Міністерство освіти і науки України
Харківський національний університет радіоелектроніки

Факультет комп'ютерної інженерії та управління
(повна назва)

Кафедра електронних обчислювальних машин
(повна назва)

КВАЛІФІКАЦІЙНА РОБОТА

Пояснювальна записка

Рівень вищої освіти перший (бакалаврський)

Метод забезпечення надійності контейнеризованого додатка в розподіленому середовищі Docker

(тема)

Виконав:
здобувач 4 року навчання,
групи КІУКІ-21-5
Богдан РОМАНЕНКО
(власне ім'я, прізвище)

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

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

Освітня програма
Комп'ютерна інженерія
(повна назва освітньої програми)

Керівник: ас. Ірина ЧЕПУРНА
(посада, власне ім'я, прізвище)

Допускається до захисту

Завідувач кафедри ЕОМ _____ Андрій КОВАЛЕНКО
(підпис) (власне ім'я, прізвище)

2025 р.

Харківський національний університет радіоелектроніки

Факультет комп'ютерної інженерія та управління

Кафедра електронних обчислювальних машин

Рівень вищої освіти перший (бакалаврський)

Спеціальність 123 «Комп'ютерна інженерія»
(код і повна назва)

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

Освітня програма Комп'ютерна інженерія
(повна назва)

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

ЗАВДАННЯ

НА КВАЛІФІКАЦІЙНУ РОБОТУ

здобувачеві Романенку Богдану Володомировичу
(прізвище, ім'я, по батькові)

1. Тема роботи Метод забезпечення надійності контейнеризованого додатку в розподіленому середовищі Docker

затверджена наказом по університету від " 26 " травня 2025 р. № 424 Ст

2. Термін подання здобувачем роботи до екзаменаційної комісії 17 червня 2025 р.

3. Вхідні дані до роботи 1) технології для розгортання додатку: віртуалізація контейнеризація; 2) інструменти оркестрації контейнерів: Docker Compose, Kubernetes, Docker Swarm; 3) методи забезпечення надійності: масштабування, балансування навантаження, кешування; 4) методи експериментального та аналітичного моделювання роботи вебдодатку.

4. Перелік питань, що потрібно опрацювати у роботі
1) аналіз проблем та існуючих рішень;
2) типи розподілених систем для розгортання вебдодатків;
3) інструменти оркестрації в Docker;
4) методи забезпечення надійності вебдодатків;
5) побудова сегменту контейнеризованого вебдодатку на базі Docker;
6) висновки.

5. Перелік графічного матеріалу із зазначенням креслеників, схем, плакатів, комп'ютерних ілюстрацій

Слайд-презентація – 12 слайдів

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

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

КАЛЕНДАРНИЙ ПЛАН

№	Назва етапів роботи	Строк / терміни виконання етапів роботи	Примітка
1	Аналіз проблеми та огляд існуючих рішень	27.05.25-30.05.25	
2	Вибір технології розробки та засобів реалізації	31.05.25-03.06.25	
3	Розробка та тестування методу	04.06.25-06.06.25	
4	Оформлення матеріалів кваліфікаційної роботи	07.06.25-10.06.25	
5	Подання кваліфікаційної роботи керівникові	11.06.25-12.06.25	
6	Підготовка презентації та попередній захист	12.06.2025 – 13.06.2025	
7	Подання кваліфікаційної роботи на рецензування	14.06.25-16.06.25	

Дата видачі завдання " 26 " травня 2025 р.

Здобувач _____
(підпис)

Керівник роботи _____ ас. Ірина ЧЕПУРНА
(підпис) (посада, власне ім'я, прізвище)

РЕФЕРАТ

Пояснювальна записка кваліфікаційної роботи: 64 с., 19 рис., 1 табл., 4 дод., 31 джерело.

КОМП’ЮТЕРНА МЕРЕЖА, DOCKER, ВЕБДОДАТОК, ОРКЕСТРАЦІЯ, КОНТЕЙНЕР, ВІДМОВОСТІЙКІСТЬ.

Метою кваліфікаційної роботи є розробка методу забезпечення надійності контейнеризованого додатку в середовищі Docker, що забезпечує стабільну роботу за умов обмежених апаратних ресурсів.

В ході виконання роботи досліджено технології оркестрації контейнерів, порівняно різні архітектурні підходи до масштабування і відмовостійкості, а також реалізовано кластерну систему на базі Docker Swarm. Обґрунтовано доцільність використання мікросервісної моделі для забезпечення гнучкості та ізоляції компонентів. Проведено моделювання типових сценаріїв збоїв і тестування продуктивності. Підтверджено, що обране рішення дозволяє підтримувати стабільну роботу додатку в умовах змінного навантаження та відповідає вимогам до сучасних розподілених систем.

ABSTRACT

Bachelor`s thesis: 64 pages, 19 figures, 1 table, 4 appendices, 31 sources.

COMPUTER NETWORK, DOCKER, WEB APPLICATION, ORCHESTRATION, CONTAINER, FAULT TOLERANCE.

The major goal of this thesis is to develop a method for ensuring the reliability of a containerized application in a Docker-based environment under limited hardware conditions.

In order to achieve this, a container-based microservice architecture was designed and deployed using Docker Swarm. The study includes a comparative analysis of orchestration tools and architectural strategies for fault tolerance and scalability. A distributed cluster was configured with automatic service replication, load balancing, overlay networking, and restart policies. Various failure scenarios and load simulations were conducted to evaluate system behavior. The final results confirmed the system’s ability to recover from failures and adapt to workload changes, meeting the reliability requirements of modern distributed environments.

ЗМІСТ

СКОРОЧЕННЯ ТА УМОВНІ ПОЗНАКИ

AKS – Azure Kubernetes Service, сервіс Kubernetes від Microsoft Azure (англ. Azure Kubernetes Service)

API – інтерфейс прикладного програмування (англ. Application Programming Interface)

AWS –хмарні сервіси компанії Amazon (англ. Amazon Web Services)

CI/CD – безперервна інтеграція і доставка (англ. Continuous Integration / Continuous Delivery)

CLI – інтерфейс командного рядка (англ. Command-Line Interface)

CPU – центральний процесор (англ. Central Processing Unit)

DB – база даних (англ. Database)

DNS – система доменних імен (англ. Domain Name System)

ECS – сервіс оркестрації контейнерів Amazon (англ. Elastic Container Service)

EKS – сервіс Kubernetes від Amazon (англ. Elastic Kubernetes Service)

GCP – хмарна платформа компанії Google (англ. Google Cloud Platform)

GKE – сервіс Kubernetes від Google (англ. Google Kubernetes Engine)

HTTP – протокол передачі гіпертексту (англ. HyperText Transfer Protocol)

IaaS – інфраструктура як сервіс (англ. Infrastructure as a Service)

MVP – мінімально життєздатний продукт (англ. Minimum Viable Product)

P2P – однорангова мережа (англ. Peer-to-Peer)

PaaS – платформа як сервіс (англ. Platform as a Service)

RAM – оперативна пам’ять (англ. Random Access Memory)

RPS – запитів на секунду (англ. Requests Per Second)

SaaS – програмне забезпечення як сервіс (англ. Software as a Service)

URL – уніфікований локатор ресурсу (англ. Uniform Resource Locator)

YAML – формат розмітки даних (англ. Yet Another Markup Language)

ВСТУП

Сучасні інформаційні системи функціонують в умовах високої інтенсивності обробки даних і постійно зростаючих вимог до масштабованості, що зумовлює підвищений попит на розподілені обчислювальні середовища та впровадження мікросервісної архітектури як ефективного підходу до розробки веборієнтованих програмних рішень.

В таких умовах питання відмовостійкості та надійного функціонування мережної інфраструктури, що забезпечує розгортання та експлуатацію вебсервісів набуває особливої актуальності.

Використання хмарних розподілених систем та технологій віртуалізації забезпечує розгортання масштабованої IT-інфраструктури, зокрема з використанням мікросервісної архітектури, проте такі рішення мають свої недоліки, що актуалізує необхідність досліджень в пошуку ефективних підходів, особливо в умовах обмежених ресурсів.

Впровадження технологій віртуалізації та контейнеризації створює передумови для побудови мікросервісної архітектури з урахуванням вимог до високої продуктивності при обмеженні апаратних ресурсів, а застосування інструментів автоматизації та оркестрації сприяє організації масштабованої та гнучкої інфраструктури.

Метою кваліфікаційної роботи є розробка методу забезпечення надійності контейнеризованого додатку на базі розподіленого середовища Docker, що дозволяє гарантувати надійну та безперервну роботу додатку в умовах обмежених ресурсів та вимог до масштабованості та надійності при динамічній зміні навантаження.

1 АНАЛІЗ ПРЕДМЕТНОЇ ОБЛАСТІ

1.1 Характеристики розподілених систем

Стрімке зростання обсягів даних та широке впровадження вебдодатків як інструментів комунікації та надання цифрових послуг зумовлюють необхідність розробки нових підходів до побудови IT-інфраструктури. В умовах постійно зростаючих вимог до продуктивності та гнучкості систем, особливої актуальності набуває концепція розподілених обчислювальних систем, здатних забезпечувати динамічне масштабування, ефективне управління обчислювальними ресурсами та стійкість до динамічно змінюваного навантаження. Одним з ефективних підходів до побудови високопродуктивних вебдодатків є впровадження мікросервісної архітектури, що характеризується високим рівнем адаптивності, масштабованості, простотою розгортання та експлуатації.

Розподілені системи формують основу для розгортання сучасних додатків, особливо в межах мікросервісної архітектури, де кожен сервіс виконує окрему функцію та може масштабуватися незалежно [1]. Такі системи складаються з обчислювальних ресурсів, сховищ даних і сервісів, які функціонують на декількох взаємодіючих вузлах у мережі. Розподілені системи призначені для забезпечення неперервної роботи інфраструктури, в тому числі інфраструктури для розгортання та експлуатації додатків з мікросервісною архітектурою в умовах високих вимог до відмовостійкості та стабільності.

Основними перевагами розподілених систем є такі характеристики:

- надійність та стійкість до збоїв, що досягається завдяки використанню реплікації та механізмів автоматичного відновлення. Стійкість системи полягає в здатності продовжувати функціонування за умов затримок, втрат з’єднання або часткової недоступності компонентів. В роботі [2]

зазначається, що стійкість до збоїв є основним чинником для розробки розподілених систем та ключовим фактором для розгортання надійної інфраструктури. Реплікація передбачає збереження копій даних на кількох вузлах системи, що дозволяє уникнути втрати інформації у випадку відмови одного з них. В роботі [3] автори акцентують увагу на тому, що основною метою реплікації даних є забезпечення підвищеної продуктивності для програм з високими вимогами до обробки даних. Це досягається шляхом усунення обмежень, пов’язаних з доступністю, надійністю, безпекою, пропускною здатністю та часом відгуку при доступі до даних. Механізми автоматичного відновлення включають в себе виявлення збоїв в режимі реального часу, повторне запускання пошкоджених сервісів, а також динамічне перенесення навантаження на доступні компоненти системи, що зменшує час простою та забезпечує високу доступність сервісів;

- гнучке масштабування, яке забезпечує адаптацію до змін у навантаженні. Масштабування дозволяє розподіляти ресурси відповідно до поточних вимог системи, і буває двох типів:

1. Вертикальне масштабування, яке полягає в збільшенні обчислювальних ресурсів окремого вузла, що дозволяє покращити продуктивність одного елемента системи без зміни архітектури. В роботі [4] автори відзначають, що для запущених сервісів з невисокими обчислювальними вимогами, що розміщуються на вузлах з обмеженими ресурсами, вертикальне масштабування в середовищі Azure часто є економічно доцільнішим порівняно з горизонтальним масштабуванням. Перевагами такого масштабування є простота реалізації та збереження поточної структури системи. До обмежень вертикального масштабування відносять обмеженість апаратних ресурсів, зростання вартості з підвищенням рівня продуктивності та обмежену відмово стійкість, оскільки в разі відмови вузла втрачається функціональність всієї системи.

