Міністерство освіти і науки України
Харківський національний університет радіоелектроніки

Факультет _____Комп'ютерних наук_____
(повна назва)

Кафедра _____Інформаційних управляючих систем_____
(повна назва)

КВАЛІФІКАЦІЙНА РОБОТА
Пояснювальна записка

рівень вищої освіти _____другий (магістерський)_____
_____Дослідження методів розгортання оточення програмного_____
забезпечення засобами контейнеризації при виконанні ІТ-проєкту_____
(тема)

Виконав:
студент _2_ курсу, групи _УПГІТм-22-1____
_____Прес Роман Дмитрович_____
(прізвище, ім'я, по батькові)
Спеціальність ____122 Комп'ютерні науки___
(код і повна назва спеціальності)
Тип програми ____освітньо-наукова_____
(освітньо-професійна або освітньо-наукова)
Освітня програма __Управління проєктами в галузі інформаційних технологій_____
(повна назва освітньої програми)

Керівник _к.т.н., доц. Кудрявцева М.С._____
(посада, власне ім'я, прізвище)

Допускається до захисту

Зав. кафедри _____ ___Костянтин ПЕТРОВ__
(підпис) (власне ім'я, прізвище)

2024 р.

Харківський національний університет радіоелектроніки

Факультет _____Комп'ютерних наук_____

Кафедра _____Інформаційних управляючих систем_____

Рівень вищої освіти _____другий (магістерський)_____

Спеціальність _____122 Комп'ютерні науки_____
(код і повна назва)

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

Освітня програма_Управління проєктами в галузі інформаційних технологій
(повна назва)

ЗАТВЕРДЖУЮ
Зав. кафедри __ ___
(підпис)
«_01__»__квітня__ 20 24_ р.

ЗАВДАННЯ
НА КВАЛІФІКАЦІЙНУ РОБОТУ

студентові_____Пресу Роману Дмитровичу_____
(прізвище, ім'я, по батькові)

1. Тема роботи ___Дослідження методів розгортання оточення програмного забезпечення засобами контейнеризації при виконанні ІТ-проєкту_____
_____
затверджена наказом університету від _01 квітня_2024_ р. № _258СТ__
2. Термін подання студентом роботи до екзаменаційної комісії _05.06.2024р
3. Вихідні дані до роботи опис методів розгортання програмного забезпечення засобами контейнеризації при виконанні ІТ-проєкту. Аналіз переваг та недоліків контейнеризації, огляд існуючих рішень контейнеризаці_____
4. Перелік питань, що потрібно опрацювати в роботі _ Аналіз сучасних інструментів контейнеризації: Docker, Kubernetes, Podman та порівняння функціональних можливостей та випадків використання. Методи розгортання контейнеризованих додатків, створення Dockerfile та базових образів, використання Docker Compose для оркестрації мультиконтейнерних додатків, розгортання додатків в Docker Swarm. Переваги та недоліки контейнеризації в ІТ-проєктах, забезпечення масштабованості та надійності, виклики, пов'язані з безпекою контейнерів, проблеми сумісності та залежностей. Кейси використання контейнеризації в реальних ІТ-проєктах, приклади успішних проєктів, аналіз невдалих впроваджень та отримані уроки. Перспективи розвитку технологій контейнеризації:, новітні тенденції у розвитку контейнеризаційних платформ, вплив контейнеризації на архітектуру програмного забезпечення у майбутньому._____
_____
_____
_____

КАЛЕНДАРНИЙ ПЛАН

№	Назва етапів роботи	Терміни виконання етапів роботи	Примітка
1	Аналіз існуючих методів розгортання оточення програмного забезпечення при виконанні ІТ-проєкту та постановка задачі	01.04.2024-13.04.2024	Виконано
2	Аналіз технологій розгортання програмного забезпечення	14.04.2024-21.04.2024	Виконано
3	Дослідження методіу контейнеризвції інструментом Docker	21.04.2024-27.04.2024	Виконано
4	Планування удосконаленя методу розгортання оточення засобом контейнеризації	27.04.2024-08.05.2024	Виконано
5	Практичне використання удосконаленого методу розгортання оточення програмного забезпеченння засобом контейнеризації	09.05.2024-17.05.2024	Виконано
6	Оформлення пояснювальної записки	18.05.2024-19.05.2024	Виконано
7	Підготовка презентації	20.05.2024	Виконано
8	Подання студентом роботи для перевірки на антиплагіат	20.05.2024	Виконано
9	Надання роботи на підпис науковому керівнику	20.05.2024	Виконано
10	Надання роботи на рецензію	24.05.2024	Виконано
11	Надання роботи на підпис завідувачу кафедри	27.05.2024	Виконано
12	Захист кваліфікаційної роботи	07.06.2024	Виконано

Дата видачі завдання 01 квітня 2024 р.

Студент _____
(підпис)

Керівник роботи _____ к.т.н., доц. Кудрявцева М.С
(підпис) (посада, власне ім'я, прізвище)

РЕФЕРАТ

Пояснювальна записка до кваліфікаційної роботи містить: 90 с., 28 рис., 4 табл., 17 джерел, 1 додаток

РОЗГОРТАННЯ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ, DOCKER, КОНТЕЙНЕРИЗАЦІЯ, МІКРОСЕРВІСНА АРХІТЕКТУРА, E-COMMERCE

Об’єкт дослідження – процес розгортання оточення ПЗ в ІТ-проєкті з акцентом на УП.

Предмет дослідження – методи розгортання оточення ПЗ засобами контейнеризації при виконанні ІТ-проєктів з точки зору УП.

Мета дослідження – вивчення та аналіз сучасних методів розгортання оточення ПЗ за допомогою засобів контейнеризації в рамках ІТ-проєктів, з акцентом на ефективне УП.

Проведено аналіз ПО та існуючих методів розгортання оточень. Розглянуто концепції контейнеризації, проаналізовано сучасні інструментальні засоби для розгортання ПЗ засобами контейнеризації. Приділено увагу аспектам УП, таким як планування, ресурсозатрати та ризик-менеджмент.

На підставі проведеного аналізу запропоновано покращений метод розгортання оточень з використанням Docker для мікросервісних архітектур в eCommerce проєктах. Виконано експериментальну перевірку удосконаленого методу. У контексті УП досліджено аспекти зменшення ризиків, покращення комунікацій між командами та підвищення ефективності.

Результати дослідження демонструють перспективи застосування Docker в eCommerce проєктах з мікросервісною архітектурою. Застосування Docker дозволяє підвищити загальну ефективність проєкту, зменшити час на розгортання та інтеграцію нових сервісів, що в свою чергу підвищує конкурентоспроможність та адаптивність бізнесу.

ABSTRACT

The explanatory note to the qualification work contains: 90 pages, 28 pictures, 4 tables, 17 sources, 1 attachment.

SOFTWARE DEPLOYMENT, DOCKER, CONTAINERIZATION, IT PROJECTS, MICROSERVICE ARCHITECTURE, E-COMMERCE

The object of the study – the process of deploying the software environment in an IT project with an emphasis on PM.

The subject of the study – methods of deploying the software environment of containerization during the implementation of IT projects in the point of view of PM.

The purpose of the research is the study and analysis of modern methods of deploying the software environment using containerization tools within the framework of IT projects, with an emphasis on effective PM.

An analysis of software and existing methods of deployment of environments was carried out. Concepts of containerization are considered, modern tools for deploying software using containerization tools are analyzed. Attention is paid to aspects of PM, such as planning, resource consumption and risk management.

Based on the analysis, an improved method of deploying environments using Docker for microservice architectures in e-commerce projects is proposed. Experimental verification of the improved method was performed. Aspects of reducing risks, improving communication between teams and increasing efficiency have been investigated in the context of PM.

The results of the study demonstrate the prospects of using Docker in eCommerce projects with a microservice architecture. The use of Docker allows you to increase project overall efficiency, reduce the time for deployment, integration of new services, which in turn increases competitiveness, adaptability of the business.

ЗМІСТ

СКОРОЧЕННЯ ТА УМОВНІ ПОЗНАКИ

БД – база даних

ВМ – віртуальна машина

ОС – операційна система

ПЗ – Програмне забезпечення

ПО – Предметна область

УП – Управління проєктами

AEM – Adobe Experience Manager

API – Application Programming Interface

ATG – Art Technology Group

AWS – Amazon Web Services

CLI – Command Line Interface

CPU – Central Processing Unit

DR – disaster recovery

GCP – Google Cloud Platform

IDE – Integrated Drive Electronics

IoC – Inversion of Control

IoT – Internet of Things

PM – Project Management

QC – Quality Control

RBAC – Role Based Access Control

RR – Round-Robin

SaaS – Software as a Service

SAP – Systems Applications and Products

WORA – write once, run anywhere

ВСТУП

Сучасний підхід до організації управління складними IT-проєктами базується на науково обґрунтованих методологіях і методах управління. Ефективність виконання IT-проєктів значною мірою залежить від організації планування, управління ресурсами, сукупностями робіт та окремими завданнями. Одним із ключових аспектів у розробці IT-проєктів є наявність декількох середовищ для розгортання програмного забезпечення, що забезпечує високу якість розробки, інтеграції та тестування. Управління цими середовищами вимагає ретельного планування, координації та контролю, що є важливими складовими успішного управління проєктами.

Контейнеризація, як сучасна технологія розгортання програмного забезпечення, грає вирішальну роль у забезпеченні ефективного управління цими середовищами. Завдяки використанню контейнерів, таких як Docker, можна забезпечити портативність, ізоляцію та консистентність середовищ для розробки, тестування, стадійного розгортання та продуктивної експлуатації. Це дозволяє значно спростити процес розгортання та управління програмними системами, підвищуючи їх надійність і масштабованість, що є критичним для успішного завершення проєкту в рамках бюджету та графіка.

У цій роботі проведено глибокий аналіз предметної області, визначено основні концепції контейнеризації та її переваги у порівнянні з традиційними методами. Розглянуто різні інструменти для розгортання програмного забезпечення засобами контейнеризації та проведено порівняння їх функціональних можливостей. Особлива увага приділена дослідженню Docker, як одного з найпопулярніших інструментів для контейнеризації, та його ролі у впровадженні мікросервісної архітектури в eCommerce проєктах.

Дослідження також охоплює аспекти управління проєктами, такі як планування, ризик-менеджмент, управління ресурсами та моніторинг прогресу, що є важливими для успішного впровадження контейнеризації. Використання

Docker в eCommerce проєктах з мікросервісною архітектурою дозволяє значно зменшити час на розгортання та інтеграцію нових сервісів, підвищуючи загальну ефективність проєкту та забезпечуючи конкурентні переваги в швидко змінюваному ринковому середовищі [1].

Кваліфікаційна робота здійснюється відповідно до вимог з організації та інструкцій щодо їх виконання [2] та національного стандарту [3].

Оформлення переліку джерел посилання відбулося відповідно до національного стандарту [4].

1. АНАЛІЗ ПРЕДМЕТНОЇ ОБЛАСТІ ТА ПОСТАНОВКА ЗАДАЧІ ДОСЛІДЖЕННЯ

1.1 Визначення та концепції контейнеризації

Контейнеризація – це пакування програмного коду разом з бібліотеками операційної системи та всіма залежностями, необхідними для запуску коду. Результатом є єдиний, легкий виконуваний файл, який стабільно працює на будь-якій інфраструктурі. Контейнери є більш портативними та ресурсоефективними, ніж віртуальні машини. Фактично, контейнери стали вимірюваною одиницею обліку для сучасних нативних хмарних додатків [5].

Контейнеризація дозволяє розробникам створювати та розгортати додатки швидше та безпечніше. При традиційному методі код розробляється в конкретному цифровому середовищі. Наприклад, розробники переносять код з комп'ютера на віртуальну машину або з операційної системи Linux на Windows. Контейнеризація вирішує цю проблему шляхом об'єднання коду програми з окремими конфігураційними файлами, бібліотеками та залежностями, необхідними для її запуску. Цей єдиний пакет або «контейнер» абстрагує додаток від операційної системи, роблячи його автономним, портативним і здатним безперешкодно працювати на будь-якій платформі або хмарі.

Контейнери мають спільне ядро операційної системи і не потребують додаткових витрат на створення операційної системи для кожної програми. Контейнери мають менший обсяг пам'яті, ніж віртуальні машини, і потребують менше часу для завантаження. Таким чином, можна запустити більше контейнерів з тією ж потужністю, що й одна віртуальна машина. Це підвищує ефективність.

І, мабуть, найголовніше, контейнеризація дозволяє застосовувати принцип write once, run anywhere (WORA). Така портативність прискорює розробку, уможливлює співпрацю із зовнішніми постачальниками рішень та

надає інші помітні переваги. Наприклад, вона ізолює помилки, полегшує управління та знижує рівень безпеки.

1.1.1 Огляд контейнеризації програмного забезпечення

Контейнер – це єдиний виконуваний програмний пакет. У межах цього коду програма збирається разом з усіма відповідними конфігураційними файлами, бібліотеками та залежностями. Контейнерні програми є «ізольованими», оскільки вони не містять копії операційної системи. Замість цього на хостовій операційній системі встановлюється середовище виконання з відкритим кодом (наприклад, Docker), яке слугує каналом для спільного використання операційної системи з іншими контейнерами на тій самій обчислювальній системі.

Інші рівні контейнера, такі як спільні контейнери та бібліотеки, також можуть бути спільними для декількох контейнерів. Це зменшує ємність контейнера за рахунок зменшення накладних витрат на запуск операційної системи у кожній програмі та пришвидшує ініціалізацію. Це підвищує ефективність роботи сервера. Ізоляція додатків у вигляді контейнерів також зменшує ймовірність поширення шкідливого коду в одному контейнері на інші контейнери або проникнення в хост-систему.

Контейнерні програми, ізольовані від основної операційної системи, стають портативними і можуть працювати рівномірно та узгоджено на всіх платформах і хмарах. Контейнери можна легко перенести з ПК на віртуальну машину або з Linux на Windows, і вони можуть стабільно працювати як на традиційних фізичних серверах, так і на віртуальних інфраструктурах, як локальних, так і хмарних. Таким чином, розробники програмного забезпечення можуть продовжувати використовувати інструменти та процеси, з якими вони найкраще знайомі.

Легко зрозуміти, чому компанії оберають контейнеризацію як кращий підхід до розробки та управління додатками. Контейнеризація дозволяє розробникам створювати і розгортати додатки швидше і надійніше, незалежно від того, чи це традиційні моноліти, чи модульні додатки, побудовані на архітектурі мікросервісів. Нові хмарні додатки можна створювати з нуля як контейнерні мікросервіси, що дозволяє розбивати складні додатки на безліч менших, спеціалізованих і керованих сервісів. Перепакування існуючих додатків у контейнери дозволяє ефективніше використовувати обмежені ресурси.

1.1.2 Переваги контейнеризації при виконанні IT-проєктів

Контейнеризація пропонує значні переваги для розробників і команд в цілому, а саме – переносимість, гнучкість, швидкість, відмовостійкість, ефективність, простота керування та безпека.

Контейнери створюють виконувані програмні пакети, які абстраговані від ОС хоста, що дозволяє не залежити від неї. Це означає, що вони є дуже портативними і можуть однаково стабільно працювати на будь-якій платформі або хмарі.

Движок Docker Engine, на якому працюють контейнери, став галузевим стандартом для контейнерів, з простими інструментами для розробників і універсальним підходом до пакування, який працює як на Linux, так і на Windows. Зараз екосистема контейнерів переходить на движок, керований ініціативою відкритих контейнерів OCI (Open Container Initiative). Розробники можуть продовжувати використовувати інструменти та процеси Agile та DevOps для швидкої розробки та оптимізації додатків.

Контейнери не споживають зайвих ресурсів. Це підвищує ефективність роботи сервера та зменшує витрати на ліцензії та час завантаження, оскільки ОС не потрібно завантажувати.

Кожен контейнерний додаток ізольований і працює незалежно від інших додатків; якщо один контейнер виходить з ладу, це не впливає на роботу інших контейнерів. Команди розробників можуть виявляти і виправляти технічні проблеми в одному контейнері, не викликаючи простою в інших контейнерах. Механізм контейнерів може використовувати методи ізоляції безпеки ОС наприклад, контроль доступу SELinux для ізоляції несправностей всередині контейнерів.

