka, є традиційним програмним забезпеченням для обміну повідомленнями з відкритим вихідним кодом, яке реалізує стандарт обміну повідомленнями AMQP та задовольняє випадки використання черги. RabbitMQ складається з набору брокерських процесів, які розміщують «обмін» для розміщення повідомлень та черг для отримання повідомлень. Доступність та довговічність є характеристиками різних пропонованих типів черг. Класичні черги пропонують найменші гарантії доступності. Класичні дзеркальні черги реплікують повідомлення іншим брокерам та покращують доступність. Більш міцна стійкість забезпечується завдяки нещодавно введеним чергам кворуму, але це може позначитися на продуктивності.

Існує безліч способів порівняти системи в цьому просторі, але всіх найбільше цікавить продуктивність. Чи так Кафка швидка, як раніше, і як вона взаємодіє з іншими системами зараз? Наша мета – перевірити продуктивність Kafka на останній хмарній інфраструктурі. Для порівняння ми взяли традиційного агента обміну повідомленнями RabbitMQ та одного з агентів на базі Apache BookKeeper, Apache Pulsar. Ми акцентували увагу на показниках пропускної здатності та затримки системи, оскільки вони ключові для оцінки продуктивності систем передачі виробничих подій. Тест пропускної здатності визначає, наскільки ефективно система використовує апаратне забезпечення, зокрема диски та центральні процесори. Тест затримки вимірює, наскільки близько система до доставки повідомлень у режимі реального часу, включаючи затримку до p99

процентилю [17], коли 99% запитів мають бути швидшими за задану затримку. Це ключовий показник для веб-додатків та архітектур мікросервісів. Затримки RabbitMQ суттєво збільшуються при пропускній здатності понад 30 МБ/с. Крім того, вплив дзеркального відображення (реплікація вмісту черги) значний при вищій пропускній здатності, а кращі результати затримок можна досягти, використовуючи лише класичну чергу без дзеркального відображення [19] (табл.3.3).

Таблиця 3.3 – Результати експерименту

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

Враховуючи зростаючу популярність потокової обробки та архітектури, що базується на подіях, важливим аспектом систем обміну повідомленнями є затримка, яку повідомлення зазнають під час пересилання від виробника до споживача через систему конвеєру. Ми провели експеримент для порівняння цього показника на трьох системах з найвищою стабільною пропускною здатністю, яку кожна система могла витримати, не виявляючи надмірного навантаження [20].

Для оптимізації затримки ми змінили конфігурацію виробника на всіх трьох системах, налаштувавши його на відправку пакетних повідомлень лише з 1 мс затримкою (у порівнянні з 10 мс, які ми використовували для тестів пропускної здатності), і залишили кожну систему у рекомендованій конфігурації за замовчуванням для забезпечення високої доступності. Кафка була налаштована на використання параметрів fsync за замовчуванням (тобто fsync був вимкнений), а RabbitMQ був налаштований таким чином, щоб не зберігати повідомлення під час дзеркального відображення черги. Ми виявили, що при пропускній здатності понад 30 тис. повідомлень в секунду RabbitMQ стикається з вузькими місцями в ЦП [21].

Виявлено, що Kafka має найкращу пропускну здатність, забезпечуючи майже ста відсотків точності наскрізних затримок, зокрема p99.9 процентіль, як вказано

на графіку у розділі 3.6. У випадку меншої пропускної здатності RabbitMQ доставляє повідомлення з мінімальними затримками. В порівнянні з RabbitMQ, Kafka має найвищу пропускну здатність, записуючи дані у 15 разів швидше.