2. Горизонтальне масштабування, яке передбачає додавання нових вузлів до кластера, дозволяє рівномірно розподіляти навантаження між

більшим числом серверів або контейнерів, покращуючи відмовостійкість та розширюючи обчислювальні можливості системи. Автори роботи [5] зазначають, що горизонтальне автоматичне масштабування подів в Kubernetes динамічно змінює їх кількість відповідно до рівня використання пам’яті на вузлах, що сприяє ефективнішому розподілу навантаження та оптимальному використанню ресурсів кластера. Переваги такого масштабування є потенційно необмежене масштабування, покращена відмовостійкість, оскільки при виході з ладу одного вузла, інші продовжують обробку, а також можливість автоматизації масштабування за допомогою інструментів, зокрема Kubernetes, Docker Swarm;

- синхронізація та цілісність даних, що забезпечуються узгодженою роботою вузлів інфраструктури з метою підтримки безперервної та стабільної роботи системи. В роботі [6] зазначено, що через вимоги до узгодженості даних можуть виникати тимчасові невідповідності, що ускладнює забезпечення цілісності транзакцій та проведення аналітики в режимі реального часу. Окрім цього, під час синхронізації конфіденційної інформації між кількома постачальниками хмарних послуг з’являються ризики безпеки, оскільки різні стандарти шифрування та механізми контролю доступу можуть створювати потенційні вразливості. Ці характеристики визначають надійність, достовірність та послідовність інформації в розподіленому середовищі, де дані можуть зберігатися та оброблятися на багатьох вузлах одночасно. Цілісність означає, що дані залишаються точними, узгодженими та неушкодженими протягом усього життєвого циклу, незалежно від кількості копій чи вузлів. Це передбачає захист від часткових оновлень, некоректної реплікації, колізій запису та несанкціонованого доступу.

Синхронізація забезпечує єдину актуальну версію даних між всіма вузлами, які беруть участь в обробці. В розподілених системах синхронізація є впливовим фактором продуктивності, оскільки в несинхронізованих системах виникають затримки через асинхронну реплікацію та паралельну

обробку даних. Існує два типи синхронізації:

- синхронна, коли зміни негайно поширюються на всі вузли. Це забезпечує високу узгодженість, але може знижувати продуктивність через очікування підтвердження від інших вузлів;

- асинхронна, яка дозволяє виконувати обробку без очікування підтвердження від усіх вузлів. Перевагами асинхронної синхронізації є підвищена продуктивність, але недоліком є можливі тимчасові розбіжності у версіях даних.

Узгодженість полягає в тому, що всі вузли використовують дані в один й той самий момент часу або в межах допустимих відхилень.

Забезпечення цілісності та синхронізації є критично важливим для систем, які працюють в режимі реального часу та для додатків, які оброблюють фінансові транзакції. Це дозволяє забезпечити безперервну, стабільну та надійну роботу мережної інфраструктури;

- адаптивність, яка досягається шляхом застосування автоматизованого керування, логування, оновлення, а також інтеграції з процесами CI/CD. Адаптивність визначається здатністю системи автоматично або з мінімальним втручанням адаптуватися до змін в навантаженні, конфігурації, середовищі виконання чи вимогах користувача. Вона забезпечує гнучкість, стійкість до збоїв та оперативне реагування на зміну зовнішніх або внутрішніх умов. В роботі [7] зазначено, що сучасні сервіси та програмні рішення повинні бути адаптованими до змін в середовищі розгортання, зокрема до таких параметрів, як географічне розташування, обсяг використаної пам’яті або кількості активних користувачів, з метою покращення користувацького досвіду. Реалізація адаптивності потребує ефективної системи постійного моніторингу інфраструктури, що забезпечує відповідний рівень функціональності та якості. Для цього в розподілених системах активно використовуються інструменти оркестрації, які централізовано керують життєвим циклом сервісів, контейнерів або мікросервісів. Інструменти оркестрації, зокрема

Kubernetes, Docker Swarm, застосовують для автоматичного масштабування, перезапуску сервісів в разі збоїв, розгортання нових версій сервісів без простою, а також балансування навантаження між вузлами або контейнерами. Адаптивність передбачає постійне спостереження за станом системи та її поведінкою з метою оперативного виявлення аномалій та потенційних загроз, можливості автоматизованого оновлення програмного забезпечення. Інтеграція з CI/CD процесами забезпечує швидке впровадження змін у коді та автоматичне тестування та збірку. Перевагами адаптивності розподілених систем є зменшення часу реакції на зміну вимог або умов, підвищення надійності через оперативне виявлення та локалізацію збоїв, зменшення витрат на адміністрування та підтримку;

- ресурсна ефективність, що дозволяє розгортати інфраструктуру з високою щільністю сервісів в умовах обмежених ресурсів, завдяки застосуванню оптимізаційних підходів до використання обчислювальних потужностей. В роботі [8] зазначено, що оптимізація використання ресурсів є ключовим фактором досягнення максимальної ефективності та продуктивності сучасних розподілених систем, що полягає в здатності максимально раціонально використовувати наявні мережні та серверні ресурси для зберігання та забезпечення стабільної продуктивної роботи без зайвих витрат чи простоїв. Такі оптимізаційні підходи дозволяють знижувати загальне енергоспоживання та вартість експлуатації додатків;

- інфраструктурна гнучкість, яка проявляється у швидкому розгортанні, масштабуванні та експлуатації додатків в хмарних або гібридних середовищах, що дозволяє адаптувати архітектуру та ресурси до змін у вимогах до навантаження та середовища виконання з мінімальними витратами часу та ресурсів. В роботі [9] зазначається, що ізоляція, абстрагування та оптимізація апаратних ресурсів, зокрема процесору, сховища, пам'яті та мережі, а також використання технологій віртуалізації дозволяє організаціям досягати більшої ефективності та гнучкості. Віртуалізація, за своєю суттю, дозволяє розподіляти ресурси сервера на

кількох віртуальних машинах або гостьових системах, що сприяє значному покращенню консолідації серверів, динамічному виділенню ресурсів та зниженню накладних витрат в умовах обмеженості апаратних ресурсів. Завдяки використанню хмарних або контейнеризованих середовищ, інфраструктура може динамічно масштабуватись, вертикально або горизонтально, відповідно до навантаження; використовувати хмарні ресурси на вимогу та автоматично реагувати на пікові навантаження або зниження потреб у ресурсах.

1.2 Типи розподілених систем

Для розгортання мікросервісних додатків у розподіленому середовищі широко застосовуються технології віртуалізації та контейнеризації.

Хмарні платформи, зокрема Amazon Web Services (AWS), Microsoft Azure, Google Cloud Platform (GCP), вважаються одними з перспективних та популярних інструментів для реалізації масштабованих та доступних розподілених середовищ. Вони базуються на великій кількості розподілених серверів, які взаємодіють через мережу. Хмарні платформи надають доступ до обчислювальних потужностей, зберігання даних, сервісів та платформ, що фізично можуть бути розміщені в різних географічних локаціях, але працюють як єдина система. Вони забезпечують широкі можливості для розгортання додатків, в тому числі з використанням мікросервісної архітектури, зокрема надають гнучкий доступ до обчислювальних ресурсів, інструментам автоматизації, аналітики та технологіям штучного інтелекту. Використання моделей хмарних обчислень, зокрема програмного забезпечення як послуги (SaaS), платформи як послуги (PaaS) та інфраструктури як послуги (IaaS) сприяє оптимізації витрат на IT-інфраструктуру шляхом переходу до моделі споживання ресурсів за потреби. Такий підхід дозволяє відмовитися від капітальних витрат на придбання та обслуговування апаратного та програмного забезпечення, замінивши їх на

операційні витрати, що формуються відповідно до фактичного обсягу використання сервісів. Це дозволяє масштабувати інфраструктуру відповідно до потреб, що стає перевагами для малого та середнього бізнесу. На противагу хмарним платформам традиційні підходи із застосуванням апаратних серверних ресурсів вимагають значних інвестицій в обладнання, програмне забезпечення та технічне обслуговування. В роботі [10] зазначається, що хмарні платформи пропонують основні функції, що полягають в наданні обчислювальних потужностей, гнучкості та масштабованому доступ. Підприємства та організації звертаються до хмарних рішень і послуг, оскільки вони надають такі переваги, як зниження витрат і підвищення ефективності. Хмарні платформи також надають широкий спектр інструментів для розгортання та оркестрації, автоматизації та моніторингу додатків та мережної інфраструктури, зокрема Elastic Container Service (ECS), Elastic Kubernetes Service (EKS) та Lambda для серверного виконання коду від AWS; Google Kubernetes Engine (GKE) для управління Kubernetes-кластерами від GCP; Azure Kubernetes Service (AKS) та Azure App Service для розгортання веб-додатків від Microsoft Azure.

Проте, незважаючи на численні переваги, хмарні платформи мають свої недоліки. Залежність від стороннього постачальника послуг може призвести до ризиків, пов’язаних з безпекою та конфіденційністю даних, особливо у випадку з чутливою інформацією, що зазначено в роботі [11]. Незважаючи на те, що хмари надають високу доступність, існують потенційні проблеми з затримками в мережі та обмеженнями в пропускній здатності, що можуть негативно вплинути на продуктивність критичних додатків. Постійне зростання витрат на послуги хмарних провайдерів теж стає серйозною проблемою для довгострокових проєктів, де висуваються вимоги до контролю та розподілу накладних витрат на інфраструктуру.

Створення розподілених середовищ для розгортання додатків передбачає використання технологій віртуалізації, зокрема на основі гіпервізорів та контейнерів. Гіпервізори, зокрема VMware ESXi, Hyper-V,

KVM, широко застосовуються для розгортання віртуалізованих середовищ завдяки своїй здатності повністю ізолювати операційні системи та забезпечувати високу стабільність. В роботі [12] зазначено, що незважаючи на те, що гіпервізор вважається інструментом віртуалізації, він дозволяє ефективно використовувати ресурси фізичних серверів для побудови масштабованої інфраструктури. Вони забезпечують можливість запуску незалежних віртуальних машин з виділеними ресурсами, що дає змогу розгортати інфраструктуру з підвищеними вимогами до безпеки та сумісності. Для оркестрації та автоматизованого розгортання кластерів віртуальних машин використовують систему Kubernetes, яка сумісна з Unix-подібними операційними системами та підтримує інтеграцію з контейнеризованими середовищами. Проте такі системи мають значні витрати на ресурси, повільніший час розгортання та складність масштабування.

Контейнеризація як одна з форм віртуалізації, яка пропонує інструменти для більш легкого та гнучкого розгортання рішень на базі віртуалізації в порівнянні з гіпервізорами. За допомогою платформ віртуалізації з підтримкою контейнеризації, зокрема Docker або Proxmox, додатки розгортаються в ізольованих контейнерах з мінімальним споживанням ресурсів. Контейнери забезпечують швидке розгортання, простоту переносу між середовищами, наприклад, з локальної машини до хмари, та легку інтеграцію з DevOps-процесами в порівнянні з віртуальними машинами, що зазначено в роботах [13, 14]. На рисунку 1.1 наведено порівняння архітектури віртуальних машин та контейнерів для розгортання інфраструктури додатку.

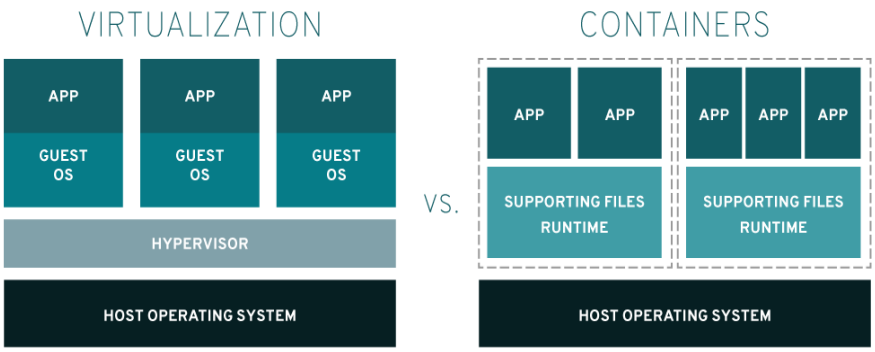

Рисунок 1.1 – Порівняння архітектури віртуальних машин та контейнеризації

Застосування контейнеризації ефективно для розгортання сервісів з мікросервісною архітектурою, де важливо забезпечити гнучкість та масштабованість окремих компонентів системи. Для керування великою кількістю контейнерів використовуються інструменти оркестрації, зокрема Kubernetes, Docker Swarm або OpenShift. Вони дозволяють автоматизувати масштабування, оновлення, балансування навантаження та моніторинг, що значно підвищує ефективність керування інфраструктурою (рисунок 1.2).