ПЗ, що працює в контейнерному середовищі, використовує ядро ОС машини, а прикладний рівень всередині контейнера може бути спільним для всіх контейнерів. Контейнери мають меншу ємність і потребують менше часу для запуску, ніж віртуальні машини, тому можна використовувати більше контейнерів з тією ж обчислювальною потужністю, що й одна ВМ.

Платформи оркестрування контейнерів автоматизують розгортання, масштабування та керування контейнерними робочими навантаженнями та сервісами. Ці платформи спрощують такі задачі управління, як масштабування контейнерних додатків, розгортання нових версій додатків, моніторинг, ведення журналів і налагодження.

Ізоляція контейнеризованих додатків запобігає потраплянню шкідливого коду в інші контейнери та хост-системи. Можна визначити параметри безпеки, щоб автоматично запобігти потраплянню небажаних компонентів до контейнера або обмежити взаємодію з небажаними ресурсами.

1.1.3 Огляд контейнеризації та мікросервісної архітектури

Компанії-розробники впроваджують мікросервіси замість традиційної монолітної моделі як ефективний підхід до розробки та управління додатками. У мікросервісах складні додатки декомпозуються на низку менших спеціалізованих сервісів, кожен з яких має власну базу даних і бізнес-логіку. Мікросервіси взаємодіють один з одним через загальні інтерфейси API або REST-інтерфейси. Використання мікросервісів дозволяє командам розробників зосередитися на оновленні конкретних областей програми, не впливаючи на додаток в цілому, прискорюючи розробку, тестування і розгортання.

Концепції мікросервісів і контейнеризації схожі і по суті є методами розробки програмного забезпечення, які перетворюють додаток на набір невеликих сервісів або компонентів, які є портативними, масштабованими, ефективними і простими в управлінні.

Більше того, мікросервіси та контейнеризація йдуть пліч-о-пліч. Контейнери полегшують інкапсуляцію будь-якого додатку, будь то традиційний моноліт або модульний мікросервіс. Мікросервіси, розроблені в контейнерах, користуються всіма перевагами контейнеризації:

- гнучкість для розробників;
- ізоляція від збоїв;
- ефективність роботи сервера;
- автоматизація встановлення, масштабування та управління;
- рівні безпеки.

Поєднання контейнерів, мікросервісів виводить розробку і доставку додатків на новий рівень, якого неможливо досягти за допомогою традиційних методологій і середовищ. Ці підходи нового покоління підвищують гнучкість, ефективність, надійність і безпеку життєвого циклу розробки, що дозволяє

пришвидшити доставку додатків і покращити досвід кінцевих користувачів і ринку.

1.1.4 Огляд безпеки контейнерів

Контейнерні програми мають певний рівень безпеки, оскільки вони можуть запускатися як ізольовані процеси і працювати незалежно від інших контейнерів. Така ізоляція запобігає впливу шкідливого коду на інші контейнери або потраплянню до хост-системи. Однак прикладний рівень всередині контейнера зазвичай є спільним для всіх контейнерів. Це позитивно з точки зору ефективності використання ресурсів, але також створює можливості для втручання в поведінку контейнера і порушення безпеки. Те ж саме стосується і спільних операційних систем, оскільки декілька контейнерів можуть бути пов'язані з однією і тією ж операційною системою хоста. Загроза безпеці спільної операційної системи може походити від усіх контейнерів, пов'язаних з нею, і навпаки, а порушення безпеки одного контейнера може призвести до порушення безпеки хостової операційної системи.

Щодо самих контейнерів то існують додатки з відкритим вихідним кодом або компоненти, упаковані в контейнери, які можуть підвищити безпеку, і постачальники контейнерних технологій, такі як Docker, продовжують активно вирішувати питання безпеки контейнерів. Вони застосовують підхід «безпечний за замовчуванням» до контейнеризації, вважаючи, що безпека повинна бути властивістю платформи, а не окремо розгорнутого і налаштовуваного рішення. З цією метою контейнерне переміщення підтримує всі функції ізоляції за замовчуванням, наявні в базовій операційній системі. Угрози безпеки можуть бути визначені для автоматичного блокування небажаних компонентів від доступу до контейнера або для обмеження взаємодії з небажаними ресурсами.

Наприклад, у просторі імен Linux кожному контейнеру можна надати ізольоване представлення системи, наприклад

– мережева взаємодія
– ідентифікатори процесів;
– ідентифікатори користувачів;
– міжпроцесні взаємодії;
– настройки імені хоста.

Простори імен можна використовувати для обмеження доступу до цих ресурсів через процеси всередині кожного контейнера. Як правило, непідтримувані підсистеми та простори імен не можуть бути доступні з контейнера. Адміністратори можуть легко створювати та керувати цими «обмеженнями ізоляції» для кожного контейнерного додатку за допомогою простого користувацького інтерфейсу.

1.2 Порівняння контейнеризації та віртуалізації

Контейнери часто порівнюють з віртуальними машинами. Це пов'язано з тим, що ці технології можуть значно підвищити ефективність обчислень, дозволяючи декільком типам програмного забезпечення працювати в одному середовищі. Однак, як показує практика, контейнерна технологія має значні переваги над віртуалізацією, і IT-фахівці часто віддають перевагу саме їй.

Технологія віртуалізації дозволяє декільком операційним системам і програмним додаткам одночасно використовувати ресурси одного фізичного комп'ютера. Наприклад, IT-організація може використовувати як Windows, так і Linux, або кілька версій операційної системи та кілька додатків на одному сервері. Кожна програма та пов'язані з нею файли, бібліотеки та залежності

включно з копією ОС упаковуються у віртуальну машину. Використання декількох віртуальних машин на одному фізичному комп'ютері може значно зменшити операційні та енергетичні витрати.

З іншого боку, контейнеризація дозволяє ще ефективніше використовувати обмежені ресурси. Контейнери утворюють єдиний повний пакет, який поєднує код програми з усіма пов'язаними конфігураційними файлами, бібліотеками та залежностями, необхідними для її запуску. Однак, на відміну від віртуальних машин, контейнери не містять копії операційної системи. Замість цього, середовище виконання контейнера встановлюється в операційній системі хост-системи, що забезпечує канал для всіх контейнерів в обчислювальній системі для спільного використання однієї і тієї ж ОС.

Як згадувалося раніше, контейнери часто називають «легкими». Оскільки вони мають спільне ядро ОС машини, вони не потребують додаткових ресурсів, щоб пов'язати операційну систему з кожним додатком, як у випадку з віртуальними машинами. Оскільки інші рівні контейнерів також можуть бути спільними для декількох контейнерів, ємність контейнерів, природно, менша, і вони завантажуються швидше, ніж віртуальні машини. У той же час, кілька контейнерів можуть працювати з тією ж критичною потужністю, що і одна віртуальна машина, що більше підвищує ефективність сервера і знижує витрати.

1.3 Огляд та порівняння інструментальних засобів для розгортання оточення програмного забезпечення засобами контейнеризації

Одним з найпопулярніших інструментів контейнеризації є Docker. Docker надає зручний інтерфейс для створення, управління та розгортання контейнерів, що робить його вибором багатьох розробників та інженерів.

Docker забезпечує ізольоване середовище для додатків та їх залежностей, дозволяє їм працювати незалежно від конкретної ОС [6].

Podman – це інструмент контейнеризації, який пропонує альтернативу Docker, надаючи безпечніше та більш ефективне управління контейнерами. Podman дозволяє запускати контейнери без привілейованих прав доступу (rootless), що покращує безпеку та зменшує ризики експлуатації. Однією з його ключових переваг є відсутність необхідності запускати daemon thread, що робить його більш привабливим для деяких випадків використання [7].

Крім того, у сучасному IT-світі важливу роль відіграють системи оркестрації контейнерів на основі процесу організації декількох контейнерів на мережевому рівні, наприклад, Kubernetes. Kubernetes дозволяє автоматизувати управління контейнерами у розподіленому середовищі, забезпечуючи високу доступність та надійність додатків [8].

Одним із головних критеріїв порівняння контейнерів є зручність використання та ступінь складності конфігурації.

З цього погляду Docker відзначається своєю простотою та інтуїтивністю, його команди тривіальні. Наявність додатку Docker Desktop дозволяє працювати з контейнерами не лише з консолі, а і зі зручного графічного інтерфейсу, в якому можна запускати контейнери, керувати ними, створювати образи, налаштовувати мережу і т.ін.

У той же час, Podman також має свій графічний інтерфейс Podman Desktop для керування та моніторингу контейнерів, його команди теж нативні і налаштування прав доступу самі прості з даної лінійки технологій.

Kubernetes поступається своїм конкурентам в питаннях простоти налаштування конфігурацій. Ця технологія має настільки багато концепцій та можливостей, що моніторити та конфігурувати їх стає дуже важко навіть для досвідчених користувачів.

Також важливим аспектом порівняння є ефективність та продуктивність. Docker та Kubernetes показують найкращі та схожі результати. Обидва інструменти дозволяють ефективно використовувати ресурси та забезпечують

надійність у роботі додатків. Варто відзначити, що Kubernetes може бути більш потужним у випадках, коли необхідно масштабувати додатки на великі кластери.

Вибір найкращого методу контейнеризації залежить від конкретних вимог проєкту, його масштабу та характеристик інфраструктури. Наприклад, для проєктів критичної галузі, з високим рівнем безпекових вимог та вимог ізоляції контейнерів, проєктів, які вимагають розширених можливостей управління, Podman буде кращим вибором. Для простих проєктів використання Docker буде кращим рішенням.

Kubernetes буде ідеальним вибором для складних розподілених систем з великими обсягами робіт та вимогами до масштабованості. Його можливість автоматично масштабувати контейнери, резервно копіювати їх та моніторити робить Kubernetes відмінним вибором для складних проєктів з високими вимогами до надійності та продуктивності.

При виборі методу контейнеризації важливо враховувати рівень складності проєкту, безпекові вимоги, масштабованість, ресурси, наявність експертної підтримки, активності спільноти. Вибір методу контейнеризації важливо здійснювати на основі конкретних характеристик проєкту, а також з урахуванням його масштабів та особливостей інфраструктури [9].

1.4 Аналіз оточення розгортання програмного забезпечення

Розгортання ПЗ середовище – це комп'ютерна система, в якій поширюється і виконується комп'ютерна програма або програмний компонент. У найпростішому випадку розгортання і безпосереднє виконання програми на одному комп'ютері може бути реалізоване в одному середовищі, але при промисловій розробці розрізняють середовище розробки (development), виробниче середовище (production) та проміжне середовище (stage) Цей

структурований процес управління випуском включає розгортання, тестування та відкат у разі виникнення проблем [10].

Масштаб середовища дуже різниться. Середовище розгортання – це, як правило, одна робоча станція розробника, в той час як виробниче середовище – це мережа з багатьох географічно розподілених машин у випадку центру обробки даних або ВМ у випадку хмарного рішення. Код, дані та конфігурація можуть розгортатися паралельно без необхідності підключення до відповідного рівня.

1.4.1 Архітектури IT-проєктів

Архітектури розгортання широко варіюються, але загалом ієрархія починається фази development і закінчується фазою production Типова чотирирівнева архітектура – це каскад рівнів розгортання, тестування, моделювання та виробництва: development, testing, staging, production. ПЗ розгортається на кожному рівні по черзі. Інші поширені середовища включають QC (quality control) для приймального тестування, пісочниці або експериментальні середовища для експериментів, які не призначені для впровадження у виробництво, а DR (disaster recovery) для негайного переходу до резервного варіанту в разі виникнення проблем у виробничому середовищі [11].

Такий розподіл особливо підходить для серверних додатків, коли сервери розташовані у віддалених центрах обробки даних. Для коду, що виконується на кінцевих точках користувача, таких як додатки або клієнти, останній рівень називається користувацьким або локальним середовищем.

Точні визначення і межі між середовищами варіюються. Testing environment вважається частиною development environment, тоді як stage environment вважається частиною testing environment. Основні ієрархії

обробляються в певному порядку, і з новими релізами проштовхуються або виштовхуються в кожну з рівнів. Експериментальні релізи – це фінальні релізи, в той час як recovery релізи – це старіші версії або копії production environment, які зазвичай розгортаються після виробничого релізної фази прокту. У разі виникнення проблем, старі версії можна відкотити назад, і в більшості випадків старі версії стають доступними так само, як і нові версії. Останній крок, push to prod, є найбільш чутливим, оскільки будь-які проблеми на ньому безпосередньо впливають на користувачів. Тому він зазвичай управляється по-іншому, але принаймні ретельніше контролюється, можливо, з відкатом або просто перехідним етапом [12].

Іноді розгортання відбувається поза звичайним процесом, щоб забезпечити термінові або незначні зміни, які не потребують повного релізу. Це може бути один патч, великий пакет оновлень або хотфікс.

Середовища можуть бути різного розміру. Local environment– це зазвичай окремі комп'ютери розробників, тоді як production environment може складатися з тисяч географічно розподілених комп'ютерів. Testing і stage environments можуть бути невеликими або великими, залежно від наданих ресурсів, а забезпечення може варіюватися від однієї машини до повної реплікації production environment.

1.4.2 Огляд development environment

Development environment – це середовище, в якому розробляється ПЗ, зазвичай це лише комп'ютер розробника. Development environment багато в чому відрізняється від кінцевого цільового середовища. Це може бути не ПК, смартфони, вбудовані системи, безпілотні автомобілі у центрах обробки даних, і навіть якщо це ПК, середовище розробника може включати різні або додаткові

версії компіляторів, IDE, бібліотек та допоміжних випусків ПЗ тощо, а також інші інструменти розробника.

У контексті контролю ревізій робиться більш тонке розмежування, особливо там, де задіяно більше ніж один розробник. Розробники мають робочу копію вихідного коду на власних машинах, а зміни вносяться до репозиторію і позначаються як гілка, залежно від методології розробки. Середовище на одній робочій машині, де вносяться і тестуються зміни, називається local environment або пісочницею. Створення копії вихідного коду в репозиторії в чистому середовищі є ще одним кроком інтеграції і це середовище може називатися інтеграційним або середовищем розробника. На рівні вихідного коду концепція внесення змін до сховища з подальшим створенням гілок відповідає переходу від local environment до інтеграційного середовища Поганий реліз на цьому етапі означає, що зміни порушують збірку, а відкат релізу відповідає відкату всіх змін коду та наластувать оточення.

1.4.3 Огляд testing environment

Мета testing environment – дозволити тим, хто виконує тести, пропускати новий або змінений код через автоматизовані перевірки або неавтоматизовані методи. Після того, як розробники проведуть модульне тестування нового коду і конфігурацій в середовищі розробки, код переноситься в одне або кілька testing environment. Якщо тест не пройшов, testing environment може видалити несправний код з тестової платформи, зв'язатися з відповідальним розробником методом його нотифікації відповідним повідомленням і надати детальні журнали та результати тестів. Якщо всі тести пройшли успішно, тестове середовище або фреймворк безперервної інтеграції, який керував тестом, може автоматично перенести код до наступного середовища розгортання.

Для різних типів тестування потрібні різні типи testing environment, деякі або всі з яких можна віртуалізувати, щоб забезпечити швидке паралельне тестування. Наприклад, автоматизоване тестування UI можна виконувати на декількох віртуальних ОС і дисплеях (реальних або ВМ). Для тестування продуктивності можуть знадобитися нормалізовані базові апаратні конфігурації, щоб результати тестування продуктивності можна було порівнювати в часі. Тестування доступності та відмовостійкості може базуватися на віртуальному обладнанні або віртуальних симуляторах збоїв мережі.

Залежно від складності тестового середовища, тести можуть бути послідовними або паралельними. Важливою метою Agile та інших високопродуктивних методологій розробки ПЗ є скорочення часу CI/CD програмного забезпечення до виробництва. Високоавтоматизовані та розпаралелені testing environment роблять значний внесок у швидку розробку ПЗ.

1.4.4 Огляд stage environment

Stage environment – це тестове середовище, яке точно таке ж, як і production environment. Воно має якомога точніше відображати реальне виробниче середовище і може бути підключене до інших виробничих сервісів, БД та інши. Наприклад, сервер працює на віддаленій машині, а не локально, щоб перевірити вплив мережі на систему.