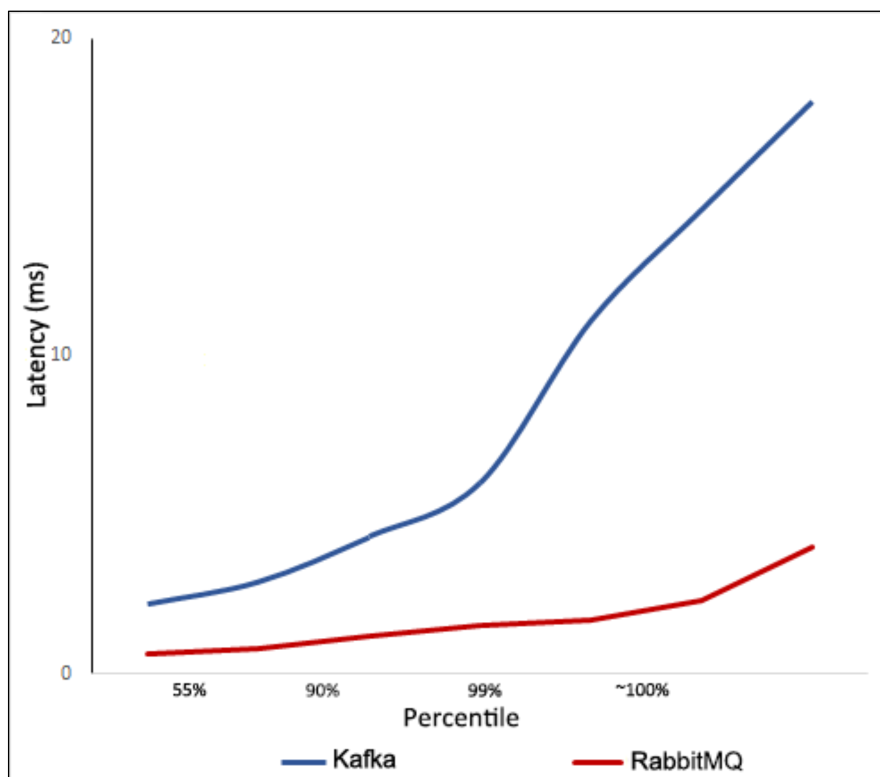

Рисунок 3.6 – Графік, який показує зміну пропускної здатності у відсотковому співвідношенні для систем Kafka та RabbitMQ

Kafka виявляє найменшу затримку при підвищеній пропускній здатності, крім того, забезпечує міцну довготривалу працездатність та високу доступність. З іншого боку, RabbitMQ може мати меншу поперечну затримку в порівнянні з Kafka, проте лише при значно нижчій пропускній здатності.

ВИСНОВКИ

Дослідження методів та інструментів забезпечення безпеки в мікросервісних архітектурах, які використовують технології контейнеризації та оркестрації, надало глибокий інсайт у складні аспекти розробки та експлуатації розподілених систем. Розвиток мікросервісів у поєднанні з контейнеризацією та оркестрацією дозволяє підприємствам забезпечити гнучкість та швидкість у розробці, але це також створює нові виклики у сфері інформаційної безпеки.

У роботі було виконано дослідження методів та інструментів забезпечення безпеки в мікросервісних архітектурах з використанням технологій контейнеризації та оркестрації. Проведено аналіз предметної галузі, що включає вивчення основних концепцій мікросервісної архітектури. Досліджено сучасні технології контейнеризації та оркестрації, що дозволило виявити їхні особливості, можливості та переваги для забезпечення безпеки в мікросервісних архітектурах. Проаналізовано безпекові загрози та виклики, що виникають при використанні мікросервісів. Визначено основні загрози та ризики, з якими стикаються мікросервісні архітектури.

Розроблено методи забезпечення безпеки на рівні контейнерів та мікросервісів, включаючи створення рекомендацій щодо використання інструментів для моніторингу та управління безпекою, а також запропоновано стратегії захисту від потенційних загроз. Проведено експериментальні дослідження з використанням обраних інструментів контейнеризації та оркестрації. Розроблені методи безпеки були випробувані на практичних прикладах, що дозволило оцінити їх ефективність у реальних умовах. Розроблено комплекс практичних рекомендацій щодо впровадження розроблених методів у реальних проектах.

Результати роботи можуть бути використані для підвищення рівня безпеки в інформаційних системах, які базуються на мікросервісній архітектурі, а також служити основою для подальших досліджень у цій галузі.