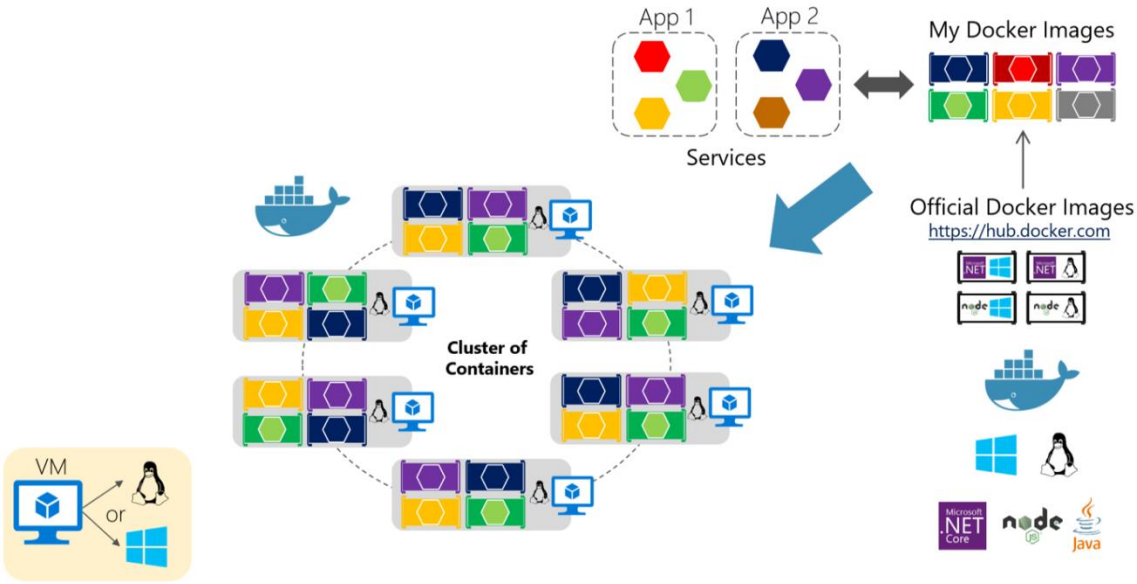

Рисунок 1.2 – Приклад кластеризованої розподіленої інфраструктури Docker-контейнерів

Наразі використовують такі основні типи розподілених систем, що

можуть бути застосовані для розгортання додатків, включно з мікросервісною архітектурою:

1. Клієнт-серверні системи, які є основною архітектурною моделлю, де клієнти здійснюють запити на отримання ресурсів або сервісів, що надаються одним або кількома серверами. Така архітектура найчастіше застосовується для вебдодатків, REST API та баз даних. Зокрема хмарні сервіси PaaS теж базуються на цій моделі (рисунок 1.3).

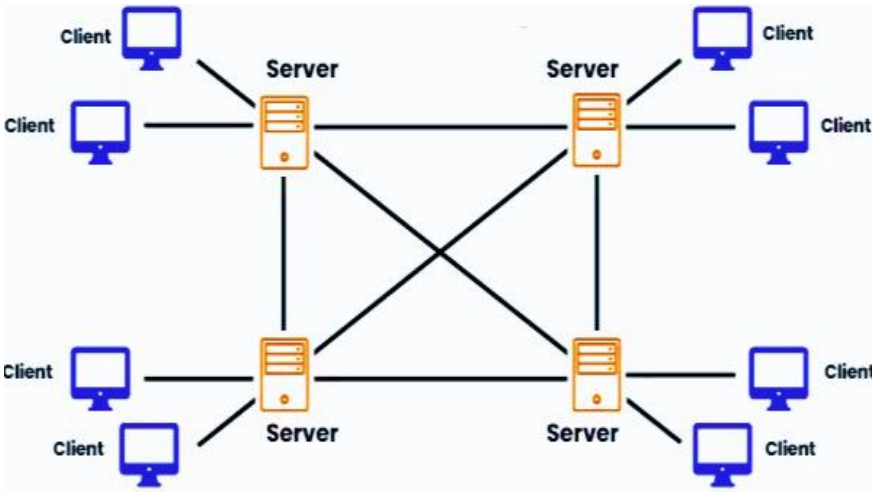

Рисунок 1.3 – Схема клієнт-серверної архітектури в розподілених системах

2. Рівноправні (P2P) системи, в яких всі вузли мають рівні права та можуть одночасно виступати як клієнтами, так і серверами. P2P-архітектура використовується в мережах обміну файлами, блокчейн-системах та деяких типах мікросервісів (рисунок 1.4).

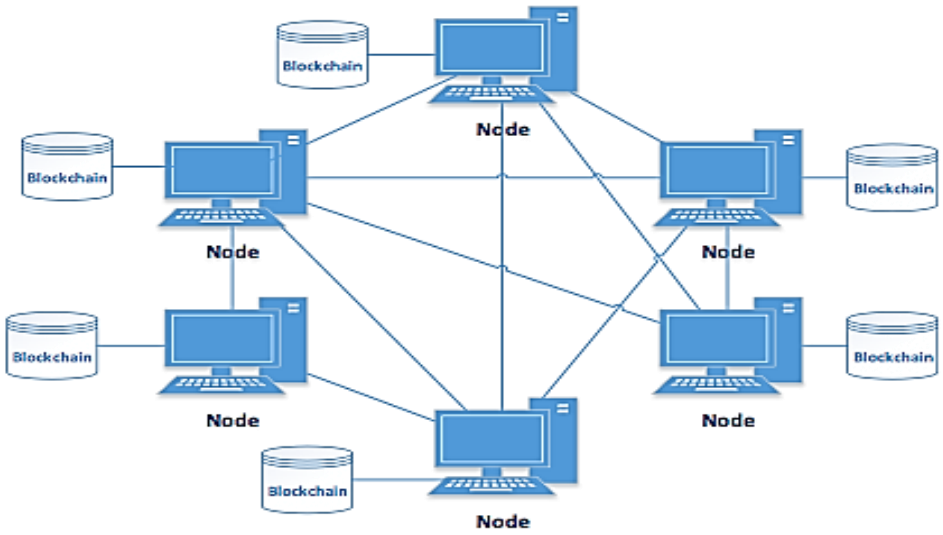

Рисунок 1.4 – Схема P2P –мережі для блокчейну

3. Кластерні системи, які об'єднують декілька вузлів в єдину логічну систему, що працює як одне ціле (рисунок 1.5). Кластери використовуються для підвищення продуктивності, надійності та масштабованості. Керування кластерними системами виконується за допомогою інструмента Kubernetes або інтегрованих інструментів автоматизації та оркестрації на платформах віртуалізації.

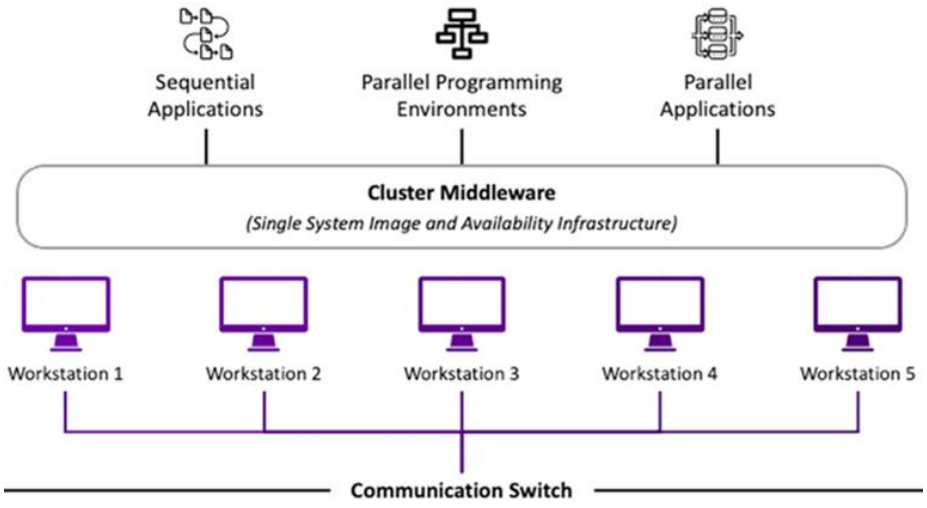

Рисунок 1.5 – Архітектура кластерної системи

4. Хмарні розподілені системи, які забезпечують динамічне масштабування ресурсів, високу доступність та географічно розподілену інфраструктуру. Вони базуються на технологіях віртуалізації та

контейнеризації, що дозволяє швидко розгортати та керувати мікросервісними додатками. Найвідомішими хмарними вендорами є AWS, GCP та Microsoft Azure (рисунок 1.6).

5. Гібридні системи, які в своїй архітектурі поєднують декілька типів архітектур, наприклад інфраструктура компанії складається з декількох сегментів, які частиною розташовані на хмарній платформі, а інший сегмент базується на локальному кластері. Гібридні системи використовують в компаніях з високими вимогами до безпеки.

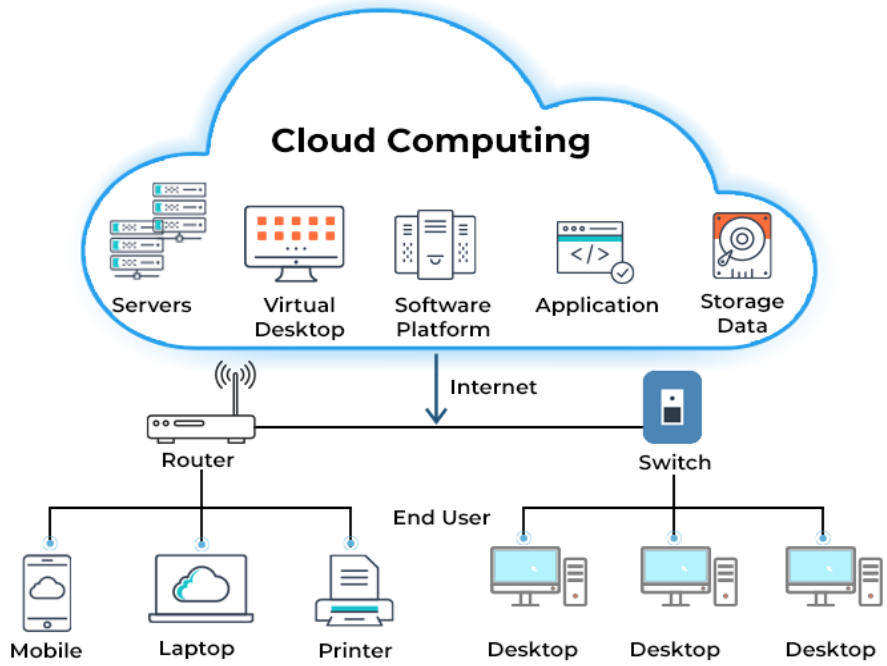

Рисунок 1.6 – Архітектура хмарної розподіленої системи

1.3 Постановка задачі

В сучасному цифровому середовищі, на тлі стрімкого зростання обсягів даних та зростаючих вимог до забезпечення стабільної та безперервної роботи вебдодатків, спостерігається підвищений попит на розподілені обчислювальні середовища, які забезпечують умови для ефективного розгортання вебзастосунків. Розподілені системи також сприяють впровадженню мікросервісної архітектури, що дозволяє підвищити надійність, відмовостійкість і безперервність функціонування вебсервісів.

Проте особливої актуальності набувають питання масштабування та підтримки стабільної роботи вебзастосунків в умовах динамічної зміни навантаження. Використання платформ контейнеризації, зокрема Docker, дозволяє розгортати вебдодатки в ізольованих контейнерах, використовуючи вбудовані інструменти для масштабування та оркестрації у відповідь на зростання навантаження в умовах обмежених апаратних ресурсів.