Основна мета stage environment – протестувати всі сценарії і процедури встановлення, конфігурації і міграції до того, як вони будуть застосовані у зкщвгсешщт environment. Це гарантує, що всі основні та другорядні оновлення виробничого середовища будуть виконані ефективно, без помилок і якомога швидше.

Іншим важливим застосуванням stage environment є тестування продуктивності (performance testing), тому що налаштування аппаратного забезпечення оточення напряму впливає на фактор продуктивності роботи ПЗ та системи сервера в цілому.

Деякі організації також використовують stage environment для попереднього перегляду нових функцій, які можуть бути обрані клієнтами, або для підтвердження інтеграції з існуючими версіями зовнішніх інтеграцій.

1.4.5 Огляд production environment

Production environment також відоме як live environment, оскільки це середовище, в якому безпосередньо працюють користувачі.

Розгортання у production environment є найбільш делікатним кроком. Він може бути реалізований шляхом прямого розгортання нового коду так званим перезаписом старого коду так, щоб у будь-який момент часу була доступна лише одна його копія або шляхом розгортання змін конфігурації. Також це може бути паралельне розгортання нової версії коду де реліна версія завантажується на певну ноду сервера, а бекап лишається на іншій не активній ноді.

Розгортання нової версії зазвичай вимагає перезапуску, якщо немає можливості гарячої заміни. Це вимагає або переривання роботи сервісу у користувацькому програмному забезпеченні, якщо додаток потрібно перезавантажити, або реплікації – шляхом поступового перезапуску екземпляра LB, або просто шляхом попереднього запуску нового сервера і перенаправлення трафіку на новий сервер. розгортання у production environment.

Під час розгортання нового випуску у виробничому середовищі, замість того, щоб розгортати його одразу на всіх екземплярах і користувачах,

розгортайте його спочатку на одному екземплярі або деяких користувачах, а потім на всіх екземплярах, або розгортайте його поетапно, щоб швидко реагувати у разі виникнення проблем. Це схоже на поетапне розгортання, за винятком того, що це робиться у production environment. Одночасний запуск декількох релізів збільшує складність і зазвичай має відбуватися швидко, щоб уникнути проблем із сумісністю.

1.5 Постановка задачі дослідження

Об’єктом дослідження в рамках кваліфікаційної роботи є процес розгортання оточення програмного забезпечення при виконанні ІТ-проєкту.

Предметом дослідження є процес розгортання оточення ПЗ в ІТ-проєкті з акцентом на УП.

Метою даної роботи є дослідження та аналіз сучасних методів розгортання оточення ПЗ за допомогою засобів контейнеризації в рамках ІТ-проєктів, з акцентом на ефективне УП.

Для досягнення мети, необхідно дослідити наступні задачі:

– методи розгортання оточення ІТ-проєктів;

– опис засобів контейнеризаії;

– типи архітектур ІТ-проєктів;

– планування проєкту;

– експериментальна оцінка впливу засобу контейнеризації на ресурсозатрати та ризики для первісного розгортання програмного забезпечення на сервері.

Науковими результата дослідження є досконалення методу розгортання інфраструктури мікросервмиісних eCommerce проєктів засобом Docker.

2. ДОСЛІДЖЕННЯ МЕТОДІВ КОНТЕЙНЕРИЗАЦІЇ

2.1 Дослідження ісінтрументу Docker

Docker – це відкрита платформа для розробки, доставки та запуску додатків. Docker відокремлює додатки від інфраструктури, дозволяючи розробникам швидше доставляти програмне забезпечення. Використовуючи методологію Docker для доставки, тестування та розгортання коду, можна значно скоротити затримки між написанням коду та переходом до виробництва.

Docker надає можливість пакувати та запускати програми у вільно ізольованому середовищі, яке називається контейнером. Ізоляція та безпека дозволяють одночасно запускати декілька контейнерів на одному хості. Контейнери містять все необхідне для запуску програми, тому розробникам не потрібно покладатися на те, що встановлено на хості. Контейнерами можна ділитися під час запуску, гарантуючи, що кожен, користувач, отримає той самий контейнер, який працює однаково.

Docker надає інструменти та платформу для управління життєвим циклом контейнерів. Контейнери використовуються для розробки додатків та їхніх допоміжних компонентів. Контейнери - це одиниці, які використовуються для розгортання та тестування додатку. Коли додаток готовий, він розгортається у production environment як контейнер або сервіс оркестрування. Це працює однаково, незалежно від того, чи є production environment локальним центром обробки даних, хмарним провайдером або гібридо.

Docker спрощує життєвий цикл розробки, дозволяючи розробникам працювати у local environment з використанням нативних контейнерів, які доставляють додатки та сервіси. Контейнери ідеально підходять для робочих процесів безперервної інтеграції та безперервної доставки (CI/CD) [13].

Розглянемо наступний приклад сценарію:

1) розробник пише код локально і ділиться своєю роботою з колегами, використовуючи контейнер Docker;

2) використовуючи Docker, розробники відправляють свій код до testing environment та проводять автоматизоване та мануальне тестування;

3) якщо розробник знаходить помилку, він виправляє її в local environment і повторно відправляє до testing environment для тестування і перевірки;

4) коли тестування завершено, розробник може відправити оновлений образ до клієнта у production environment.

Контейнерна платформа Docker дозволяє виконувати робочі навантаження застосунка з високою переносимістю. Контейнери Docker можуть працювати в різних середовищах, включаючи локальний ноутбук розробника, фізичну або віртуальну машину в центрі обробки даних або в хмарних провайдерах.

Портативність і легкість Docker також полегшують динамічне керування робочими навантаженнями, дозволяючи додаткам і сервісам масштабуватися майже в реальному часі відповідно до потреб бізнесу.

Docker забезпечує життєздатну та економічно ефективну альтернативу віртуальним машинам на базі гіпервізора, дозволяючи виділяти більше серверних потужностей для досягнення бізнес-цілей. Docker ідеально підходить не тільки для середовищ з високою щільністю, але й для малих і середніх розгортань, які потребують великоку кількість ресурсів.

Попередником контейнерів Docker були віртуальні машини. Як і контейнери, віртуальні машини ізолюють програми та їхні залежності від зовнішнього середовища. Однак Docker-контейнери мають переваги перед віртуальними машинами. Наприклад, вони споживають менше ресурсів, їх дуже легко переміщувати та швидко запускати.

Контейнери Docker — це образи Docker. Образ контейнера Docker – це шаблон, який визначає вміст і параметри операційного середовища контейнерної системи. Образ містить усі файли та налаштування, необхідні для створення та запуску контейнера. Образ може включати операційну систему, програмне забезпечення, бібліотеки, залежності та інші компоненти, необхідні для запуску певної програми або сервісу.

Docker використовує архітектуру клієнт-сервер. Клієнт Docker взаємодіє з Docker daemon thread , який виконує роботу зі створення, запуску та розповсюдження контейнерів Docker. Клієнт та daemon thread Docker можуть працювати в одній системі, або віддалено. Клієнт та daemon thread Docker взаємодіють за допомогою REST API, через сокети UNIX або мережевий інтерфейс. Docker Compose – це ще один клієнт Docker, який дозволяє працювати з програмами, що складаються з набору контейнерів. На рисунку 2.1 приведено приклад архітектури Docker. На ньому зображено взаємодію Docker-клієнта, Docker-daemon та Docker-registry.

Docker-клієнт є основним способом взаємодії користувачів з докером. При використанні команд, клієнт відправляє ці команди до Docker-daemon, який їх виконує.

Docker-daemon прослуховує Docker API і керує такими об’єктами Docker, як контейнерами, мережею, зображеннями та volumes. Docker-daemon також може спілкуватися з іншими daemon thread для керування службами Docker.

Docker-registry зберігає зображення Docker. Docker Hub — це загальнодоступний реєстр, яким може користуватися будь-хто, і Docker за замовчуванням шукає зображення в Docker Hub.

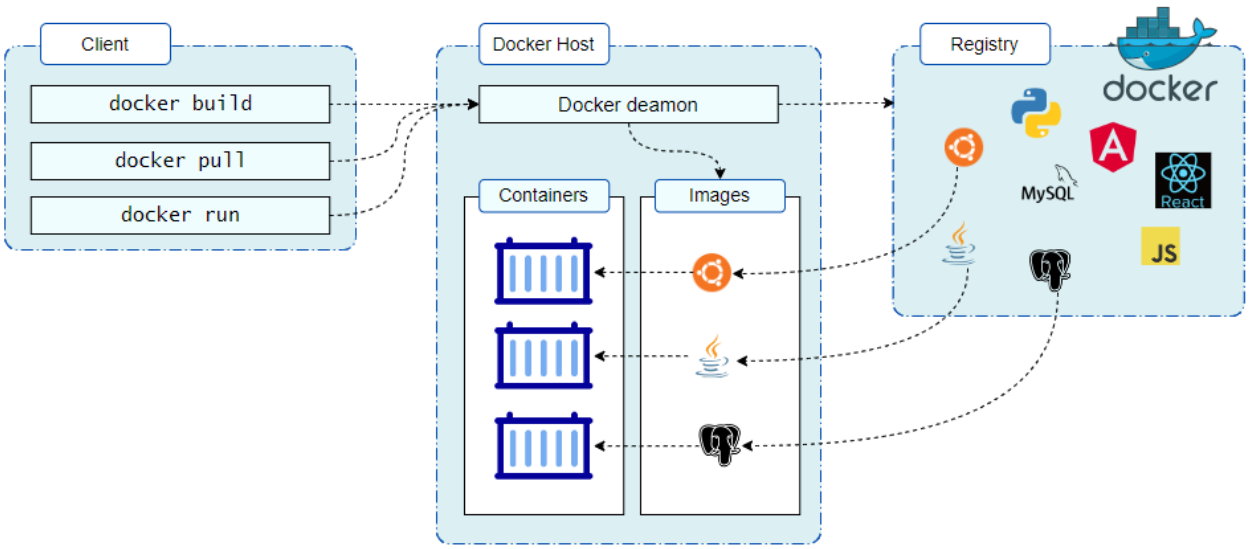

Рисунок 2.1 – Архітектура Docker

Образ (image) – це шаблон, доступний лише для читання, що містить інструкції про те, як створити Docker-контейнер. У більшості випадків образ базується на іншому образі з подальшими змінами. Наприклад, розробник може створити образ на основі образу ubuntu, який встановлює веб-сервер Apache, програму, що розробляється, та деталі конфігурації, необхідні для її запуску.

Можливо створювати власні образи або використовувати лише образи, створені іншими розробниками та опубліковані у Docker-registry. Щоб створити власний образ, потрібно створити Dockerfile з простим синтаксисом, який визначає кроки, необхідні для створення і запуску образу Кожна інструкція в Dockerfile створює шар образу Якщо Dockerfile змінюється і образ перебудовується, то перебудовуються тільки змінені шари. Це частина того, що робить образи легшими, меншими та швидшими, ніж інші технології віртуалізації.

Контейнер – це екземпляр виконуваного образу; який можна створювати, запускати, зупиняти, переміщувати або видаляти контейнери за допомогою Docker API або CLI. Контейнери також можуть підключатися до однієї або декількох мереж, додавати сховище і створювати нові образи на основі свого поточного стану.

За замовчуванням контейнери відносно добре ізольовані від інших контейнерів та хостів. Це дозволяє контролювати ступінь ізоляції мережі, сховища та інших важливих підсистем контейнера від інших контейнерів і хостів.

Контейнер ідентифікується за його образом і параметрами конфігурації, вказаними під час його створення або ініціалізації. Коли контейнер видаляється, всі зміни стану, не збережені в постійному сховищі, втрачаються.

Розглянемо ключові аспекти Docker Desktop, включаючи його механізми роботи, принципи функціонування та методи використання у сучасній розробці програмного забезпечення. Аналізуючи технічні деталі та можливості Docker Desktop. На рисунку 2.2 представлено графічний інтерфейс Docker Desktop.

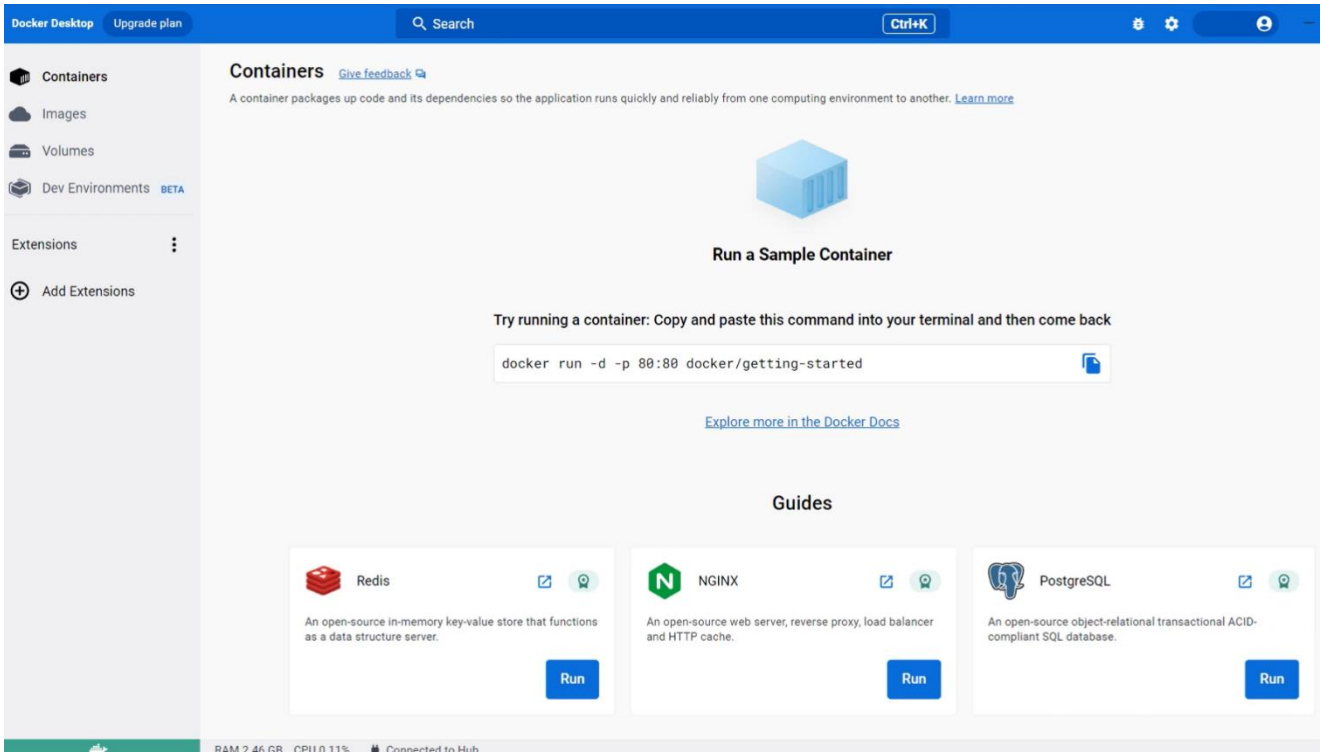

Рисунок 2.2 – Графічний інтерфейс Docker Desktop

На головній сторінці ми можемо побачити розділ з containers, images та volumes. Також застосунок нам пропонує запустити команду docker run -d -p 80:80 docker/getting-started для перевірки працездатності та правильності завантаження програмного забезпечення Docker. Також це можна зробити через UI. Цей образ містить технічну документацію, найкращі архітектурні практик що до використання Docker та ін. На рисунку 2.3 представлено етап пошуку образу контейнера docker/getting-started з DockerHub через UI застосунку Docker Desktop.