ПЕРЕЛІК ДЖЕРЕЛ ПОСИЛАННЯ

1. What are Microservices? URL - https://learn.microsoft.com/en-us/azure/architecture/guide/architecture-styles/microservices (дата звернення: 01.04.2024);
2. What are Microservices? URL - https://aws.amazon.com/ru/microservices/#:~:text=Microservices%20are%20an%20architectural%20and,small%2C%20self%2Dcontained%20teams. (дата звернення: 01.04.2024);
3. Container Design Patterns for Distributed Systems URL - https://medium.com/tech-bits/container-design-patterns-7020b132675 (дата звернення: 05.04.2024);
4. Microservice Orchestration Best Practices URL - https://blog.getambassador.io/microservice-orchestration-best-practices-f32314dd6a12 (дата звернення: 05.05.2024);
5. Microservices security: How to protect your architecture URL - https://www.atlassian.com/microservices/cloud-computing/microservices-security#:~:text=Microservices%20security%20safeguards%20each%20small,it%20also%20introduces%20unique%20risks (дата звернення: 07.05.2024);
6. Як захистити кластер. Kubernetes і безпека контейнерів URL - https://dou.ua/forums/topic/36341/ (дата звернення: 07.05.2024);
7. Pattern: Messaging URL – https://microservices.io/patterns/communication-style/messaging.html (дата звернення: 08.05.2024);
8. RESTful Web Services / Leonard Richardson, Sam Ruby, Released May, 2007 – O'Reilly Media Inc., 30 p.;
9. What is RabbitMQ. URL - https://www.cloudamqp.com/blog/2015-05-18-part1-rabbitmq-forbeginners-what-is-rabbitmq.html, (дата звернення: 09.05.2024);
10. Optimizing Kafka Performance. URL – https://granulate.io/optimizing-kafka-performance/, (дата звернення: 10.05.2024);
11. Docker Wikipedia. URL – https://uk.wikipedia.org/wiki/Docker, (дата звернення: 10.05.2024);

12. Docker, Run swarm and kubernetes interchangeably. your choice of swarm or kubernetes for flexible and powerful orchestration options. URL – https://www.docker.com/products/orchestration, (дата звернення: 10.05.2024);

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

14. Connecting Applications with Services URL – https://kubernetes.io/docs/concepts/services-networking/connect-applications-service, (дата звернення: 11.05.2024);

15. Swarm mode overview. URL – https://docs.docker.com/engine/swarm/, (дата звернення: 11.05.2024);

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

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

18. Як обчислюється процентиль. URL – https://uk.economy-pedia.com/11039689-percentile, (дата звернення: 12.05.2024);

19. Kubernetes Documentation learning environment. URL – https://kubernetes.io/docs/setup/, (дата звернення: 12.05.2024);

20. Filatov V. O., Yerokhin A. L., Zolotukhin O. V., Kudryavtseva M. S. Hybrid simulation models for complex decision-making problems with partial uncertainty. Information Extraction and Processing. 2022, 50(126), 78-86. DOI:https://doi.org/10.15407/vidbir2022.50.078;

21. Dmytro Panchenko, Daniil Maksymenko, Olena Turuta, Andriy Yerokhin, Yana Daniiel, Oleksii Turuta . Evaluation and Analysis of the NLP Model Zoo for Ukrainian Text Classification // Communications in Computer and Information Science, 2022, 1698 CCIS, pp. 109–123. DOI: 10.1007/978-3-031-20834-8_6.

ПЕРЕЛІК ДЖЕРЕЛ ПОСИЛАННЯ ЗА НАУКОВИМИ НАПРЯМАМИ КЕРІВНИКА ТА НАУКОВЦІВ КАФЕДРИ ПРОГРАМНОЇ ІНЖЕНЕРІЇ

20. Filatov V. O., Yerokhin A. L., Zolotukhin O. V., Kudryavtseva M. S. Hybrid simulation models for complex decision-making problems with partial uncertainty. Information Extraction and Processing. 2022, 50(126), 78-86. DOI:https://doi.org/10.15407/vidbir2022.50.078;

21. Dmytro Panchenko, Daniil Maksymenko, Olena Turuta, Andriy Yerokhin, Yana Daniiel, Oleksii Turuta . Evaluation and Analysis of the NLP Model Zoo for Ukrainian Text Classification // Communications in Computer and Information Science, 2022, 1698 CCIS, pp. 109–123. DOI: 10.1007/978-3-031-20834-8_6.