Метою даної кваліфікаційної роботи є розробка методу забезпечення надійності контейнеризованого додатку в середовищі Docker. Запропонований метод дозволяє забезпечити відмовостійкість, стабільну та безперервну роботу додатку в умовах динамічної зміни навантаження в режимі реального часу.

2 ТЕХНОЛОГІЇ ТА МЕТОДИ ЗАБЕЗПЕЧЕННЯ НАДІЙНОЇ РОБОТИ ДОДАТКІВ

2.1 Технології віртуалізації для розгортання додатків

Сучасні рішення проблем продуктивності та гнучкості вебдодатків все частіше базуються на застосуванні віртуалізації та контейнеризації, які дозволяють створювати масштабовану інфраструктуру з ефективним розподілом обчислювальних ресурсів [15]. При розробці розподілених додатків важливо враховувати особливості архітектури додатку, що забезпечує вибір середовища для розгортання.

Використання гіпервізорів відкриває можливість запуску ізольованих віртуальних середовищ з власними операційними системами, що покращує управління, безпеку та адаптивність сервісів. Віртуалізація на основі гіпервізорів підтримує масштабування, автоматизацію управління віртуальними машинами, зокрема через Kubernetes, та розгортання додаткових сервісів, необхідних для стабільної роботи вебдодатків.

Водночас віртуалізація на базі гіпервізорів характеризується суттєвим споживанням апаратних ресурсів та підвищеною складністю адміністрування. В цьому контексті контейнеризація виступає кращим рішенням, що забезпечує ізоляцію середовища, стабільність роботи сервісів та можливість гнучкого масштабування відповідно до змін у навантаженні. Вона дозволяє запускати окремі компоненти додатку в легких, автономних контейнерах, які спільно використовують ядро операційної системи, що значно знижує накладні витрати в порівнянні з традиційними віртуальними машинами. Завдяки цьому контейнеризація сприяє більш швидкому розгортанню, оновленню та масштабуванню сервісів, що важливо для вебдодатків з динамічно змінним навантаженням (рисунок 2.1).

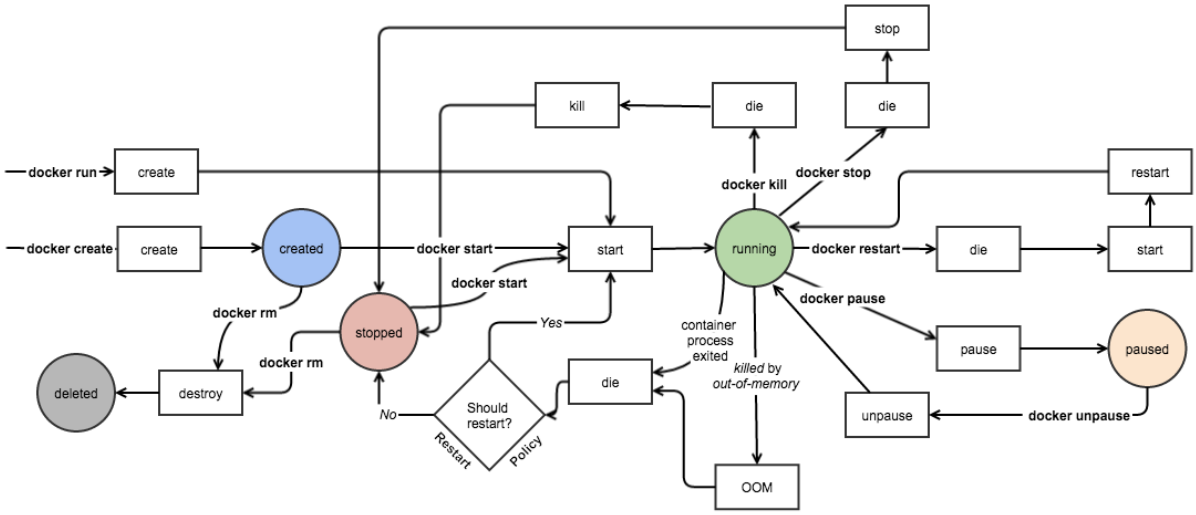

Рисунок 2.1 – Життєвий цикл контейнера в Docker з урахуванням перезапусків та обробки збоїв

На відміну від віртуальних машин, контейнери не потребують окремої гостьової ОС, що зменшує їх ресурсоємність та дозволяє запускати більше контейнерів на одному хості. Вони забезпечують високу портативність між середовищами, ізоляцію на рівні процесів та більш уніфікований процес розробки.

Платформи для контейнеризації, зокрема Docker та Podman, дозволяють розгортати незалежні сервіси в межах однієї ОС з невеликими витратами ресурсів, а також включають інструменти для масштабування та оркестрації. Контейнеризація відзначається кількома ключовими перевагами, що сприяють ефективному використанню ресурсів та безперервності роботи. По-перше, це динамічне масштабування кількості контейнерів відповідно до навантаження. По-друге, висока портативність контейнерів, що забезпечує сумісність з різними середовищами та полегшує міграцію додатків між локальними та хмарними платформами; а також оптимальним управлінням обчислювальними ресурсами завдяки використанню ядра ОС хоста. Це дозволяє розміщувати більше сервісів в порівняні з віртуальними машинами.

Docker надає стандартизовані образи контейнерів, які включають

необхідний набір компонентів для запуску сервісів, включно з вихідним кодом, бібліотеками та конфігураціями. Це забезпечує високу портативність та незалежність від інфраструктури, а також широкий спектр засобів для оркестрації та автоматизації, що суттєво прискорює процеси розробки, тестування та експлуатації. Завдяки простоті адміністрування та великій кількості доступних образів Docker є однією з найпопулярніших платформ для створення масштабованих та надійних вебдодатків.

Podman, як альтернатива Docker, підтримує ті ж формати образів та інтегрується з системами автоматизації, зокрема Systemd. Водночас його головною відмінністю є відсутність необхідності запускати демон, що підвищує рівень безпеки та спрощує інтеграцію в середовища з жорсткими вимогами контролю доступу. Такий підхід допомагає знизити потенційні ризики безпеки в порівнянні з Docker [16].

Контейнеризація передбачає запуск додатків в ізольованих середовищах, які спільно використовують ядро операційної системи, але мають окремі файлові системи, мережні стеки та змінні середовища. Це відрізняє їх від класичних віртуальних машин, де кожна машина вимагає власної операційної системи та потребує значно більше обчислювальних ресурсів. На відміну від віртуальних машин контейнери запускаються швидше, споживають менше пам’яті, та краще підходять для сценаріїв з високою частотою деплоїв та CI/CD-процесами.

Портативність, як перевага контейнеризації, забезпечує легку міграцію контейнера із зібраним додатком з локального середовища розробника до хмари або edge-серверів без додаткових змін. Такий підхід дозволяє забезпечити однакову поведінку сервісу на будь-якій інфраструктурі, зменшуючи кількість помилок, які пов’язані з залежностями або зміною середовища розгортання.

Платформа Docker підтримує overlay-мережі, що дозволяє організовувати складні топології зв’язку між контейнерами, а також використовує томи для збереження даних, що гарантує збереження стану

додатків поза межами життєвого циклу контейнера [17]. Конфігураційні файли у форматі YAML спрощують опис та управління багатокомпонентними додатками, забезпечуючи легкість та зручність налаштувань. Особливості функціонування Docker та широкий спектр вбудованих інструментів для автоматизації процесів забезпечують перевагу платформи Docker для розгортання додатків різного масштабу.

2.2 Інструменти оркестрації в Docker

Оркестрація контейнерів є фундаментальним елементом побудови надійної та масштабованої розподіленої інфраструктури, що забезпечує автоматизацію процесів розгортання, масштабування, моніторингу та оновлення сервісів в кластерному середовищі [18]. Зі зростанням складності додатків, особливо тих, що реалізовані за мікросервісною архітектурою, виникає нагальна потреба в централізованому управлінні життєвим циклом контейнеризованих додатків, що підвищує значущість інструментів оркестрації.

Найпоширенішими інструментами оркестрації, які застосовуються в платформі контейнеризації Docker, є Docker Compose, Kubernetes та Docker Swarm. Вибір інструменту оркестрації залежить від масштабів майбутньої мережної архітектури, наявних апаратних ресурсів, а також складності та вимог до адміністрування середовища додатку.

Інструмент Docker Compose найчастіше використовується в локальному середовищі для запуску та управління кількома контейнерами на одній фізичній або віртуальній машині. Він дозволяє описати конфігурацію сервісів у зручному YAML-файлі, що забезпечує простоту та швидкість налаштування. Це дозволяє керувати інфраструктурою на всіх етапах розробки та тестування [19]. Проте Docker Compose має суттєві обмеження щодо масштабування через те, що відсутнє автоматичне управління кількістю контейнерів, які необхідні для забезпечення стабільної роботи

додатку. Саме тому для масштабування необхідно безпосередньо вносити зміни до конфігураційного файлу. Також він не підтримує розподілену кластерну архітектуру та не забезпечує автоматичного відновлення сервісів в разі збоїв. Через такі обмеження Docker Compose не використовують для управління додатками, де критично важливими є висока доступність та автоматизація управління.

Kubernetes підтримує широке коло функцій, необхідних для складних production-сценаріїв. На рисунку 2.2 наведено ключові відмінності інструментів Docker Compose та Kubernetes в управлінні контейнерів (нодів), на яких розгорнуто додаток. Kubernetes забезпечує автоматичне масштабування, безперервне оновлення без простоїв, складне управління мережею, а також інтеграцію з системами зберігання даних. Цей інструмент застосовують в масштабованих кластерах для розгортання додатків, які вимагають масштабування, високої доступності та відмовостійкості.

Проте впровадження Kubernetes пов’язане зі значними ресурсними затратами та складнощами в адмініструванні, що потребує досвіду адміністратора. Саме тому Kubernetes не використовують для проектів малого та середнього бізнесу через значні накладні витрати на адміністрування та вимог до апаратних ресурсів, що забезпечують стабільну та безперервну роботу Kubernetes. Основними перевагами Kubernetes є забезпечення масштабованості, гнучкості та підтримка складних production-сценаріїв. Також до недоліків Kubernetes відносять застосування додаткових систем моніторингу для забезпечення контролю роботи кластера [20].

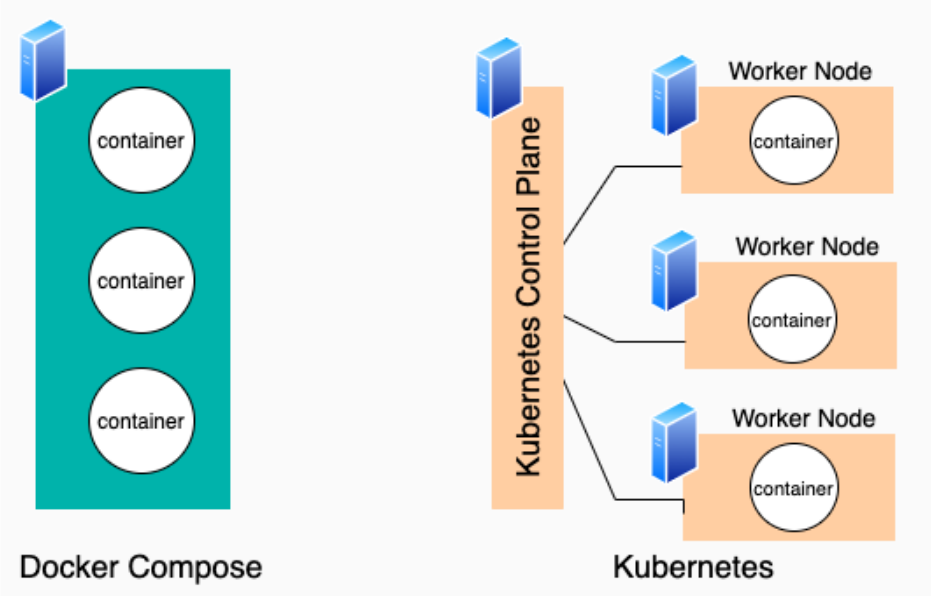

Рисунок 2.2 – Порівняння архітектур Docker Compose та Kubernetes

На противагу попередньо розглянутих інструментів, Docker Swarm поєднує просту конфігурацію з необхідним рівнем функціональності, що застосовується для побудови кластерів малого та середнього масштабу (рисунок 2.3).