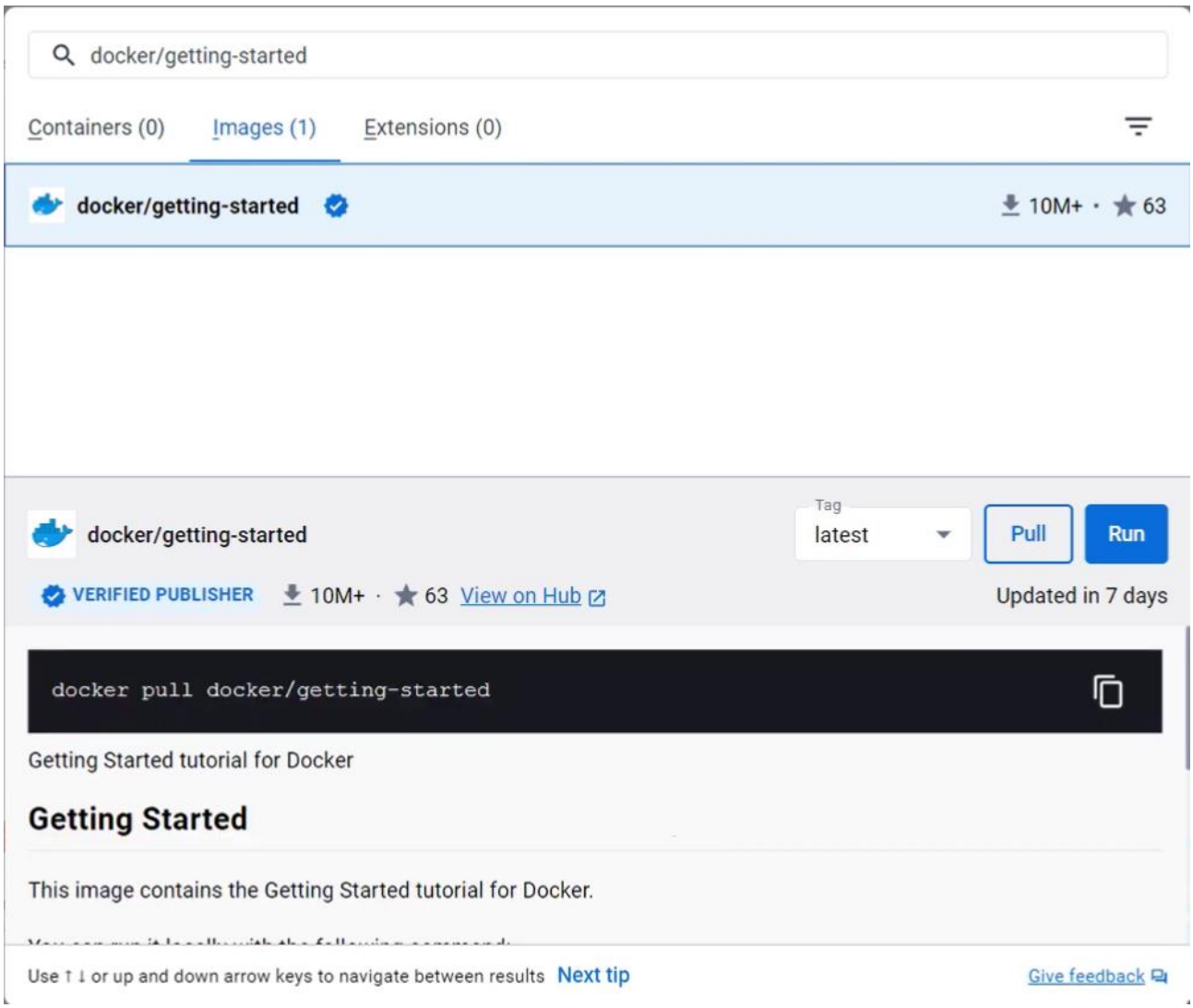

Рисунок 2.3 – Пошук образу контейнера docker/getting-started з DockerHub через UI застосунку Docker Desktop

На даній сторінці ми бачимо інструкцію до мануального розгортання образу за допомогою консолі (CLI) та створення нового контейнера з нього та альтернативний механізм за допомогою UI. Для цього обираємо версію образу у випадаючому списку та натискаємо на кнопку pull щоб завантажити образ у локальний репозиторій та бачимо спливаюче вікно з пропозицією створити новий контейнер з завантаженого образу docker/getting-started:latest. На рисунку 2.4 представлено спливаюче вікно створення нового контефнера.

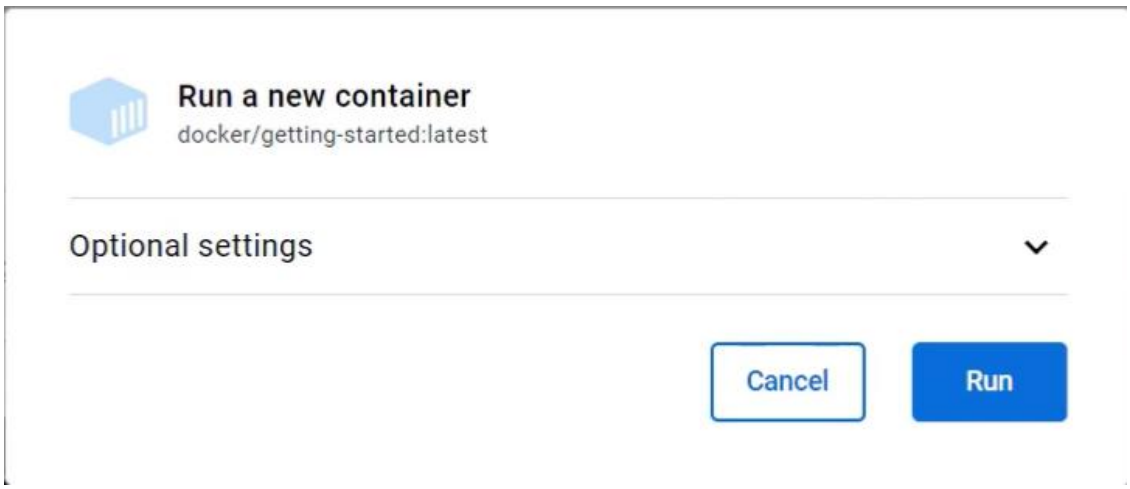

Рисунок 2.4 – Спливаюче вікно створення нового контефнера docker/getting-started:latest

По завершенню створення контейнера гомо можна побачити у розділі containers. На рисунку 2.5 представлено список контейнерів в локальному робочому середовищі.

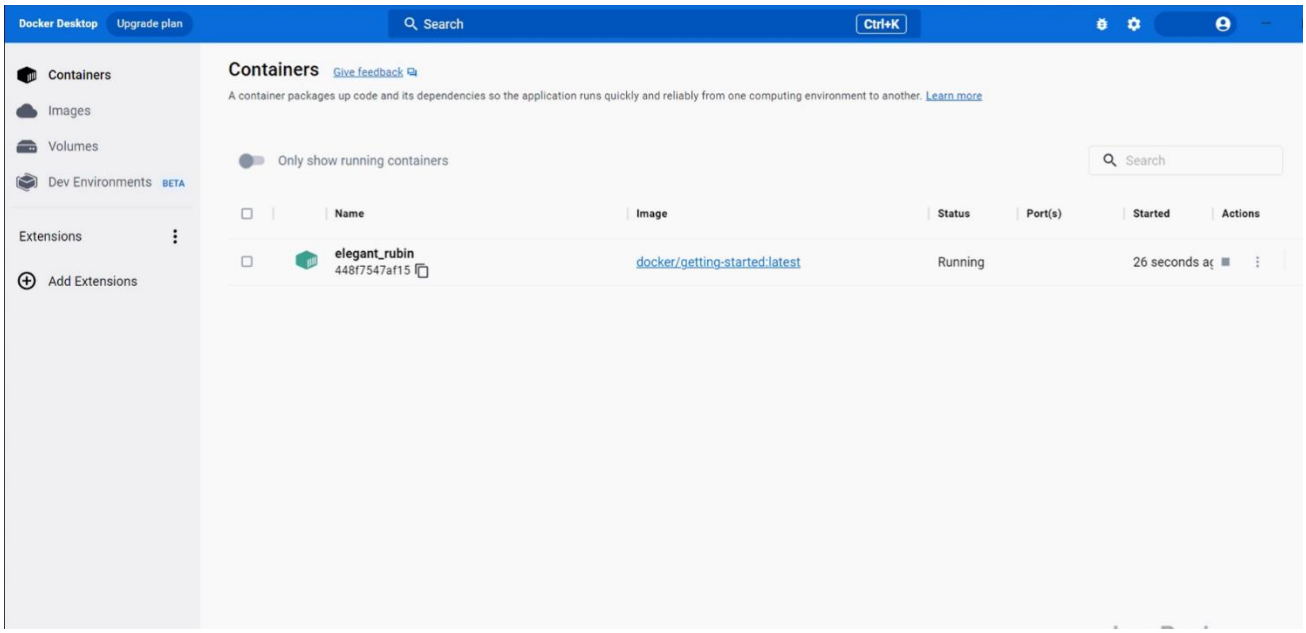

Рисунок 2.5 – Список контейнерів в локальному робочому середовищі

В даній списку контейнерів можна побачити щойно створений контейнер з образу docker/getting-started:latest. Йому призначився ідентифікаційний номер та згенерувалась назва так як при при створені контейнеру її не було вказано. Також у контейнера є статус життєвого циклу, відкриті порти та час створення. Натиснувши на контейнер можна побачити його логування у реальному часі, інспект зі змінними оточення, термінал для маніпуляцій з контейнером в мануальному режимі та метрики контейнера. На рисунках 2.6 – 2.10 показані таби з логуванням, інспектом, терміналом та метриками контейнера відповідно.

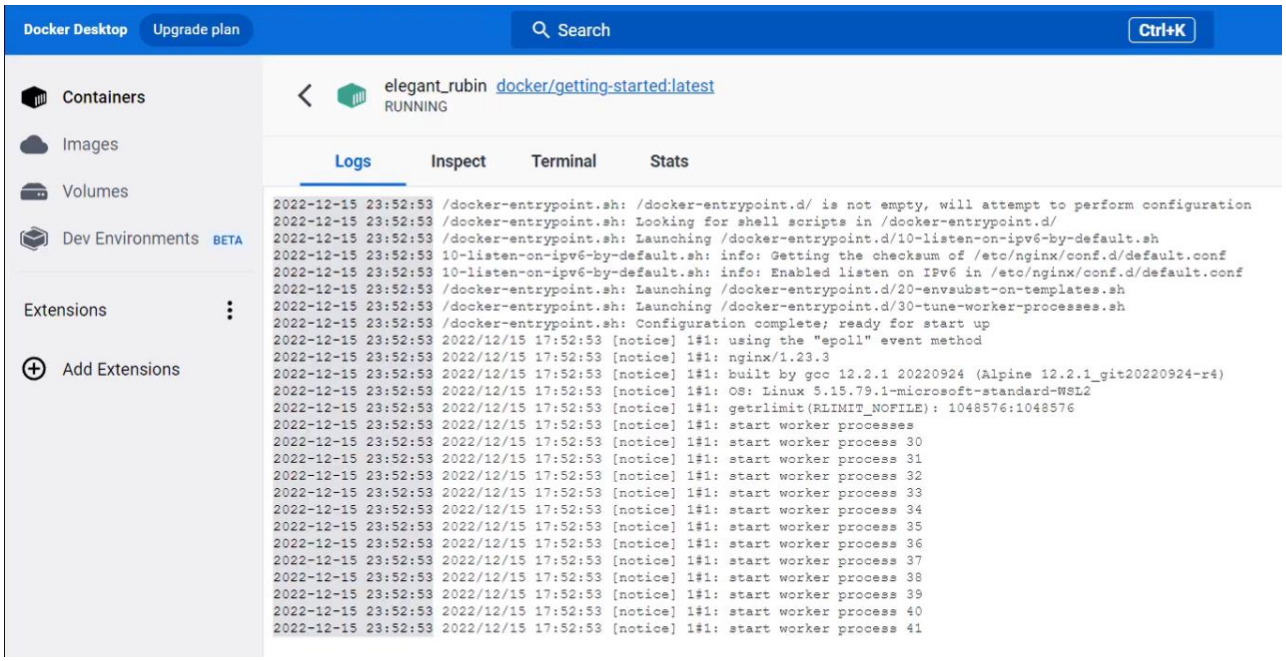

Рисунок 2.6 – Логування контейнера

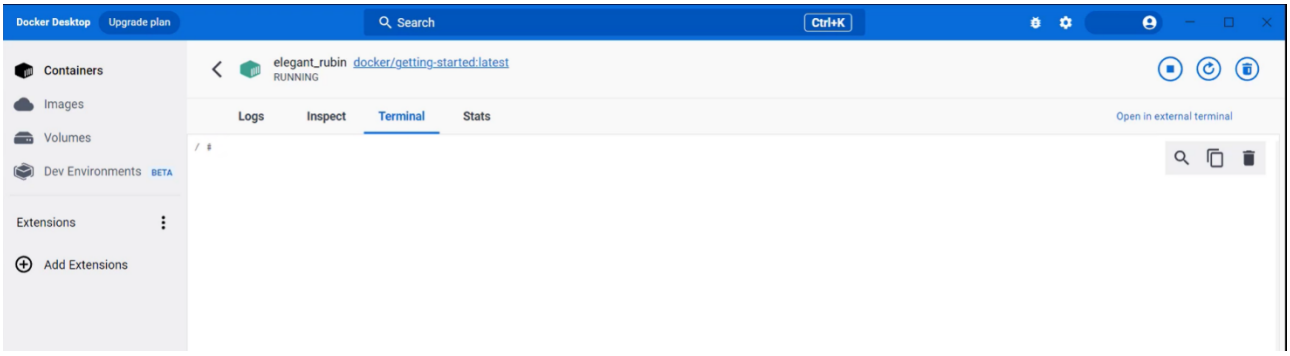

Рисунок 2.8 – Термінал контейнера

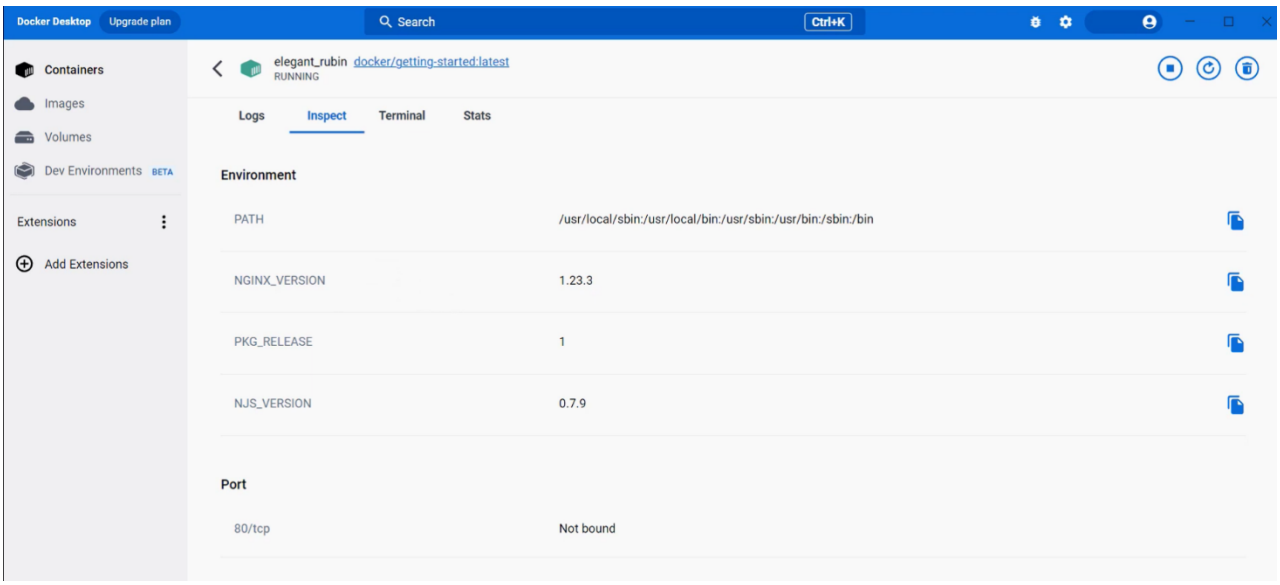

Рисунок 2.7 – Інспект контейнера

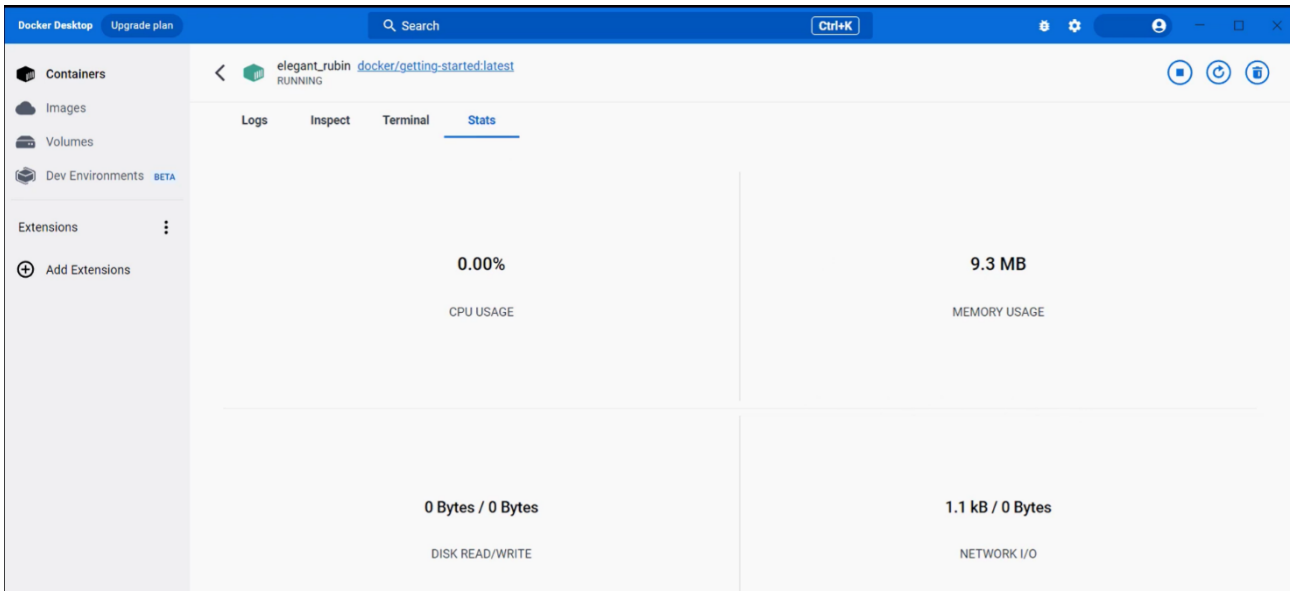

Рисунок 2.9 – Метрики контейнера

3. УДОСКОНАЛЕННЯ МЕТОДУ РОЗГОРТАННЯ ОТОЧЕННЯ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ ПРИ ВИКОНАННІ E-COMMERCE ПРОЄКТУ З МІКРОСЕРВІСНОЮ АРХІТЕКТУРОЮ ЗАСОБОМ DOCKER