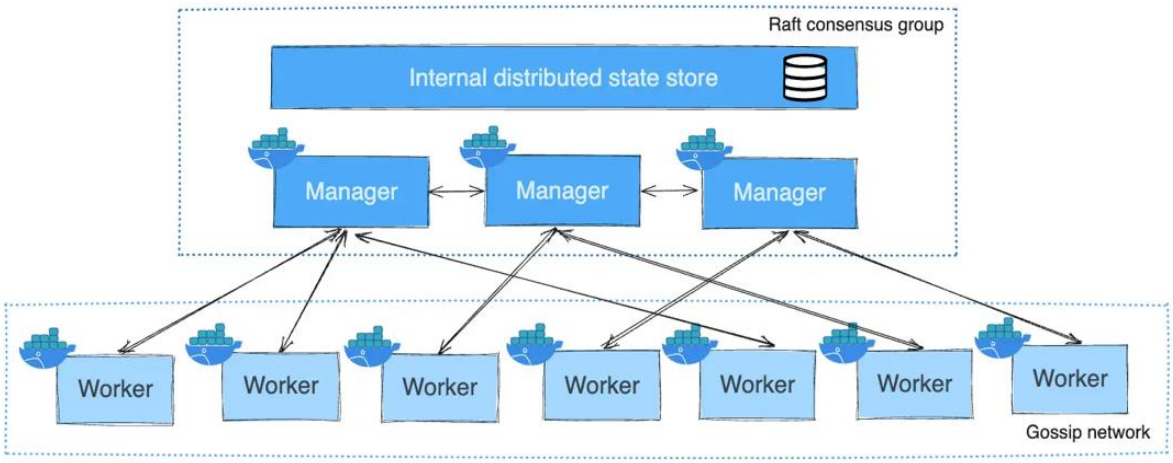

Рисунок 2.3 – Структура кластера із застосуванням Docker Swarm

На відміну від Kubernetes, Docker Swarm не вимагає встановлення додаткових компонентів чи створення складної інфраструктури. Ініціалізація Docker Swarm виконується за допомогою однієї команди, яку виконують в інтерфейсі Docker на кожному вузлі кластеру, що спрощує початкове

налаштування та скорочує час розгортання інфраструктури. Після створення таким чином кластера всі вузли об'єднуються в єдиний віртуальний хост, який автоматично розподіляє контейнери між контейнерними нодами та підтримує балансування навантаження на рівні сервісів (рисунок 2.4).

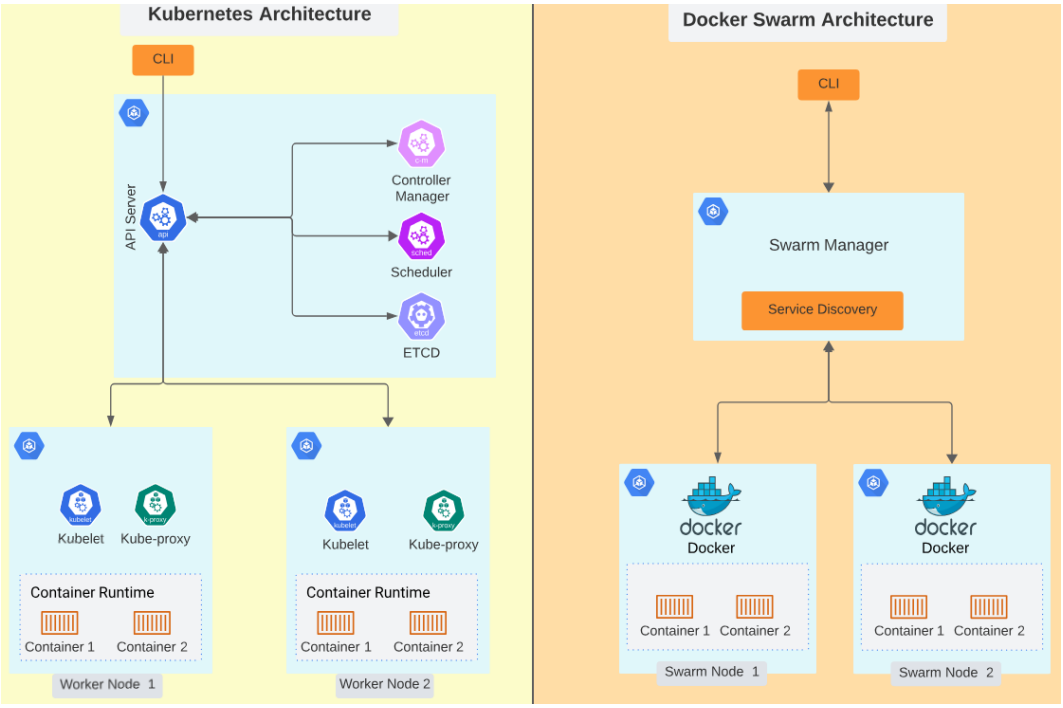

Рисунок 2.4 – Порівняння архітектури Kubernetes та Docker Swarm

До основних переваг Docker Swarm належать легкість адміністрування, відсутність залежностей від зовнішніх систем чи плагінів, а також інтеграція зі стандартним набором інструментів Docker. Незважаючи на те, що Docker Swarm має менш детальні налаштування та спрощену політику управління мережею кластера в порівнянні з Kubernetes, його функціоналу достатньо для управління додатками з високими вимогами до швидкості розгортання, складності адміністрування та масштабування.

Недоліком цього інструменту є менша кількість доступних плагінів для управління кластером контейнерів, що знижує його інтеграцію з різними CI/CD-процесами, сумісністю з системами моніторингу та логування, а також на відсутності інтеграції з мережними драйверами для додаткового використання апаратних ресурсів. Таким чином застосування Docker Swarm

доцільно для забезпечення стабільної та безперервної роботи кластеру контейнерів в умовах обмежених апаратних ресурсів [21].

2.3 Методи забезпечення надійної роботи додатків

Одним з ключових напрямів забезпечення надійності сучасних розподілених систем є впровадження методів, що підвищують їх відмовостійкість, яка полягає в здатності продовжувати функціонування в разі збоїв окремих компонентів. Ці методи ґрунтуються на сукупності архітектурних підходів і технологічних рішень, які мінімізують ризики повного виходу системи з ладу та забезпечують стабільну роботу додатків в режимі реального часу.

Мікросервісна архітектура є одним з найбільш ефективних архітектурних рішень для розгортання додатків з високими вимогами до стабільної та безперервної роботи, яка передбачає декомпозицію програмного забезпечення на окремі, ізольовані сервіси, кожен з яких виконує чітко визначену функцію. Такий підхід дозволяє локалізувати помилки в межах окремого сервісу, не впливаючи на функціонування всієї системи, спрощує впровадження механізмів автоматичного масштабування, відновлення та оновлення. Мікросервіси значно зменшують площу відмови та підвищують адаптивність системи до змін у навантаженні на відміну від монолітних додатків, де збій одного модуля може призвести до повної зупинки системи або недоступності певного сервісу.

Іншим підходом до забезпечення високого рівня відмовостійкості є модульна або serverless-архітектура. Проте така архітектура часто поступається мікросервісам в гнучкості масштабування та контролі. Водночас мікросервісна архітектура має свої недоліки, які полягають в зростанні технічної складності, потребі в розвинутій системі моніторингу, централізованому логуванні, трасуванні запитів та забезпеченні надійної взаємодії між сервісами [22].

Проте, мікросервісний підхід до розгортання додатків, попри недоліки, вважається базовим для побудови високонадійних додатків в хмарних та кластерних середовищах.

Другим ключовим механізмом забезпечення надійності є горизонтальне масштабування, що реалізується через реплікацію сервісів в межах кластера. В протилежність вертикальному масштабуванню, що вимагає додавання ресурсів на рівні окремого вузла, горизонтальний підхід дозволяє динамічно розгортати нові екземпляри сервісу на різних вузлах без зупинки системи [23, 24]. Такий підхід більш ефективний у випадках нерівномірного навантаження, оскільки забезпечує гнучке розподілення запитів між екземплярами сервісу. Хоча вертикальне масштабування є простішим на рівні конфігурації, воно має фізичні обмеження апаратних ресурсів та не забезпечує захисту системи від відмови всього вузла. Реплікація в Docker Swarm реалізується шляхом додавання визначеної кількості екземплярів в конфігураційному файлі, а завдяки політикам автоматичного перезапуску, збої окремих екземплярів сервісів не призводять до втрати доступності всього додатку.

Іншим важливим підходом до забезпечення надійної роботи додатку є балансування навантаження [25, 26]. В середовищі Docker з використанням інструменту Docker Swarm балансування навантаження реалізується на двох рівнях. На внутрішньому рівні Docker Swarm автоматично розподіляє вхідні запити між репліками сервісів, забезпечуючи рівномірне використання ресурсів та безперервність обслуговування в разі відмови окремих контейнерів. Для реалізації балансування навантаження на зовнішньому рівні використовується додатковий сервіс, зокрема проксі-сервер. Такий підхід може бути реалізовано за допомогою проксі-сервера Nginx, який виконує функцію точки входу до системи, забезпечуючи додаткову маршрутизацію, а також має функції кешування та управління політиками доступу до відповідних сервісів або ресурсів додатку. Така дворівнева структура дозволяє досягти високої гнучкості та адаптивності: В разі недоступності

окремого сервісу або вузла запити автоматично перенаправляються до працездатних екземплярів, що мінімізує ризик збоїв та втрат даних. Найбільш відомими проксі-серверами, які використовують в якості балансувальників навантаження в мікросервісах є Nginx, HAProxy та Envoy. Кожен з цих проксі-серверів має свої особливості, що визначають доцільність його використання в залежності від архітектури системи, вимог до продуктивності, масштабованості та складності конфігурації.

Nginx відрізняється простотою налаштування, високою продуктивністю в якості зворотного проксі та можливістю обробки статичного контенту (рисунок 2.5). Завдяки своїй легкості його використовують для додатків з нескладною архітектурою та мінімальними витратами на обслуговування для забезпечення високої продуктивності додатків [27].

HAProxy орієнтований на високонавантажені системи, через що його використовують у великих production-середовищах для забезпечення стабільної роботи . низької затримки та забезпечення додаткової сумісності з системами моніторингу [28].

До ключових функцій проксі-сервера Envoy належать автоматичне виявлення сервісів, динамічна маршрутизація запитів, а також інтеграція з системами автентифікації та контролю доступу. Таким чином, Envoy може використовуватися в якості зворотного проксі-серверу для забезпечення підвищених вимог до захисту ресурсів. Також він забезпечує високу гнучкість та керованість, що дозволяє використовувати його для побудови реалізації відмовостійких та масштабованих мережних архітектур мікросервісів в хмарних середовищах [29].

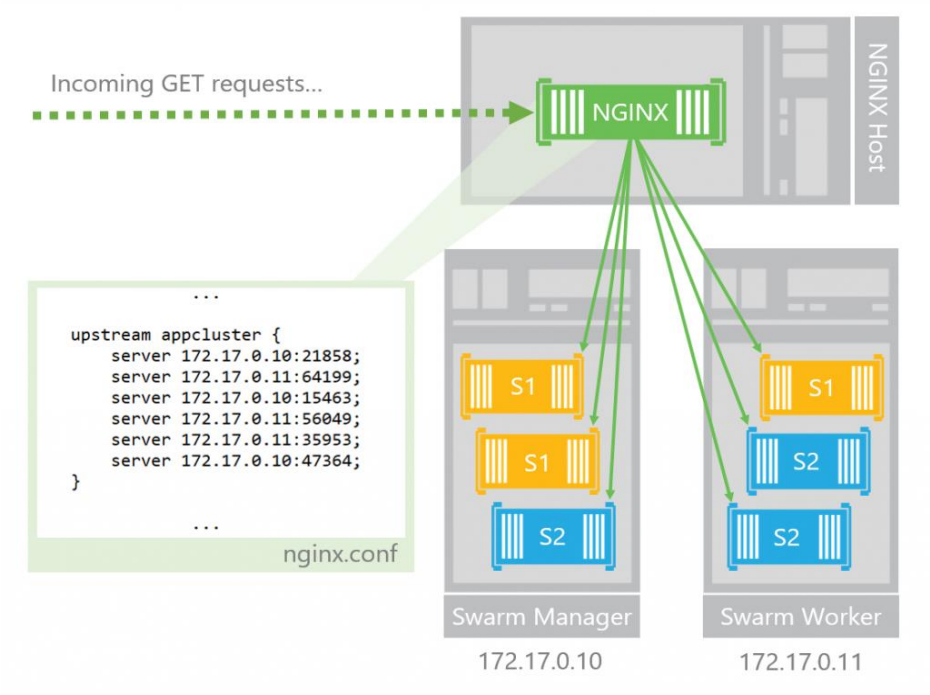

Рисунок 2.5 – Схема застосування проксі-сервера Nginx для балансування навантаження в кластері Docker Swarm

Для забезпечення високої надійності та безперервності роботи застосовують кешування, яке дозволяє зменшити навантаження та підвищити загальну продуктивність. Кешування мінімізує затримки доступу до даних, підвищуючи стійкість системи до пікових навантажень, коли повторювані запити можуть швидко оброблятися з кешу без звернення до основних сервісів додатку. Для мікросервісної архітектури найпоширеним сервісом кешування є Redis, який також може виконувати функцію розподілу навантаження [30]. Інші підходи до кешування, зокрема HTTP-проксі або кешування в веббраузері користувача, не забезпечують належної гнучкості та масштабованості для застосування в кластерному або контейнерному середовищі.

Ключовим елементом віртуалізованої інфраструктури контейнерних додатків є використання overlay-мереж, які забезпечують логічну ізоляцію сервісів в межах кластера. Завдяки цьому сервіси обмінюються інформацією через DNS-імена, незалежно від фізичного розташування контейнерів або вузлів. Така мережна абстракція спрощує масштабування та оркестрацію,

оскільки при додаванні нових вузлів або контейнерів не виникає потреби в додатковому налаштуванні мережної інфраструктури, що може спричинити помилки та підвищує витрати на обслуговування. Overlay-мережі сприяють підвищенню відмовостійкості, оскільки мережний трафік в разі відмови одного з вузлів, автоматично перенаправляється на доступні сервіси, забезпечуючи безперервність комунікації в кластері.

Використання зовнішніх Docker-томів є ще одним підходом до забезпечення цілісності та збереження даних. Такий підхід доцільно застосовувати у випадках збоїв або оновлень, забезпечуючи таким чином стабільну та безперервну додатку. На відміну від зберігання у файловій системі самого контейнера, томи існують незалежно від життєвого циклу сервісу. Це дозволяє зберігати дані в разі видалення чи перезапуску контейнера, а також значно спрощує процес резервного копіювання та перенесення інформації між середовищами.

Таким чином, забезпечення відмовостійкості контейнеризованих додатків ґрунтується на комплексному підході до забезпечення стабільної та безперервної роботи та технологіях віртуалізації на всіх рівнях мікросервісної інфраструктури. Поєднання механізмів балансування навантаження, реплікації сервісів та автоматичного масштабування забезпечує функціонування контейнеризованого додатку в умовах підвищеного навантаження та ризиків часткових або повних збоїв розгорнутої інфраструктури додатку.

3 РОЗРОБКА СЕГМЕНТУ КОНТЕЙНЕРИЗОВАНОГО ДОДАТКУ В DOCKER

3.1 Моделювання сегменту додатку в Docker

Для моделювання функціонування контейнеризованого додатку було створено тестове середовище на базі ноутбука MacBook Air, обладнаного процесором з щонайменше вісьмома ядрами та максимальною тактовою частотою до 2,9 ГГц. Обсяг оперативної пам’яті становить 8 ГБ, при цьому розмір доступного простору для зберігання образів контейнерів, логів та іншої службової інформації становить 256 ГБ, що відповідає умовам обмежених апаратних ресурсів. Пропускна здатність мережного каналу досягає 1 Гбіт/с. Тестове середовище реалізовано з використанням платформи контейнеризації Docker, що забезпечує розгортання мікросервісного додатку на основі Node.js в ізольованих контейнерах. На рисунку 3.1 представлено схему розгортання додатку, яка складається з чотирьох основних сервісів, кожен з яких розміщено в окремому контейнері: веб-додаток на базі Node.js, база даних PostgreSQL, сервіс кешування Redis та проксі-сервер Nginx. Всі сервіси взаємодіють між собою в межах overlay-мережі, створеної для забезпечення внутрішньої комунікації між контейнерами. Система зберігання даних реалізована застосуванням зовнішнього Docker-тома, що гарантує збереження інформації в разі перезапуску чи відмови окремих контейнерів.

Процес розгортання додатку реалізовано в кілька етапів. Спочатку виконується збірка образу Node.js-додатку за допомогою вбудованого інструменту Docker Compose наступною командою:

```
docker build -t undergraduate_thesis ./node
```

Наступним кроком є ініціалізація запуску всього стеку додатку. Запуск додатку виконується з використанням конфігураційного файлу docker-compose.yml, вміст якого наведено в додатку В, за наступною командою:

```
docker stack deploy -c docker-compose.yml undergraduate_thesis
```

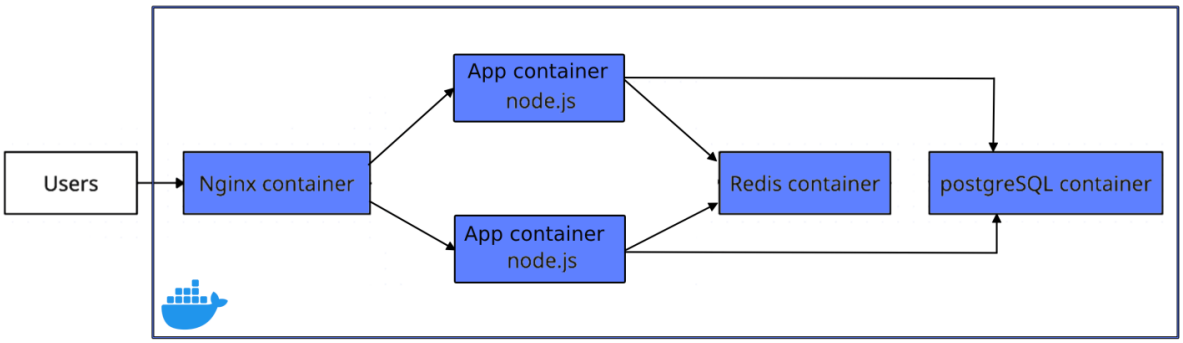

Рисунок 3.1 – Схема загальної взаємодії контейнеризованих компонентів додатку

Наступним кроком є ініціалізація кластеру контейнерів для налаштування overlay-мережі та керування контейнерами додатку. Таким чином менеджер контейнерів Docker Engine переходить в режим роботи оркестратору Docker Swarm, що дозволяє в подальшому масштабувати основний сервіс додатку Node.js в разі потреби. Ініціалізація Docker Swarm виконується наступною командою:

```
docker swarm init
```

Після ініціалізації сервіси автоматично запускаються у власних контейнерах визначеної overlay-мережі. Кожен з них отримує визначені в конфігурації ресурси та політики поведінки, як показано на рисунку 3.2. Зокрема, для додатку визначено restart-політику, яка дозволяє автоматично перезапустити контейнер в разі завершення роботи з помилкою. Стан усіх

сервісів додатку перевіряється командою:

docker service ls

```
ID             NAME                          MODE         REPLICAS   IMAGE                          PORTS
v9ltrgvl47mf   undergraduate_thesis_app      replicated   2/2        undergraduate_thesis:latest
hah6bmsx4ctn   undergraduate_thesis_db       replicated   1/1        postgres:14-alpine
zpvzumj38xc6   undergraduate_thesis_nginx    replicated   1/1        nginx:latest                   *:80->80/tcp
548wx4bwu5li   undergraduate_thesis_redis    replicated   1/1        redis:7-alpine
```

Рисунок 3.2 – Стан сервісів додатку після ініціалізації Docker Swarm

Перевірка успішного розгортання додатку виконується за допомогою переходу за посиланням в браузері за адресою http://localhost. Додатково в застосунку було налаштовано вивід активних з’єднань на головну сторінку додатку, як показано на рисунку 3.3.

Рисунок 3.3 – Вивід активних з’єднань додатку на головній сторінці вебдодатку

Після успішного розгортання додатку було виконано тестування основних сервісів на стійкість до збоїв. Для тестування було обрано декілька сценаріїв, які дозволяють оцінити поведінку системи у випадку збоїв, оновлень та можливості системи до масштабування в разі збільшення навантаження та відновлення після збоїв.

За першим сценарієм передбачено відновлення контейнеру за допомогою Docker Swarm в результаті видалення одного з контейнерів Node.js. Після фіксації збою в роботі контейнера Docker Swarm автоматично перезапустив контейнер, використовуючи функцію автоматичного

перезапуску та відновлення роботи сервісу, що показано на рисунку 3.4.

```
bogdanromanenko@MacBook-Air-Bogdan undergraduate_thesis % docker kill 8a6f2f0757be
8a6f2f0757be
bogdanromanenko@MacBook-Air-Bogdan undergraduate_thesis % docker service ps undergraduate_thesis_app
ID             NAME                           IMAGE                          NODE             DESIRED STATE   CURRENT STATE            ERROR
               PORTS
f68f6ecqwxa3   undergraduate_thesis_app.1      undergraduate_thesis:latest    docker-desktop   Running         Running 4 minutes ago

sv8n8xq6u8ha   undergraduate_thesis_app.2      undergraduate_thesis:latest    docker-desktop   Running         Starting 6 seconds ago

9l1jxq5ibiwj    \_ undergraduate_thesis_app.2  undergraduate_thesis:latest    docker-desktop   Shutdown        Failed 11 seconds ago    "task: non-zer
o exit (137)"
```

Рисунок 3.6 – Автоматичне відновлення репліки після примусового завершення

В другому сценарії тестування було реалізовано масове оновлення додатку, яке передбачає поетапний перезапуск контейнерів із заданим часовим інтервалом. Це забезпечує безперервність роботи сервісів під час оновлення, знижуючи ризик простоїв та підвищуючи надійність системи. На рисунку 3.7 показано, що оновлення сервісів в контейнерах виконується послідовно, тобто після повного оновлення одного контейнера починається оновлення другого контейнеру.

```
bogdanromanenko@MacBook-Air-Bogdan undergraduate_thesis % docker service update --force undergraduate_thesis_app
undergraduate_thesis_app
overall progress: 2 out of 2 tasks
1/2: running   [==================================================>]
2/2: running   [==================================================>]
verify: Service converged
bogdanromanenko@MacBook-Air-Bogdan undergraduate_thesis % docker service ps undergraduate_thesis_app
ID             NAME                           IMAGE                          NODE             DESIRED STATE   CURRENT STATE                ERROR
               PORTS
mzkh0fucq7w8   undergraduate_thesis_app.1      undergraduate_thesis:latest    docker-desktop   Running         Running 15 seconds ago

f68f6ecqwxa3    \_ undergraduate_thesis_app.1  undergraduate_thesis:latest    docker-desktop   Shutdown        Shutdown 46 seconds ago

q74kp1qtou7h   undergraduate_thesis_app.2      undergraduate_thesis:latest    docker-desktop   Running         Running about a minute ago

sv8n8xq6u8ha    \_ undergraduate_thesis_app.2  undergraduate_thesis:latest    docker-desktop   Shutdown        Shutdown about a minute ago

9l1jxq5ibiwj    \_ undergraduate_thesis_app.2  undergraduate_thesis:latest    docker-desktop   Shutdown        Failed 4 minutes ago         "task: no
n-zero exit (137)"
```

Рисунок 3.7 – Результати поетапного оновлення сервісів додатку

В третьому сценарії тестування, після перевірки механізму оновлення двох контейнерів з Node.js було змодельовано відмову бази даних PostgreSQL. Контейнер з базою даних був примусово зупинений, що

спричинило автоматичний перезапуск цього сервісу за допомогою оркестратора Docker Swarm. Інші компоненти системи залишилися працездатними, незважаючи на те, що сервіс Node.js тимчасово повертав повідомлення про помилки підключення (рисунок 3.8). Після відновлення роботи бази даних додаток продовжив функціонувати в штатному режимі без необхідності додаткового втручання адміністратора.