Оскільки сучасні проєкти електронної комерції вимагають високої гнучкості, масштабованості та швидкості розгортання, архітектура мікросервісів та контейнеризація за допомогою Docker є ключовими елементами розробки. Однак, такі eCommerce рішення як IBM WebSphere Commerce, SAP Commerce Cloud (Hybris), Oracle ATG, Adobe Experience Manager Management Services (AEM Managed Services), Salesforce Commerce Cloud та багато інших відомих платформ електронної комерції не пропонують повноцінних мікросервісних рішень з використанням Docker [14].

Ці платформи спочатку були розроблені як монолітні системи або мають обмежену підтримку мікросервісів і контейнеризації IBM WebSphere Commerce і SAP Commerce Cloud зробили кроки до архітектури мікросервісів Oracle ATG спочатку була розроблена як монолітна система, і перехід до мікросервісів і використання Docker вимагатиме значних зусиль. Хоча Adobe Experience Manager Managed Services підтримує мікросервіси, вона не пропонує повноцінного рішення для мікросервісів з використанням Docker; Salesforce Commerce Cloud, будучи SaaS рішенням, не вимагає традиційної контейнеризації та розгортання, що обмежує його використання в рамках мікросервісної архітектури з Docker.

У цьому розділі розглядається, як Docker може покращити розгортання програмного середовища електронної комерції з архітектурою мікросервісів, подолати ці обмеження платформи і забезпечити більш ефективне, гнучке і масштабоване рішення.

3.1 Мікросервісна архітектура в e-Commerce проєктах

Загальна архітектура платформи електронної комерції на основі архітектури мікросервісів включає мережу невеликих незалежних сервісів, які працюють разом для забезпечення повного життєвого циклу електронної комерції. Важливою особливістю цієї архітектури є те, що кожен сервіс відповідає лише за певну функціональну область, що спрощує розробку та підтримку, а також забезпечує високу масштабованість і доступність системи.

У такій архітектурі кожен компонент системи може бути реалізований як окремий мікросервіс, який взаємодіє з іншими сервісами через API. Кожен мікросервіс може бути розгорнутий на власному сервері або контейнері та мати власну базу даних і логіку бізнес-процесів. Наприклад, мікросервіс управління замовленнями може мати власну базу даних, що містить інформацію про замовлення, тоді як мікросервіс рекомендаційної системи може мати базу даних, що містить історію переглядів та покупок користувачів.

Ці мікросервіси взаємодіють через мережу API. Наприклад, коли створюється нове замовлення, сервіс управління замовленнями може зв'язатися з сервісом каталогу товарів, щоб перевірити наявність товару та отримати інформацію про нього, а сервіс обробки платежів може здійснити платіж. Це забезпечує гнучку і масштабовану систему, яка може легко адаптуватися до змін у бізнес-процесах і потребах користувачів [15].

Архітектура дозволяє окремим мікросервісам масштабуватися незалежно від їхнього навантаження, а це означає, що ресурси сервера можуть використовуватися ефективно. Крім того, кожен мікросервіс можна розгортати, тестувати та оновлювати незалежно від інших компонентів, що спрощує процес розробки та обслуговування системи. Такий підхід дозволяє створювати гнучкі, масштабовані та надійні платформи електронної комерції, які можуть ефективно реагувати на мінливі ринкові умови та вимоги користувачів.

Мікросервісна архітектура в проєктах електронної комерції має такі переваги:

Гнучкість і масштабованість. Завдяки тому, що вона розбита на менші компоненти, команди можуть самостійно розробляти, тестувати та впроваджувати окремі сервіси, швидко реагувати на зміну вимог та масштабувати окремі частини системи відповідно до навантаження.

Швидка доставка продукту. Мікросервіси можна розробляти і випускати незалежно від інших сервісів, що дозволяє швидше і безпечніше надавати нові функції.

Висока доступність і надійність; якщо мікросервіс стає недоступним або виходить з ладу, інші сервіси можуть продовжувати працювати без перерви, забезпечуючи високу доступність системи.

Технологічна різноманітність. Кожен мікросервіс може бути розроблений з використанням різних технологій і мов програмування, що дозволяє використовувати найбільш підходящий інструмент для конкретного завдання.

Зв'язок архітектури мікросервісів з основними компонентами платформи електронної комерції відображається в декомпозиції функцій на невеликі, автономні сервіси, кожен з яких відповідає за виконання певного завдання. Розглянемо детальніше взаємозв'язок між кожним компонентом і мікросервісами:

Каталог товарів відповідає за зберігання та обробку інформації про товари, їхні характеристики, ціни та наявність. Мікросервіс, що реалізує каталог товарів, може мати власну базу даних та API для отримання даних про товари. Він може надавати можливість пошуку, фільтрування та сортування товарів. Сервіс також може включати можливість додавання нових товарів, редагування існуючих товарів і видалення товарів з каталогу.

Кошик та оформлення замовлення відповідає за додавання товарів до кошика, управління замовленнями, оплату та доставку. Цей компонент може включати мікросервіси для обробки платежів, взаємодії з системами управління запасами та логістики, а також формування рахунків і квитанцій. Профілі

користувачів та управління обліковими записами забезпечують автентифікацію користувачів, управління персональними даними та налаштуваннями облікових записів. Мікросервіси цього компонента включають системи автентифікації та авторизації, управління персональними даними користувачів, а також сервіси для відновлення паролів і запитів на видалення облікових записів.

Аналітика та звітність відповідає за збір, аналіз та візуалізацію даних, пов'язаних з продажами, відвідуваністю сайту, конверсіями тощо. Мікросервіси цього компонента включають системи аналізу трафіку, відстеження покупок і взаємодії з користувачами, а також формування звітів і статистики.

Рекомендаційні системи аналізують поведінку користувачів і рекомендують продукти на основі їхніх інтересів та попередніх покупок. Мікросервіси в цьому компоненті включають системи, які аналізують дані користувачів і рекомендують продукти та послуги для персоналізованої реклами та пропозицій.

Система управління запасами та логістикою відповідає за облік запасів товарів на складі, управління поставками та оптимізацію процесу доставки. Мікросервіси в цьому компоненті можуть включати систему управління запасами, яка відстежує кількість товарів на складі та автоматично видає замовлення на поповнення запасів у разі потреби. Також можуть бути впроваджені сервіси, які взаємодіють з постачальниками та логістичними компаніями для організації ефективної доставки товарів клієнтам.

Система управління контентом забезпечує можливість додавання, редагування та видалення контенту, такого як описи товарів, зображення та відео на веб-сайті. Мікросервіси в цьому компоненті можуть включати системи управління контентом, які зберігають і керують інформацією про товари та їхні характеристики, а також сервіси для управління статичним і динамічним контентом веб-сайту.

Пошукові системи забезпечують можливість швидкого та ефективного пошуку продуктів за різними критеріями, такими як назва, категорія та характеристики. Мікросервіси цього компоненту включають систему індексації

та пошуку, яка забезпечує швидкий доступ до даних про товари та відповідність запитам користувачів.

Система управління поверненням та обміном обробляє запити клієнтів на повернення та обмін товарів. Мікросервіси цього компонента включають систему обробки повернень та обмінів, яка координує процес повернення товарів, повернення коштів та видачу кредитів на придбання інших товарів.

Послуги підтримки клієнтів надають можливість отримати допомогу, консультацію та вирішити проблеми за допомогою різних каналів зв'язку, включаючи чат, телефонну підтримку та електронну пошту. Мікросервіси в цьому компоненті можуть включати систему управління запитами клієнтів, яка швидко та ефективно реагує на запити користувачів через різні канали зв'язку.

Кожен з цих компонентів може бути реалізований у вигляді незалежних мікросервісів, що дозволяє гнучко масштабувати, розгортати та оновлювати окремі частини системи незалежно одна від одної.

3.2 Роль Docker в розгортанні мікросервісної архітектури

Архітектура мікросервісів набула значної популярності в сучасній розробці програмного забезпечення як методологія, в якій додаток складається з набору незалежних компонентів, кожен з яких виконує окрему функцію. Однак, ефективне розгортання та управління такими архітектурами вимагає вирішення низки складних завдань. У цьому контексті Docker як контейнерна платформа виходить на перший план і пропонує безліч переваг, які роблять її ідеальною для розгортання мікросервісів [16].

Docker забезпечує ізоляцію контейнери Docker надають середовище виконання, в якому додатки та їх залежності можуть працювати незалежно від інших компонентів системи. Це гарантує ізоляцію ресурсів і мінімізує

потенційні конфлікти між різними сервісами, що є важливим фактором забезпечення стабільної роботи мікросервісів.

Використання контейнерів і Docker образів забезпечує консистентність додатків. Це означає, що додаток працює однаково в будь-якому середовищі. Це усуває проблеми, пов'язані з різними конфігураціями серверів, і спрощує процес розгортання та управління додатками.

Docker має можливість до масштабування мікросервісів. Завдяки легкій природі контейнерів та їх можливостям горизонтального масштабування, Docker може ефективно керувати навантаженням додатків та забезпечувати високу доступність.

Docker спрощує процес розгортання та управління мікросервісами. Використовуючи такі інструменти, як Docker Compose і Container Orchestrator, розробники можуть автоматизувати процес розгортання і масштабування додатків, спрощуючи управління інфраструктурою і скорочуючи час, що витрачається на адміністративні завдання.

Docker є ефективним рішенням для розгортання мікросервісів, оскільки він може забезпечити ізоляцію, узгодженість, масштабованість і спрощене управління. Використання Docker в архітектурі мікросервісів дозволяє створювати гнучкі, масштабовані та надійні додатки, які відповідають сучасним вимогам бізнесу та розробки програмного забезпечення.

3.3 Ізоляція та незалежніть мікросервісів

В контексті мікросервісної архітектури додатків, ізоляція та незалежність сервісів має важливе значення для забезпечення стабільності, масштабованості та безпеки системи. У цьому контексті Docker, як один з провідних інструментів контейнеризації, відіграє ключову роль у забезпеченні ефективної ізоляції кожного мікросервісу та запобігає взаємодії між сервісами.

Кожен контейнер має власний ізольований простір, який включає файлові системи, мережеві ресурси, процеси та змінні оточення.

Одним з ключових механізмів забезпечення ізоляції в Docker є використання технологій ядра Linux, таких як групи контролю (cgroups) та простору імен (namespaces). Cgroups дозволяють обмежувати та керувати ресурсами, виділеними для кожного контейнера, такими як CPU, пам'ять та дисковий простір, що запобігає одному контейнеру від перевантаження системи та впливу на роботу інших контейнерів. Простір імен, у свою чергу, забезпечують ізоляцію процесів та мережевих ресурсів, що запобігає конфліктам та взаємному впливу між контейнерами.

Docker керує мережевими ресурсами для кожного контейнера, створюючи віртуальні мережеві інтерфейси і керуючи маршрутизацією трафіку між контейнерами. Це ізолює мережеві ресурси і запобігає можливим конфліктам портів або IP-адрес між службами.

Docker дозволяє визначати змінні середовища та параметри конфігурації для кожного контейнера, так що поведінку кожного контейнера можна налаштовувати незалежно один від одного. Це особливо важливо в середовищах розробки та розгортання, де параметри середовища потрібно змінювати, не впливаючи на інші сервіси.

В цілому, ці механізми забезпечують високий рівень ізоляції для кожного мікросервісу і запобігають впливу одного сервісу на інші що забезпечує стабільність та надійність роботи.

3.4 Портативність та консистентність оточення мікросервісів

Портативність означає, що кожен мікросервіс може бути упакований в окремий контейнер, що містить усі необхідні залежності та конфігурації. Це дає змогу переносити мікросервіси між різними середовищами виконання, наприклад середовищами розроблення, тестування та виробництва, без будь-яких змін коду або конфігурації.

Консистентність середовища забезпечується тим, що контейнер Docker надає ізольоване та незалежне середовище виконання для кожного мікросервісу. Це означає, що незалежно від того, де запущено мікросервіс - на локальній машині розробника або на виробничому сервері, - його оточення завжди буде ідентичним, забезпечуючи узгоджену поведінку застосунку на всіх етапах розроблення та виробництва.

Завдяки портативності Docker розробники можуть створювати й тестувати мікросервіси в тому ж локальному середовищі, що й у виробничому. Контейнеризація спрощує розгортання нових версій мікросервісів і забезпечує надійну та передбачувану поведінку додатків у будь-якому середовищі.

3.5 Спрощенне маштабування мікросервісів e-Commerce платформ

Одним з інструментів, що забезпечує потужні можливості масштабування, є Docker Swarm. Docker Swarm – інструмент для розгортання та управління кластерами Docker-вузлів. Він дозволяє об'єднати кілька Docker-хостів і керувати ними як одним кластером. Основними компонентами Docker Swarm є вузли Manager Nodes і Worker Nodes, які забезпечують управління та виконання контейнерів відповідно [17].

Docker Swarm може масштабувати мікросервіси за допомогою горизонтального маштабування, вертикального маштабування та збалансованого трафіка.

За допомогою Docker Swarm мікросервіси можна легко масштабувати горизонтально, додаючи та видаляючи екземпляри контейнерів. Це особливо корисно для електронної комерції, де навантаження може значно коливатися залежно від часу доби та сезонних факторів. Горизонтальне масштабування рівномірно розподіляє навантаження і забезпечує відмовостійкість.

На додаток до горизонтального масштабування, Docker Swarm також підтримує вертикальне масштабування, яке збільшує ресурси на кожному вузлі кластера. Це корисно для мікросервісів, які потребують великих обчислювальних ресурсів та ресурсів зберігання.

Для збалансованого трафіку Docker Swarm інтегрується з різними механізмами балансування навантаження, такими як вбудований RR балансировщик і сторонні рішення, для рівномірного розподілу трафіку між екземплярами мікросервісів. Його можна використовувати для рівномірного розподілу трафіку між екземплярами мікросервісів.