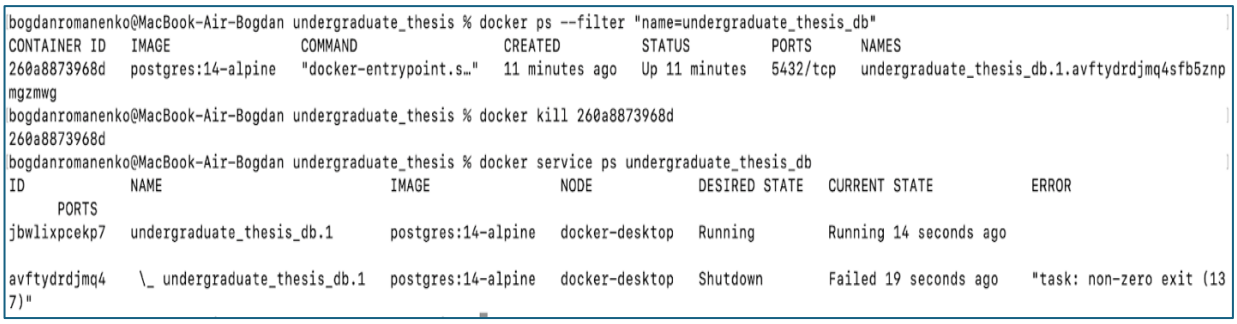

Рисунок 3.8 – Результати стабільної роботи додатку під час відновлення роботи бази даних

Додатково було проведено перевірку цілісності даних, які знаходились в базі даних під час імітації збою в роботі PostgreSQL. Для цього було додано новий запис в базу даних перед виконанням симуляції. Після відновлення роботи PostgreSQL було перевірено цілісність даних (рисунок 3.9).

Таким чином, успішне розгортання додатку та експериментальне тестування доводить, що інструмент оркестрації Docker Swarm застосовується для управління контейнеризованими сервісами, забезпечуючи високу доступність, масштабованість та стійкість до збоїв.

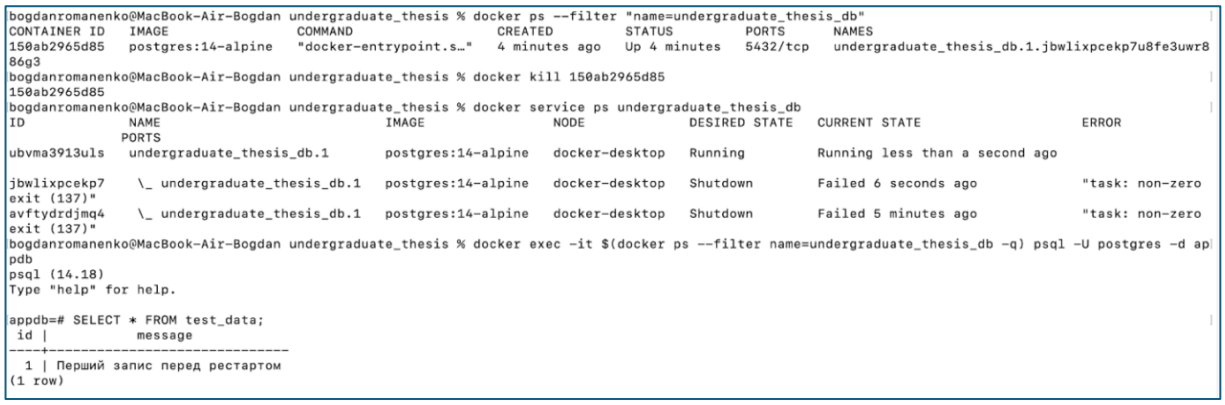

Рисунок 3.9 – Підтвердження збереження запису після симуляції збою сервіса PostgreSQL

Також було проведено тестування розгорнутого додатку в умовах підвищеного навантаження. Перевірка виконувалася за допомогою утиліти wrk, яка використовується для навантажувального тестування веб-сервісів. В межах тестування було змодельовано 100 одночасних з’єднань, розподілених між 4 потоками, а загальний час тестування склав 30 секунд. Для проведення симуляції роботи додатку в умовах підвищеного навантаження була використана наступна команда:

```
wrk -t4 -c100 -d30s http://localhost
```

За результатами тестування було отримано наступні дані, наведені на рисунку 3.10. Отримані показники експериментального моделювання свідчать про те, що система в умовах підвищеного навантаження забезпечує низький середній час перебування запитів у черзі та складає 0.0000484 с.

Також експериментальне значення середньої кількості запитів в черзі, що складає 0.0260 запита, свідчить про ефективне застосування механізмів кешування та балансування навантаження.

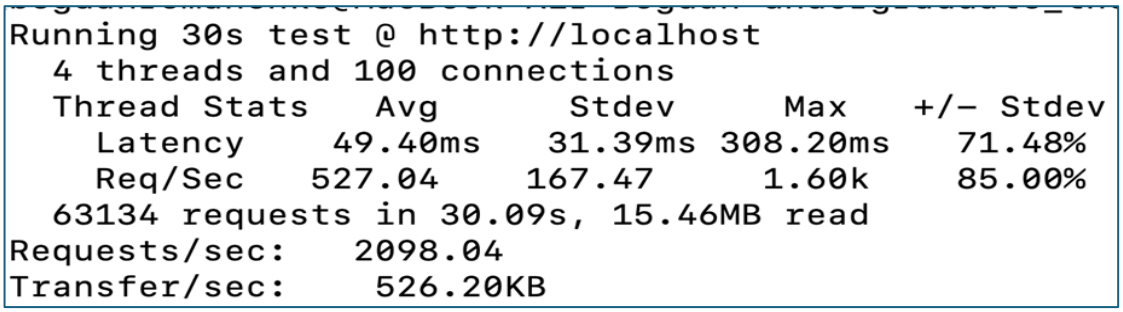

Рисунок 3.10 – Результати експериментального тестування додатку в умовах підвищеного навантаження

Проте загальний середній час перебування запиту в системі 0.0494 с та середня кількість запитів в системі 26.05 вказують обмеження апаратних ресурсів та необхідність застосування масштабування основних сервісів обробки запитів.

3.2 Аналітичне моделювання роботи додатку

Для оцінки роботи додатку в умовах реального часу було використано модель системи масового обслуговування типу M/M/n. В цій моделі передбачається наявність n незалежних ідентичних серверів, що відповідає сервісу Node.js в даній роботі, кожен з яких обробляє запити з середньою інтенсивністю обслуговування μ, а вхідний потік запитів надходить згідно з потоком Пуассона з інтенсивністю λ. Ця модель найбільш точно описує роботу масштабованого додатку, який функціонує в середовищі контейнеризації Docker з використанням оркестратору Docker Swarm, де кілька екземплярів сервісу Node.js паралельно обробляють запити, що надходять до додатку. Розрахунки ключових показників цієї моделі дозволяють моделювати поведінку додатку при підвищеному навантаженні та прогнозувати потребу в масштабуванні [31]. Основними показниками ефективності роботи системи масового обслуговування моделі M/M/n є коефіцієнт завантаження системи, ймовірність утворення черги, середня кількість запитів у черзі, середня кількість запитів у системі, середній час

перебування запиту в системі, а також середній час очікування в черзі.

Коефіцієнт завантаження системи ρ визначає ступінь завантаження обслуговуючих каналів, в даному випадку екземплярів сервісу Node.js, причому система вважається стабільною та стійкою до збоїв, коли цей коефіцієнт не перевищує 1. Коефіцієнт завантаження системи розраховується наступним чином:

$$\rho = \frac{\lambda}{n\mu}, \tag{1}$$

де λ – інтенсивність вхідного потоку запитів, зап/с;

μ – інтенсивність обслуговування одним каналом, зап/с;

n – кількість паралельних каналів (серверів) обслуговування, од.

Ймовірність утворення черги P_q визначає можливість вхідного запиту очікувати на обробку в той час, коли сервер обробляє попередній запит. Цей показник вказує на необхідність масштабування системи або застосування додаткових сервісів, зокрема балансування навантаження або кешування, для обробки запитів в умовах стрімкого зростання обсягу вхідних запитів. Ймовірність утворення черги P_q може бути обчислена як:

$$P_q = \frac{\frac{(n\rho)^n}{n!} \cdot \left(\frac{1}{1-\rho}\right)}{\sum_{k=0}^{n-1} \frac{(n\rho)^k}{k!} + \frac{(n\rho)^n}{n!} \cdot \left(\frac{1}{1-\rho}\right)}, \tag{2}$$

де k – кількість одночасно зайнятих каналів (серверів), од;

λ – інтенсивність вхідного потоку запитів, зап/с;

μ – інтенсивність обслуговування одним каналом, зап/с;

n – кількість паралельних каналів (серверів) обслуговування, од.

Середня кількість запитів в черзі L_q дозволяє оцінити рівень затримки в системі та спрогнозувати потребу в масштабуванні серверів для обробки запитів. Середня кількість запитів в черзі L_q розраховується наступним чином:

$$L_q = \frac{P_q \cdot \rho}{1 - \rho}, \tag{3}$$

де P_q – ймовірність утворення черги, од;

ρ – коефіцієнт завантаження системи, од.

Середня кількість запитів в системі L складається з заапитів, що перебувають в системі в черзі та на обробці. Цей показник визначає середне навантаження системи та може бути розрахований як:

$$L = L_q + \frac{\lambda}{\mu}, \tag{4}$$

де L_q – середня кількість запитів в черзі, од;

λ – інтенсивність вхідного потоку запитів, зап/с;

μ – інтенсивність обслуговування одним каналом, зап/с.

Середній час перебування запиту в системі W визначає час перебування кожного запиту в системі та розраховується як:

$$W = \frac{L}{\lambda}, \tag{5}$$

де L – середня кількість запитів в системі, од;

λ – інтенсивність вхідного потоку запитів, зап/с.

Середній час очікування запиту в черзі W_q характеризує загальну

затримку системи

$$W_q = \frac{L_q}{\lambda}, \tag{6}$$

де L_q – середня кількість запитів в черзі, од;

λ – інтенсивність вхідного потоку запитів, зап/с.

Розрахунок ключових показників цієї моделі було реалізовано в середовищі MATLAB за наведеним у додатку В кодом. В розрахунку враховано, що інтенсивність вхідного потоку запитів складає 1000 зап/с, інтенсивність вихідного потоку запитів, тобто інтенсивність оброблених додатком потоку запитів складає 1000 зап/с, а кількість обслуговуючих каналів (серверів) становить 2, що відповідає кількості контейнерів з сервісом Node.js. За розрахунком було отримано наступні результати: коефіцієнт завантаження дорівнює 0.5000; ймовірність черги складає 0.3333; середня кількість заявок у черзі складає 0.3333; середня кількість у системі складає 1.3333; середній час у черзі складає 0.000333 с; середній час у системі складає 0.001333 с.

Таким чином, результати аналітичного моделювання дозволяють оцінити ефективність роботи контейнеризованого додатку у умовах підвищеного навантаження.

За отриманими результатами було побудовано графік залежності середньої затримки в системі від кількості запитів вхідного трафіку, що дозволяє оцінити навантаження додатку в умовах підвищеного навантаження (рисунок 3.12).

Отримані результати експериментального та аналітичного моделювання, наведених у порівняльній таблиці 3.1, свідчать про суттєві відмінності між аналітичним розрахунком та реальною поведінкою системи в умовах підвищеного навантаження.

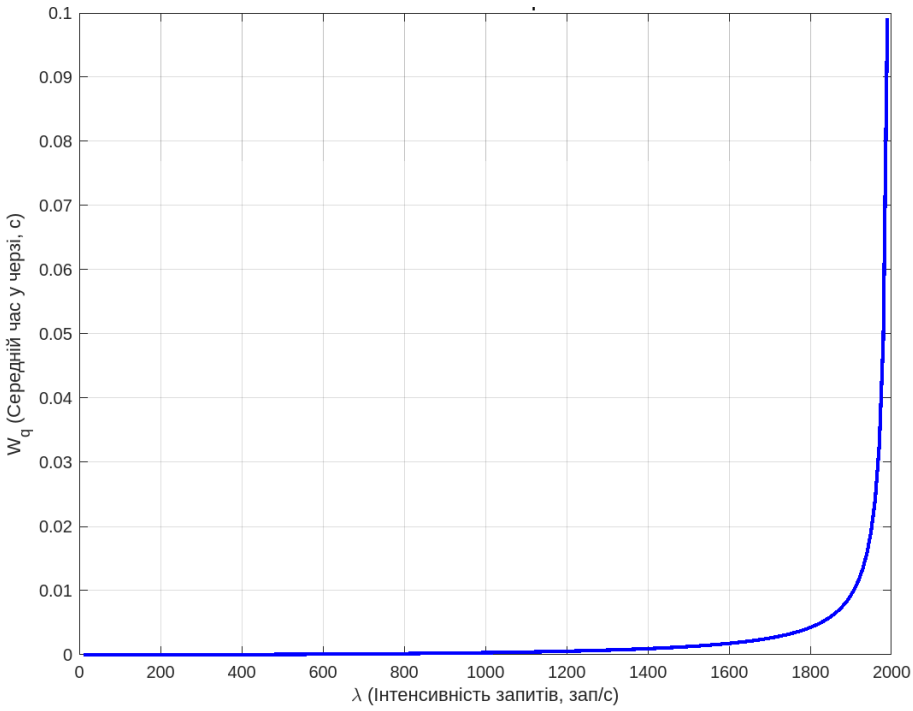

Рисунок 3.12 – Графік залежності середньої затримки в системі від кількості вхідних запитів

Таблиця 3.1 – Результати експериментально та аналітичного тестування роботи додатку в умовах підвищеного навантаження

Показник	Позначення	Аналітичне моделювання	Експериментальне моделювання
Коефіцієнт завантаження	ρ	0.5000	0.2635
Ймовірність черги	P_q	0.3333	–
Середня кількість в черзі	L_q	0.3333	0.0260
Середня кількість у системі	L	1.3333	26.05
Середній час в черзі	W_q	0.000333	0.0000484
Середній час в системі	W	0.001333	0.0494

Експериментальне моделювання виявило наявність додаткових затримок, зумовлених взаємодією між контейнерами, обробкою вхідних запитів балансувальником навантаження в умовах обмежених апаратних ресурсів. Проте показники середньої кількості запитів в черзі та середнього часу в черзі в експериментальному моделюванні вказують на ефективне

використання кешування, що забезпечує зниження навантаження на основні сервіси додатку. Також зниження коефіцієнта завантаження в порівнянні з аналітичним моделюванням вказує на ефективність застосування в архітектурі додатку механізмів балансування навантаження та використання інструменту Docker Swarm для масштабування основних сервісів, зокрема Node.js, під час обробки запитів в умовах підвищеного навантаження.

ВИСНОВКИ

В даній кваліфікаційній роботі було розроблено метод забезпечення надійності контейнеризованого вебдодатку в розподіленому середовищі Docker, що відповідає головній меті роботи. Запропонований метод дозволяє забезпечити стабільну та безперервну роботу додатку, який розгорнуто в контейнерах Docker та забезпечують стабільне функціонування додатку в умовах підвищеного навантаження.

На підставі аналізу наукових досліджень технологій вітуалізації, зокрема контейнеризації, та методів забезпечення відмово стійкості додатків в умовах підвищенного навантаження було запропоновано технологічне рішення, що дозволяє забезпечити стабільну роботу додатку в умовах обмежених ресурсів та динамічно змінюваного навантаження.

Аналітичне моделювання функціонування системи було здійснено з використанням моделі масового обслуговування M/M/n, яка відповідає роботі додатку з кількома основними сервісами для обробки вхідних запитів в режимі реального часу в умовах підвищеного навантаження, які працюють паралельно та одночасно обробляють вхідний потік запитів, що надходять до системи.

Запропонований метод може бути застосовано для застосунків з обмеженими апаратними ресурсами, які функціонують в режимі динамічного змінюваного навантаження. Отримані результати експериментальної перевірки роботи додатку підтверджують практичну доцільність впровадження розробленої архітектури в реальних умовах експлуатації.

Подальші дослідження доцільно спрямувати на вдосконалення даного підходу шляхом інтеграції використаних методів масштабування з механізмами адаптивного моніторингу продуктивності сервісів в режимі реального часу.

ПЕРЕЛІК ДЖЕРЕЛ ПОСИЛАННЯ

1. Shabani, I., Mëziu, E., Berisha, B., & Biba, T. (2021). Design of modern distributed systems based on microservices architecture. International Journal of Advanced Computer Science and Applications, 12(2).

2. Kumari, P., & Kaur, P. (2021). A survey of fault tolerance in cloud computing. Journal of King Saud University-Computer and Information Sciences, 33(10), 1159-1176.

3. Шакарамі, А., Гхобаеї-Арані, М., Шахідінежад, А., Масдарі, М. та Шакарамі, Х. (2021). Схеми реплікації даних у хмарних обчисленнях: огляд. Кластерні обчислення , 24 , 2545-2579.

4. Бліновський, Г., Ойдовська, А. та Пшибилек, А. (2022). Монолітна проти мікросервісної архітектури: оцінка продуктивності та масштабованості. IEEE access , 10 , 20357-20374

5. Jang, HC, & Luo, SY (квітень 2023 р.). Підвищення відмовостійкості вузлів за допомогою кластерів високої доступності в Kubernetes. У 2023 році на 3-й Міжнародній конференції IEEE з електронних комунікацій, Інтернету речей та великих даних (ICEIB) (стор. 30-35). IEEE.

6. Oloruntoba, O. (2025). Architecting Resilient Multi-Cloud Database Systems: Distributed Ledger Technology, Fault Tolerance, and Cross-Platform Synchronization. International Journal of Research Publication and Reviews, 6(2), 2358-2376.

7. Cabrera, O., Oriol, M., Franch, X., & Marco, J. (2021). A context-aware monitoring architecture for supporting system adaptation and reconfiguration. Computing, 103(8), 1621-1655.

8. Ramachandran, K. K. (2023). Optimizing IT Performance: A Comprehensive analysis of Resource Efficiency. International Journal of Marketing and Human Resource Management (IJMHRM), 14(3), 12-29.

9. Arogundade, O. R., & Palla, K. (2023). Virtualization revolution: Transforming cloud computing with scalability and agility.

10. Kesavan, E. (2025). The Impact of Cloud Computing on Software Development: A Review. International Journal of Innovations in Science, Engineering And Management, 269-274.

11. Abdulsalam, Y. S., & Hedabou, M. (2021). Security and privacy in cloud computing: technical review. Future Internet, 14(1), 11.

12. Thyagaturu, A. S., Shantharama, P., Nasrallah, A., & Reisslein, M. (2022). Operating systems and hypervisors for network functions: A survey of enabling technologies and research studies. IEEE Access, 10, 79825-79873.

13. Pandey, B., Mishra, A. K., Yadav, A., Tiwari, D., & Pandey, M. S. (2022). Virtualization using Docker container. In Emerging Real-World Applications of Internet of Things (pp. 157-181). CRC Press.

14. Silva, D., Rafael, J., & Fonte, A. (2024). Toward optimal virtualization: An updated comparative analysis of Docker and LXD container technologies. Computers, 13(4), 94.

15. Sturley, H., Fournier, A., Salcedo-Navarro, A., Garcia-Pineda, M., & Segura-Garcia, J. (2024). Virtualization vs. Containerization, a Comparative Approach for Application Deployment in the Computing Continuum Focused on the Edge. Future Internet, 16(11), 427.

16. Kanthed, S. (2023). Docker vs. Podman: Architecture Differences and Their Impact on Modern Workflows.

17. Chamoli, S., & Mittal, V. (2023, April). Docker Networking: A Security Review. In 2023 7th International Conference on Trends in Electronics and Informatics (ICOEI) (pp. 624-629). IEEE.

18. Eyvazov, F., Ali, T. E., Ali, F. I., & Zoltan, A. D. (2024, March). Beyond containers: orchestrating microservices with minikube, kubernetes, docker, and compose for seamless deployment and scalability. In 2024 11th International Conference on Reliability, Infocom Technologies and Optimization (Trends and Future Directions)(ICRITO) (pp. 1-6). IEEE.

19. Overview of Docker Compose [Електронний ресурс] / Docker Documentation. – Режим доступу: https://docs.docker.com/compose/ – Дата звернення: 10.05.2025.

20. Kubernetes Documentation: Overview [Електронний ресурс] / Kubernetes.io. – Режим доступу: https://kubernetes.io/docs/home/ – Дата звернення: 10.05.2025.

21. Docker Documentation. Docker Swarm overview [Електронний ресурс]. – Режим доступу: https://docs.docker.com/engine/swarm/ – Дата звернення: 27.05.2025.

22. Kamisetty, A., Narsina, D., Rodriguez, M., Kothapalli, S., & Gummadi, J. C. S. (2023). Microservices vs. Monoliths: Comparative Analysis for Scalable Software Architecture Design. Engineering International, 11(2), 99-112.

23. Yekollu, R. K., Haldikar, S. V., Ghuge, T. B., Kader, O. F. M. A., & Biradar, S. S. (2024, December). Resource Management and Scalability in Container Orchestration Platforms: A Comparative Study. In 2024 IEEE 16th International Conference on Computational Intelligence and Communication Networks (CICN) (pp. 1146-1151). IEEE.

24. Naik, N. (2021, April). Performance evaluation of distributed systems in multiple clouds using docker swarm. In 2021 IEEE International Systems Conference (SysCon) (pp. 1-6). IEEE.

25. Kazemi, M. Optimizing Web Service Performance: A Comparative Analysis of Load Balancing Strategies Using NGINX and HAProxy with StoRM WebDAV Deployment.

26. Tkachov V. A method for enhancing the resilience of multisegment corporate computer networks in healthcare institutions / V. Tkachov, Y. Mikhnov // Проблеми інформатизації : тези доп. 12-ї міжнар. наук.-техн. конф., 21-22 листопада 2024 р., м. Баку, м. Харків, м. Бельсько-Бяла : [у 3 т.]. Т. 2 / Нац. ун-т оборони Азерб. республіки [та ін.]. – Харків : Impress, 2024. – С. 60.

27. NGINX Documentation. HTTP Load Balancing [Електронний ресурс]. – Режим доступу: https://docs.nginx.com/nginx/admin-guide/load-balancer/http-load-balancer – Дата звернення: 01.06.2025.

28. HAProxy Technologies. HAProxy Configuration Basics: Load Balance Your Servers [Електронний ресурс]. – Режим доступу: https://www.haproxy.com/blog/haproxy-configuration-basics-load-balance-your-servers – Дата звернення: 01.06.2025.

29. Envoy Proxy Documentation. Load Balancing Overview [Електронний ресурс]. – Режим доступу: https://www.envoyproxy.io/docs/envoy/latest/intro/arch_overview/upstream/load_balancing/load_balancing – Дата звернення: 01.06.2025.

30. Iurchenko, A. (2025). Optimization of Microservices Architecture Performance in High-Load Systems. The American Journal of Engineering and Technology, 7(05), 123-132.

31. J. Tong, M. Wei, M. Pan and Y. Yu, "A Holistic Auto-Scaling Algorithm for Multi-Service Applications Based on Balanced Queuing Network," 2021 IEEE International Conference on Web Services (ICWS), Chicago, IL, USA, 2021, pp. 531-540, doi: 10.1109/ICWS53863.2021.00074.