Docker Swarm має кілька важливих переваг. По-перше, ним легко керувати: Docker Swarm надає простий та інтуїтивно зрозумілий інтерфейс для управління кластерами контейнерів. Це дозволяє DevOps-інженерам і системним адміністраторам зосередитися на таких важливих завданнях, як розробка нових функцій і забезпечення безпеки, замість того, щоб витрачати час на конфігурацію та управління інфраструктурою.

Ключовою перевагою є відмовостійкість – Docker Swarm забезпечує високий ступінь відмовостійкості з можливістю горизонтального масштабування та автоматичного відновлення після збоїв. Це мінімізує час простою і забезпечує безперервність критично важливих сервісів у разі збою.

Гнучкість – ще одна важлива перевага. Docker Swarm дозволяє гнучко налаштовувати параметри масштабування, такі як кількість копій мікросервісів та ресурси, виділені для кожного контейнера. Це дозволяє кластерам швидко

адаптуватися до мінливих потреб бізнесу без необхідності переналаштування всієї системи.

Нарешті, Docker Swarm сприяє ефективному використанню ресурсів. Він оптимізує розподіл навантаження між вузлами кластера і автоматично масштабує додатки на основі їх поточного навантаження, найкращим чином використовуючи обчислювальні ресурси і знижуючи витрати.

3.6 Управління залежностями та конфігураціями мікросервісів

Для ефективного управління залежностями сервісів у мікросервісній архітектурі широко використовуються інструменти контейнеризації, такі як Docker.

Один з ключових інструментів у Docker – це Dockerfile. Dockerfile – це текстовий файл, що містить інструкції для побудови Docker-образу. В ньому можна визначити всі залежності та конфігураційні параметри необхідні для коректного функціонування сервісу. Наприклад, в Dockerfile можна вказати, які пакети або бібліотеки потрібно встановити у контейнері, які файли слід скопіювати до контейнера, а також які команди потрібно виконати для налаштування середовища [18]. На рисунку 3.1 зображено приклад Dockerfile.

Додатково, для управління багатьма контейнерами та їх залежностями використовується Docker Compose. Docker Compose дозволяє описати структуру всієї мікросервісної системи у YAML-файлі, включаючи всі потрібні сервіси, їх залежності та конфігураційні параметри. Наприклад, в файлі docker-compose.yml можна описати, які контейнери потрібно створити, як вони пов'язані між собою, та які порти потрібно прокинути для забезпечення зв'язку між сервісами. На рисунку 3.2 зображено приклад docker-compose.yml.

```dockerfile
# Використовуємо офіційний образ Tomcat з Docker Hub, побудований на Java 11
FROM tomcat:11-jdk11-adoptopenjdk-hotspot

  # Відкриваємо порти для Tomcat (порт 8080 для HTTP і порт 8009 для AJP)
EXPOSE 8080 8009

  # Копіюємо файли конфігурації та додатки у Tomcat
COPY ./config/* /usr/local/tomcat/conf/
COPY ./webapps/* /usr/local/tomcat/webapps/
```

Рисунок 3.1 – Приклад Dockerfile

```yaml
version: '3'

services:
  nginx:
    image: nginx:latest
    ports:
      - "80:80"
    volumes:
      - ./nginx/conf:/etc/nginx/conf.d
      - ./nginx/html:/usr/share/nginx/html
    depends_on:
      - nodejs

  nodejs:
    image: node:latest
    working_dir: /usr/src/app
    volumes:
      - ./nodejs:/usr/src/app
    environment:
      - NODE_ENV=production
    ports:
      - "3000:3000"
    depends_on:
      - postgres

  postgres:
    image: postgres:latest
    environment:
      POSTGRES_USER: user
      POSTGRES_PASSWORD: password
      POSTGRES_DB: mydatabase
    ports:
      - "5432:5432"
```

Рисунок 3.2 – Приклад docker-compose.yml

Приклад використання Dockerfile та Docker Compose для управління залежностями та конфігураціями мікросервісів може виглядати наступним чином. Нехай у нас є мікросервісна система, що складається з трьох сервісів:

веб-сервера, бази даних та служби аутентифікації.

У Dockerfile для веб-сервера ми можемо визначити, що потрібно встановити веб-сервер Nginx, скопіювати файли додатка до контейнера та налаштувати веб-сервер для їх обробки. У Dockerfile для бази даних можна вказати, які СУБД та драйвери потрібно встановити, а також налаштувати базу даних. У Dockerfile для служби аутентифікації можна встановити сервер Node.js, встановити необхідні залежності за допомогою npm та налаштувати нашу службу.

У файлі docker-compose.yml можна описати структуру мікросервісної системи, вказавши всі потрібні сервіси та їх залежності. Наприклад, визначити, що веб-сервер залежить від бази даних та служби аутентифікації. Також можна вказати, які порти мають бути прокинуті для зв'язку між сервісами та зовнішніми системами.

Таким чином, за допомогою Dockerfile та Docker Compose відбувається ефективне управління залежностями та конфігураціями мікросервісів у складних системах, забезпечуючи їх коректну роботу та легкість установки.

3.7 Докеризація кожного мікросервіса

Докеризація кожного мікросервісу платформи електронної комерції є важливим кроком у розвитку та обслуговуванні інфраструктури. Цей процес гарантує, що кожен компонент буде ізольований у власному контейнері.

Представимо, що платформа має кілька основних сервісів, таких як управління каталогом, обробка платежів та управління замовленнями. Створемо Docker-контейнер для кожного з цих сервісів.

Почнемо з управління каталогом. Для цього створимо Docker-файл, який визначає конфігурацію контейнера: У Dockerfile вкажемо базовий образ, наприклад, Ubuntu або Alpine, і встановіть необхідні залежності, такі як бази

даних або фреймворки. Потім код копіюємо в контейнер і налаштовуємо середовище виконання.

Аналогічний підхід можна використовувати для сервісу обробки платежів. Створимо Docker-файл, вкажемо базовий образ, встановимо залежності, скопіюємо код і налаштуємо середовище.

Так само зробимо для сервісу управління замовленнями та іншими сервісами. Створимо Docker-контейнер з відповідними конфігурацією та залежностями для кожного з них.

Після створення контейнера Docker організовує сервіси за допомогою Docker Compose, визначивши всі сервіси, залежності та параметри конфігурації у файлі docker-compose.yml. Це дозволяє легко керувати всією інфраструктурою за допомогою однієї команди.

Коли платформа працює з Docker Compose, кожен сервіс запускається у власному контейнері, забезпечуючи ізоляцію та надійність. Окремі сервіси можна розширювати за потреби, додаючи або видаляючи контейнери.

Безпека інфраструктури також важлива: Docker Security Scanning можна використовувати для виявлення вразливостей у контейнерах, а Docker Content Trust – для автентифікації зображень.

Загалом, докеризація кожного мікросервісу на платформі електронної комерції може зробити розробку, розгортання та оновлення додатків більш ефективними та надійними. Це дозволяє швидше реагувати на зміни та забезпечує високу доступність платформи.

3.8 Безпека мікросервісів в Docker контейнерах

Враховуючи чутливість даних клієнтів, фінансових транзакцій та персональних даних, в контексті електронної комерції важливо захищати мікросервіси за допомогою контейнерів Docker. У цьому контексті Docker

забезпечує необхідну ізоляцію ресурсів для запобігання витоку даних та несанкціонованого доступу. Важливо правильно налаштувати параметри безпеки, такі як обмеження прав доступу та використання інструментів контролю доступу.

Важливо постійно оновлювати образи Docker, щоб мінімізувати ризик вразливостей на платформах електронної комерції. Використання тільки перевірених джерел зображень також може зменшити ймовірність атаки. Однак безпека не обмежується Docker. Важливо також приділяти увагу автентифікації та авторизації користувачів і шифрувати конфіденційні дані під час передачі та зберігання.

Для ефективного захисту систем електронної комерції також рекомендується здійснювати моніторинг і аудит дій, щоб швидко реагувати на потенційні загрози. Контроль доступу до мережі відіграє важливу роль у безпеці мікросервісів, а API-шлюзи можуть забезпечити централізований контроль доступу до різних частин системи.

Системи контролю доступу, такі як RBAC, можуть надавати більш деталізовані права доступу для різних ролей користувачів на платформах електронної комерції. Крім того, можна використовувати суміші послуг, щоб додати додатковий рівень безпеки, наприклад, контроль доступу та шифрування трафіку між мікросервісами [19].

Захист платформи електронної комерції за допомогою контейнерів Docker вимагає комплексного підходу, який поєднує технологію Docker з методами автентифікації, авторизації, шифрування та контролю доступу.

3.9 Відмовостійкість та моніторинг

Існує кілька важливих аспектів відмовостійких принципів проєктування мікросервісів, які слід враховувати при розробці та розгортанні додатків у

середовищі Docker. Відмовостійкість – це здатність системи продовжувати працювати у випадку відмови або переривання роботи компонента. Забезпечення відмовостійкості є ще більш важливим в контексті мікросервісної архітектури, де додаток складається з багатьох незалежних сервісів.

Одним з основних принципів є ізоляція. Мікросервіси повинні бути незалежними та ізольованими, щоб відмова одного сервісу не призвела до відмови всієї системи. Це досягається завдяки використанню контейнеризації, наприклад, Docker, де кожен сервіс працює у власному контейнері з власним часом виконання та залежностями.

Також важливо враховувати горизонтальне масштабування. Мікросервіси повинні бути горизонтально масштабованими, щоб забезпечити рівномірний розподіл навантаження і зменшити ризик збою через перевантаження, а Docker полегшує горизонтальне масштабування, дозволяючи декільком екземплярам одного і того ж сервісу працювати на різних вузлах кластера Docker полегшує горизонтальне масштабування, дозволяючи декільком екземплярам одного і того ж сервісу працювати на різних вузлах кластера.

Крім того, важливо забезпечити механізм аварійного відновлення. Це включає використання оркестратора контейнерів Docker Swarm для автоматичного перезапуску сервісів у разі збою та використання відмовостійкого сховища для резервного копіювання даних.

Але забезпечення відмовостійкості – це лише частина рівняння. Важливо також впровадити механізми моніторингу, які дозволять швидко виявляти проблеми та вживати коригувальні дії, такі як Docker Stats API, який дозволяє Docker відстежувати використання ресурсів контейнерів, визначати стан контейнерів та автоматично перезапускати їх при виявленні проблеми.

Крім того, інструменти моніторингу на рівні додатків, такі як Prometheus, можна використовувати для збору та аналізу показників продуктивності та доступності. Це дозволяє швидко вирішувати проблеми і виправляти їх до того, як вони вплинуть на користувачів.

3.10 Перспективи застосування Docker в e-Commerce проєктах з мікросервісною архітектурою

Використання Docker у проєктах електронної комерції продовжуватиме зростати. Очікується, що для покращення розгортання та управління мікросервісами розробники покладатимуться на контейнеризацію додатків. Розвиток інструментів управління контейнерами та інтеграція з іншими технологіями, такими як машинне навчання та аналіз даних, ще більше розширить можливості платформ онлайн-комерції.

Важливим напрямком розвитку є зростаюча автоматизація процесів розробки та експлуатації. За допомогою Docker, DevOps команди можуть значно прискорити цикл розробки та випуску нових функцій. Це дозволяє їм швидко реагувати на мінливі вимоги ринку і швидше відповідати на запити клієнтів. В результаті компанії можуть надавати більш актуальні та інноваційні рішення і ставати більш конкурентоспроможними.

Іншим важливим аспектом є підвищення масштабованості та гнучкості. Особливо в електронній комерції, де робочі навантаження значно коливаються в періоди пікових продажів, Docker надає можливість швидко масштабувати ресурси відповідно до поточних потреб. Це запобігає перебоям і збиткам, спричиненим перевантаженням системи, та забезпечує високий рівень обслуговування клієнтів.

Інтеграція з хмарними платформами також відіграє важливу роль у розвитку Docker в електронній комерції: Найбільші хмарні провайдери, такі як AWS, GCP та Microsoft Azure, пропонують широку підтримку Docker. Це дозволяє компаніям легко розгортати контейнерні додатки в хмарі та керувати ними, а також користуватися перевагами глобальної інфраструктури, такими як висока доступність і географічна масштабованість.

У майбутньому очікується подальша інтеграція технологій Docker та IoT. Зі збільшенням кількості підключених пристроїв компанії електронної комерції

зможуть використовувати контейнери для управління та аналізу даних з різних джерел. Це створить нові можливості для персоналізації та покращення користувацького досвіду, що дозволить їм надавати клієнтам більш точні та релевантні пропозиції.

Розвиток штучного інтелекту та машинного навчання також має значний вплив на майбутнє Docker. Контейнеризація полегшує розгортання та управління складними моделями машинного навчання і покращує процес рекомендацій та прогнозування в електронній комерції. Це призведе до створення більш інтелектуальних та адаптивних платформ, які зможуть ефективно взаємодіяти з користувачами та пропонувати їм саме те, що вони шукають.

Крім того, концепція інверсії управління (IoC) стає все більш важливою в рамках використання Docker в архітектурах мікросервісів; IoC дозволяє розробникам створювати більш гнучкі і керовані системи, де компоненти можуть взаємодіяти незалежно один від одного IoC дозволяє розробникам створювати більш гнучкі і керовані системи, де компоненти можуть взаємодіяти незалежно один від одного. Це особливо актуально для проєктів електронної комерції, де швидкість і гнучкість змін є ключовими: За допомогою Інтернету речей та контейнеризації можна створювати динамічні та адаптивні системи, здатні швидко реагувати на зміни в поведінці користувачів та ринкових умовах.

Отже, використання Docker в мікросервісних архітектурах в проєктах електронної комерції обіцяє значні переваги та можливості для подальшого розвитку. Враховуючи сучасні тенденції та технології, можна з упевненістю сказати, що контейнеризація є основою для створення більш гнучких, масштабованих і надійних платформ для онлайн-торгівлі, здатних задовольнити зростаючі потреби сучасного ринку.

4. УДОСКОНАЛЕННЯ МЕТОДУ РОЗГОРМАННЯ ІНФРАСТРУКТУРИ МІКРОСЕРВІСНОГО E-COMMERCE ПРОЄКТУ ЗАСОБОМ DOCKER

4.1 Підготовка до розгортання застосунку

Етап підготовки до розгортання мікросервісного eCommerce проєкту засобами Docker включає в себе аналіз вимог до середовища, налаштування необхідних інструментів і ознайомлення з архітектурою проєкту.

Перед початком роботи необхідно забезпечити відповідне середовище для розгортання мікросервісного проєкту.

Docker може працювати на різних операційних системах, але для серверних середовищ ми будемо використовувати Linux, оскільки він забезпечує кращу продуктивність і сумісність.

Для ефективної роботи Docker необхідно мати достатню кількість ресурсів, таких як CPU, RAM та дисковий простір. Наприклад, для невеликого проєкту може бути достатньо 4 ядра CPU і 32 GB RAM, але для більш складних систем може знадобитися значно більше ресурсів.

Серед основного програмного забезпечення можна виділити 3 найважливіших: Docker, Docker Compose та Docker Swarm.

Docker – основний інструмент для створення, розгортання та керування контейнерами. Рекомендується використовувати останню стабільну версію.

Docker Compose – інструмент для визначення і запуску багатоконтейнерних Docker додатків. Він дозволяє описувати конфігурації всіх сервісів, необхідних для роботи додатку, в одному файлі.

Docker Swarm – оркестратор контейнерів, вбудований в Docker. Він використовується для автоматизації розгортання, масштабування та керування контейнеризованими додатками. Забезпечує високий рівень масштабованості та надійності.

Потрібно забезпечити правильні конфігурації мережевої інфраструктури, включаючи налаштування брандмауера, правил маршрутизації та балансування навантаження, є важливим аспектом підготовки до розгортання. Це дозволить забезпечити стабільний доступ до сервісів і мінімізувати затримки в мережі.

Розуміння архітектури eCommerce проєкту є ключовим для успішного розгортання. В таблиці 4.1 описано компоненти архітектури мікросервісного eCommerce проєкту що буде розгортатись.

Таблиця 4.1 – Опис компонентів архітектури мікросервісного eCommerce проєкту

Компонент	Опис
Frontend	Веб-інтерфейс, який взаємодіє з користувачами. Він може бути реалізований за допомогою різних фреймворків, таких як Node.js. У середовищі Docker фронтенд зазвичай працює в окремому контейнері
Backend API	Серверна частина, яка обробляє запити від фронтенду і взаємодіє з іншими сервісами та базами даних. Для розробки бекенду використано мову програмування Java та її фреймворки
Бази даних	Зберігання даних проєкту. Найчастіше використовуються реляційні бази даних, такі як PostgreSQL. Кожна база даних зазвичай розгортається в окремому контейнері.

В таблиці 4.2 описано мікросервіси які виконують специфічні функції в рамках проєкту.

Таблиця 4.2 – Опис мікросервісів що будуть розгнортатись.

Назва мікросервісу	Опис
User Service	Управління користувачами (реєстрація, авторизація, профілі). Використовує окрему базу даних для зберігання інформації про користувачів.
Product Service	Управління товарами (створення, редагування, видалення товарів, категорії). Має свою власну базу даних для зберігання даних про товари.
Order Service	Обробка замовлень (створення замовлень, статуси замовлень, історія замовлень). Використовує окрему базу даних для зберігання інформації про замовлення.
Payment Service	Управління платежами (інтеграція з платіжними системами, обробка транзакцій). має власну базу даних для зберігання транзакцій.
Inventory Service	управління запасами (відстеження наявності товарів на складах). використовує свою базу даних для зберігання інформації про запаси.
Shipping Service	Управління доставкою (обробка заявок на доставку, відстеження посилок). має власну базу даних для зберігання даних про доставку.
Promotion Service	Управління акціями та знижками (створення, редагування та застосування промоакцій). Має власну базу даних для зберігання інформації про акції.

Балансувальник навантаження – компонент, який розподіляє вхідні запити між різними сервісами для забезпечення рівномірного навантаження і високої доступності. У нашому середовищі Docker це буде реалізовано за допомогою Docker Swarm.

4.2 Розгортання мікросервісного eCommerce проєкту з використанням Docker

Для початку розгортання, нам потрібно встановити Docker на Linux. На рисунку 4.1 зображені команди для розгортання Docker на Linux ОС.

docker_install.sh

```
# Оновлення списку пакетів та інсталяція необхідних залежностей
sudo apt-get update
sudo apt-get install apt-transport-https ca-certificates curl software-properties-common

# Додавання офіційного GPG ключа Docker
curl -fsSL https://download.docker.com/linux/ubuntu/gpg | sudo apt-key add -

# Додавання Docker репозиторію до APT
sudo add-apt-repository "deb [arch=amd64] https://download.docker.com/linux/ubuntu $(lsb_release -cs) stable"

# Оновлення списку пакетів та інсталяція Docker
sudo apt-get update
sudo apt-get install docker-ce

# Перевірка встановлення Docker
docker --version
```

Рисунок 4.1 – Команди для розгортання Docker на Linux ОС.

На рисунках 4.2 та 4.3 зображено CLI з успішно встановленим Docker та з успішно розгорнутим Docker Toolbox в якому запустився Docker daemon відповідно.

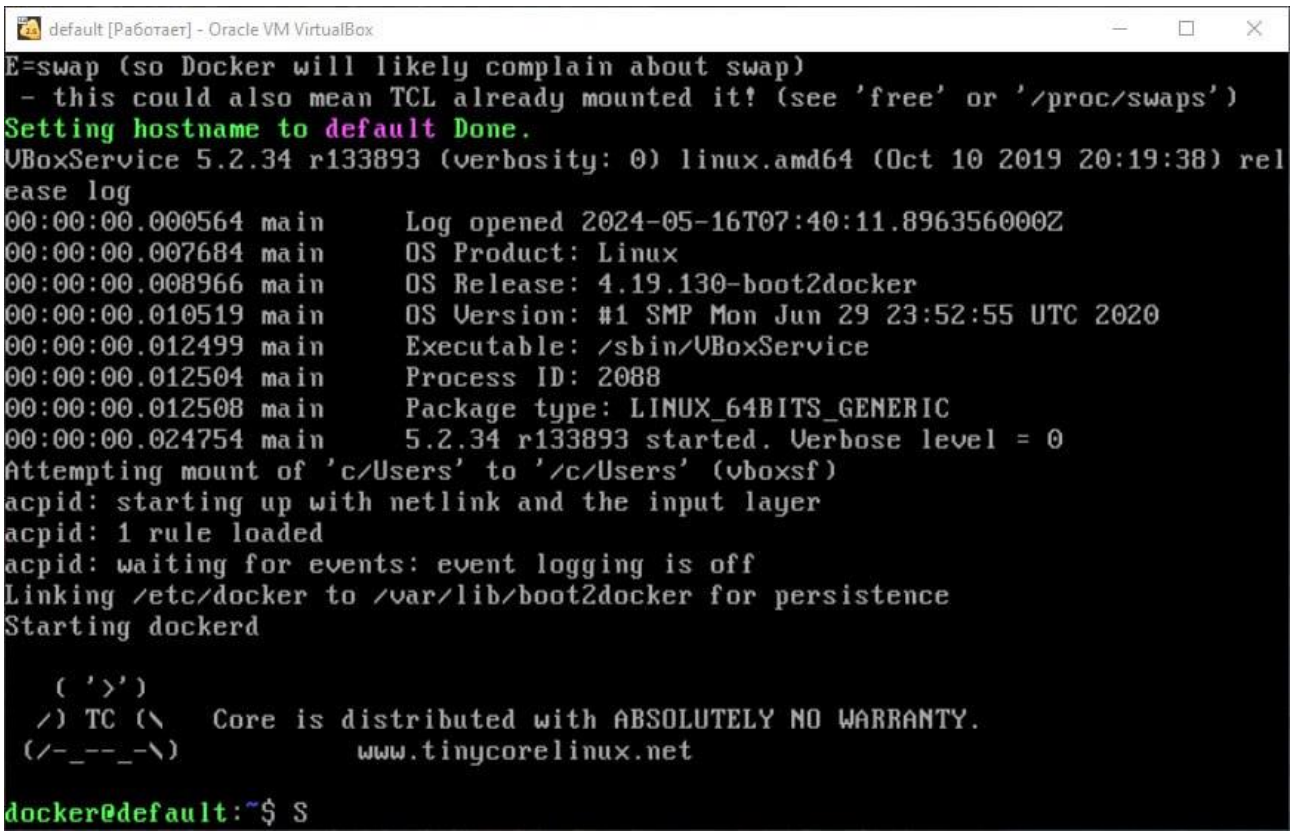

Рисунок 4.2 – CLI з успішно встановленим Docker

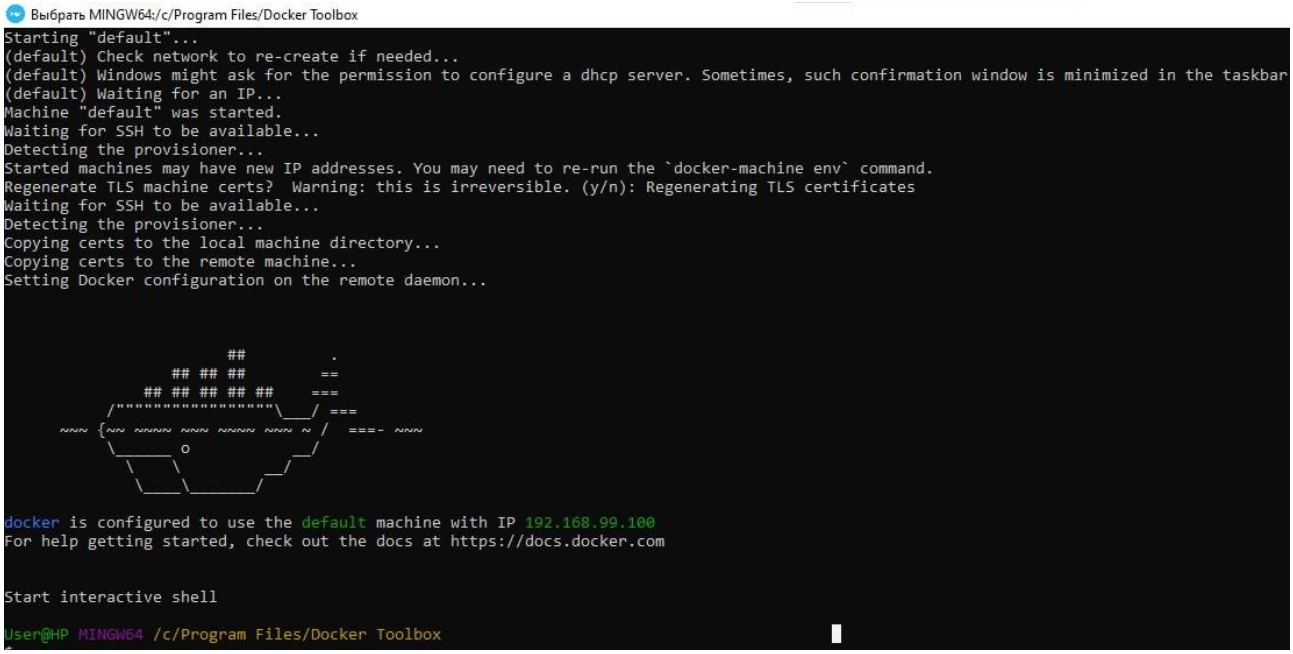

Рисунок 4.3 – Успішно розгорнутий Docker Toolbox в якому запущено Docker daemon

Наступний крок — інсталяція Docker Compose. На рисунку 4.4 зображені команди для інсталяції Docker Compose.

docker_compose_install.sh

```
# Завантаження останньої версії Docker Compose
sudo curl -L "https://github.com/docker/compose/releases/download/1.29.2/docker-compose-$(uname -s)-$(uname -m)"
        -o /usr/local/bin/docker-compose

# Надавання прав на виконання
sudo chmod +x /usr/local/bin/docker-compose

# Перевірка встановлення Docker Compose
docker-compose --version
```

Рисунок 4.4 – Команди для інсталяції Docker Compose.

Docker Swarm інтегрований у Docker і не потребує окремої інсталяції. На рисунку 4.5 зображені команди для активації Docker Swarm.

docker_swarm_install.sh

```
# Ініціалізація Docker Swarm
docker swarm init

# Перевірка статусу нод
docker node ls
```

Рисунок 4.5 – Команди для активації Docker Swarm.

Налаштування Dockerfiles відбувається окремо для кожного з мікросервісів. На рисунках 4.6 – 4.12 зображено Dockerfile для User Service, Product Service, Order Service, Payment Service, Inventory Service, Shipping Service, Promotion Service, Frontend відповідно.

Dockerfile

```
FROM openjdk:11

COPY ./target/USER_SERVICE-0.0.1-SNAPSHOT.jar ..//USER_SERVICE

CMD ["java","-jar","USER_SERVICE-0.0.1-SNAPSHOT.jar"]
```

Рисунок 4.6 – Dockerfile для User Service

Dockerfile

```
FROM openjdk:11

COPY ./target/PRODUCT_SERVICE-0.0.1-SNAPSHOT.jar ..//PRODUCT_SERVICE

CMD ["java","-jar","PRODUCT_SERVICE-0.0.1-SNAPSHOT.jar"]
```

Рисунок 4.7 – Dockerfile для Product Service

Dockerfile

```
FROM openjdk:11

COPY ./target/ORDER_SERVICE-0.0.1-SNAPSHOT.jar ..//ORDER_SERVICE

CMD ["java","-jar","ORDER_SERVICE-0.0.1-SNAPSHOT.jar"]
```

Рисунок 4.8 – Dockerfile для Order Service

Dockerfile

```
FROM openjdk:11

COPY ./target/PAYMENT_SERVICE-0.0.1-SNAPSHOT.jar ..//PAYMENT_SERVICE

CMD ["java","-jar","PAYMENT_SERVICE-0.0.1-SNAPSHOT.jar"]
```

Рисунок 4.9 – Dockerfile для Payment Service

Dockerfile

```
FROM openjdk:11

COPY ./target/INVENTORY_SERVICE-0.0.1-SNAPSHOT.jar ..//INVENTORY_SERVICE

CMD ["java","-jar","INVENTORY_SERVICE-0.0.1-SNAPSHOT.jar"]
```

Рисунок 4.10 – Dockerfile для Inventory Service

Dockerfile

```
FROM openjdk:11

COPY ./target/PROMOTION_SERVICE-0.0.1-SNAPSHOT.jar ..//PROMOTION_SERVICE

CMD ["java","-jar","PROMOTION_SERVICE-0.0.1-SNAPSHOT.jar"]
```

Рисунок 4.11 – Dockerfile для Promotion Service

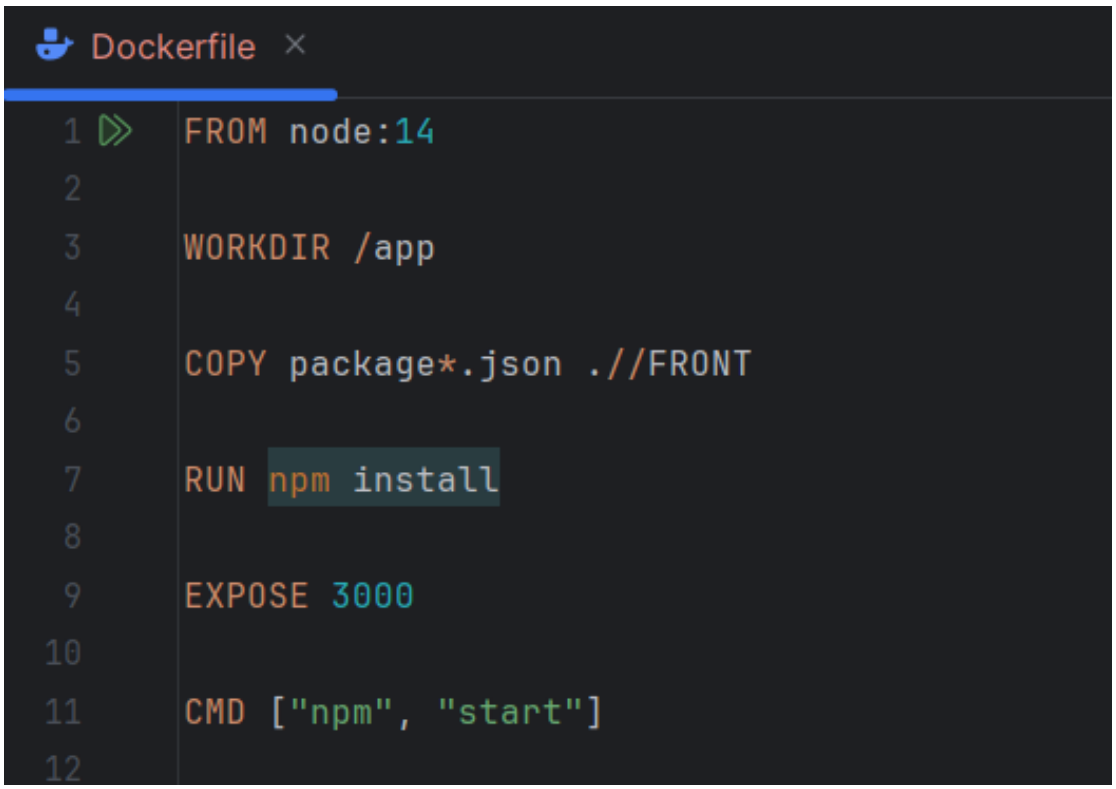

Рисунок 4.12 – Dockerfile для Frontend

На рисунку 4.13 представлеено файл docker-compose.yml для опису конфігурації сервісів проєкту.

Цей файл docker-compose.yml визначає конфігурацію всіх сервісів та баз даних, необхідних для нашого мікросервісного проєкту. Кожен сервіс запускається в окремому контейнері, що дозволяє нам масштабувати та керувати ними незалежно один від одного. Він є конфігурацією для розгортання багатокомпонентної мікросервісної програми з використанням Docker Compose. У цьому файлі визначено декілька сервісів, кожен з яких відповідає за різні аспекти системи, включаючи бази даних, мікросервіси та інструменти моніторингу.

docker-compose.yml

```yaml
version: '3.7'

services:
  user_service_db:
    image: postgres:latest
    container_name: 'user_service_db'
    environment:
      - POSTGRES_PASSWORD=${USER_DB_PW}
      - POSTGRES_DB=${USER_DB}
      - POSTGRES_USER=${USER_USER}
    ports:
      - "5432:5432"
    volumes:
      - ./postgres-user:/var/lib/postgresql/user
  user_service:
    container_name: 'user_service'
    build: ../USER_SERVICE
    ports:
      - "8080:8080"

  product_service_db:
    image: postgres:latest
    container_name: 'product_service_db'
    environment:
      - POSTGRES_PASSWORD=${PRODUCT_DB_PW}
      - POSTGRES_DB=${PRODUCT_DB}
      - POSTGRES_USER=${PRODUCT_USER}
    ports:
      - "5432:5432"
    volumes:
      - ./postgres-product:/var/lib/postgresql/product
  product_service:
    container_name: 'product_service'
    build: ../PRODUCT_SERVICE
    ports:
      - "8081:8081"

```

Рисунок 4.13 – Файл docker-compose.yml

docker-compose.yml

```
3   services:
38    order_service_db:
39      image: postgres:latest
40      container_name: 'order_service_db'
41      environment:
42        - POSTGRES_PASSWORD=${ORDER_DB_PW}
43        - POSTGRES_DB=${ORDER_DB}
44        - POSTGRES_USER=${ORDER_USER}
45      ports:
46        - "5432:5432"
47      volumes:
48        - ./postgres-order:/var/lib/postgresql/order
49    order_service:
50      container_name: 'order_service'
51      build: ../ORDER_SERVICE
52      ports:
53        - "8082:8082"
54
55    payment_service_db:
56      image: postgres:latest
57      container_name: 'payment_service_db'
58      environment:
59        - POSTGRES_PASSWORD=${PAYMENT_DB_PW}
60        - POSTGRES_DB=${PAYMENT_DB}
61        - POSTGRES_USER=${PAYMENT_USER}
62      ports:
63        - "5432:5432"
64      volumes:
65        - ./postgres-payment:/var/lib/postgresql/payment
66    payment_service:
67      container_name: 'payment_service'
68      build: ../PAYMENT_SERVICE
69      ports:
70        - "8083:8083"
```

Рисунок 4.13, аркуш 2

docker-compose.yml

```
services:
  inventory_service_db:
    image: postgres:latest
    container_name: 'inventory_service_db'
    environment:
      - POSTGRES_PASSWORD=${INVENTORY_DB_PW}
      - POSTGRES_DB=${INVENTORY_DB}
      - POSTGRES_USER=${INVENTORY_USER}
    ports:
      - "5432:5432"
    volumes:
      - ./postgres-inventory:/var/lib/postgresql/inventory
  inventory_service:
    container_name: 'inventory_service'
    build: ../INVENTORY_SERVICE
    ports:
      - "8084:8084"

  shipping_service_db:
    image: postgres:latest
    container_name: 'shipping_service_db'
    environment:
      - POSTGRES_PASSWORD=${SHIPPING_DB_PW}
      - POSTGRES_DB=${SHIPPING_DB}
      - POSTGRES_USER=${SHIPPING_USER}
    ports:
      - "5432:5432"
    volumes:
      - ./postgres-shipping:/var/lib/postgresql/shipping
  shipping_service:
    container_name: 'shipping_service'
    build: ../SHIPPING_SERVICE
    ports:
      - "8085:8085"
```

Рисунок 4.13, аркуш 3

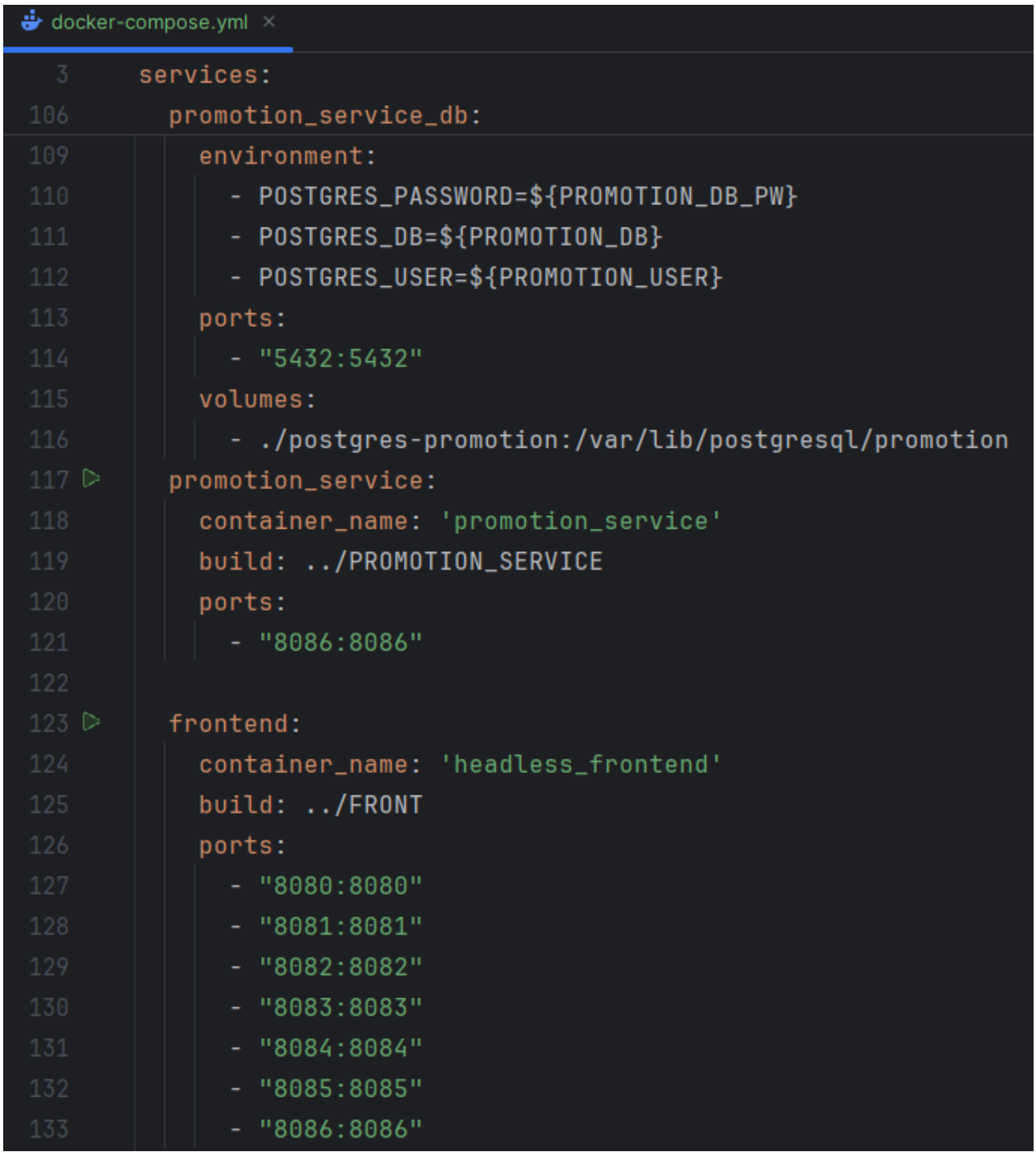

Рисунок 4.13, аркуш 4

Рисунок 4.13, аркуш 5

Кожен мікросервіс має свою власну базу даних PostgreSQL для ізоляції даних та полегшення управління. Наприклад, для сервісу користувача (user_service) визначена база даних user_service_db, що використовує образ postgres:latest. Налаштування цієї бази даних включають змінні оточення для встановлення пароля POSTGRES_PASSWORD, імені бази даних POSTGRES_DB та користувача POSTGRES_USER. База даних монтує локальний том «./postgres-user» для збереження даних та слухає на порту 5432` Сам сервіс користувача збирається з вихідного коду, розташованого в директорії «../USER_SERVICE», і стає доступним на порту 8080.

Подібні установки застосовані і до інших сервісів. Продуктовий сервіс (product_service) використовує базу даних product_service_db, яка аналогічно налаштовується через змінні оточення, монтує локальний том «./postgres-product» та слухає на порту 5432. Мікросервіс `product_service` збирається з директорії «../PRODUCT_SERVICE» та доступний на порту 8081.

Сервіс замовлень (order_service) має базу даних order_service_db з аналогічними налаштуваннями, монтує том «./postgres-order» та слухає на порту 5432. Сам сервіс order_service збирається з директорії «../ORDER_SERVICE» та доступний на порту 8082.

Платіжний сервіс (payment_service) використовує базу даних payment_service_db, яка монтує локальний том «./postgres-payment» та слухає на порту 5432. Мікросервіс payment_service збирається з директорії «../PAYMENT_SERVICE» та доступний на порту 8083.

Інвентаризаційний сервіс (inventory_service) використовує базу даних inventory_service_db, яка монтує локальний том «./postgres-inventory» та слухає на порту 5432. Сервіс inventory_service збирається з директорії «../INVENTORY_SERVICE» та доступний на порту «8084».

Сервіс доставки (shipping_service) використовує базу даних shipping_service_db, яка монтує локальний том «./postgres-shipping» та слухає на порту 5432. Мікросервіс shipping_service збирається з директорії «../SHIPPING_SERVICE» та доступний на порту 8085.

Сервіс акцій (promotion_service) має базу даних promotion_service_db, яка монтує локальний том «./postgres-promotion» і слухає на порту 5432. Сервіс promotion_service збирається з директорії «../PROMOTION_SERVICE та доступний на порту 8086.

Фронтенд сервіс (frontend) збирається з директорії «../FRONT» і маппирует всі порти мікросервісів (8080-8086), що дозволяє йому взаємодіяти з іншими сервісами.

Для моніторингу системи у конфігурацію включені Prometheus та Grafana. Prometheus використовується для збору та зберігання метрик, він слухає на порту 9090 та використовує файл конфігурації «./config/prometheus.yml». Grafana надає інтерфейс для візуалізації метрик, слухає на порту 3000, та його налаштування визначаються через змінні оточення GF_SECURITY_ADMIN_USER та GF_SECURITY_ADMIN_PASSWORD. Дашборди Grafana монтуються з локального каталогу

«./config/grafana/provisioning/dashboards`» а дані Grafana зберігаються у локальному томі «./grafana-data».

Після налаштування сервісів та баз даних, необхідно забезпечити відповідну мережеву інфраструктуру для забезпечення стабільності та доступності системи. На рисунку 4.14 зображені команди для налаштування брандмауера.

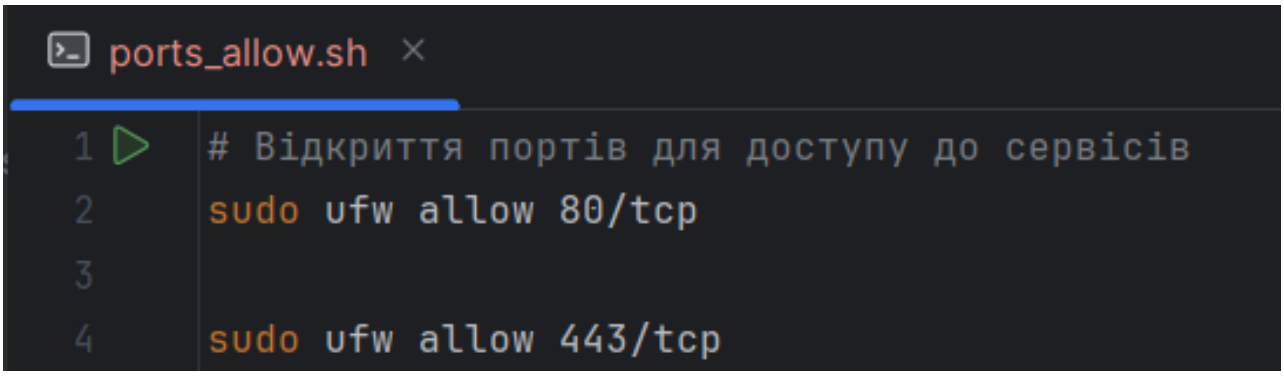

Рисунок 4.14 – Команди для налаштування брандмауера

Ці команди відкривають необхідні порти для HTTP (порт 80) та HTTPS (порт 443), щоб забезпечити доступ до веб-інтерфейсу та інших сервісів у Docker контейнерах.

Для забезпечення рівномірного розподілу навантаження між контейнерами використовується Docker Swarm. Його можна налаштувати за допомогою наступних команд. На рисунку 4.15 зображені команди для створення кластера Docker Swarm. Ці команди створюють Docker Swarm кластер та дозволяють розгортати сервіси у цьому кластері, забезпечуючи автоматичне балансування навантаження.

docker_swarm_config.sh

```
# Ініціалізація Docker Swarm на вузлі
docker swarm init

# Додавання інших вузлів до кластера (за потреби)
docker swarm join --token <token> <IP-адреса>

# Створення сервісу у Swarm
docker service create --name my-service <image>
```

Рисунок 4.15 – Команди для налаштування кластера Docker Swarm

Після налаштування всіх компонентів системи важливо провести тестування, щоб переконатися у правильному функціонуванні всіх сервісів та їх взаємодії. Це допоможе виявити та виправити можливі проблеми до впровадження системи в реальному середовищі.

4.3 Порівняльний аналіз ресурсовитрат

Для оцінки ефективності розгортання мікросервісного eCommerce проєкту з використанням Docker у порівнянні з традиційними методами розгортання, розглянемо ключові аспекти, такі як час розгортання, кількість використовуваних ресурсів та необхідні людські ресурси. Це дозволить об'єктивно оцінити переваги і недоліки кожного підходу. Для простоти порівняння будемо використовувати атомарну кількіть ресурсів. Для кожного з підходів.

Традиційне розгортання передбачає використання фізичних серверів або віртуальних машин для кожного компонента системи. Процес включає налаштування операційної системи, встановлення необхідного програмного

забезпечення, конфігурацію мережі та балансування навантаження. На рисунку 4.16 зображено діаграму Ганта для традиційного розгортання ПЗ.

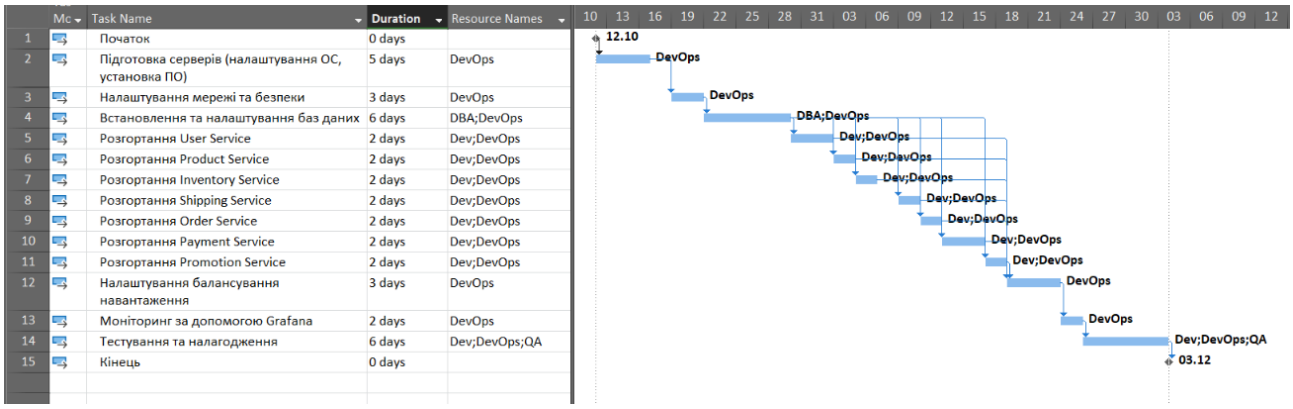

Рисунок 4.16 – Діаграма Ганта для традиційного розгортання ПЗ

Загальний час розгортання традиційним способом складає 71 людино-день. Майже кожен етап вимагає координації між робітниками, що значно збільшує час, зусилля та вартість проєкту.

Розгортання за допомогою Docker передбачає використання контейнерів для кожного мікросервісу та бази даних. Використання Docker Compose та Docker Swarm дозволяє автоматизувати та спростити процес розгортання і масштабування. На рисунку 4.17 зображено діаграму Ганта для розгортання ПЗ за допомогою Docker.

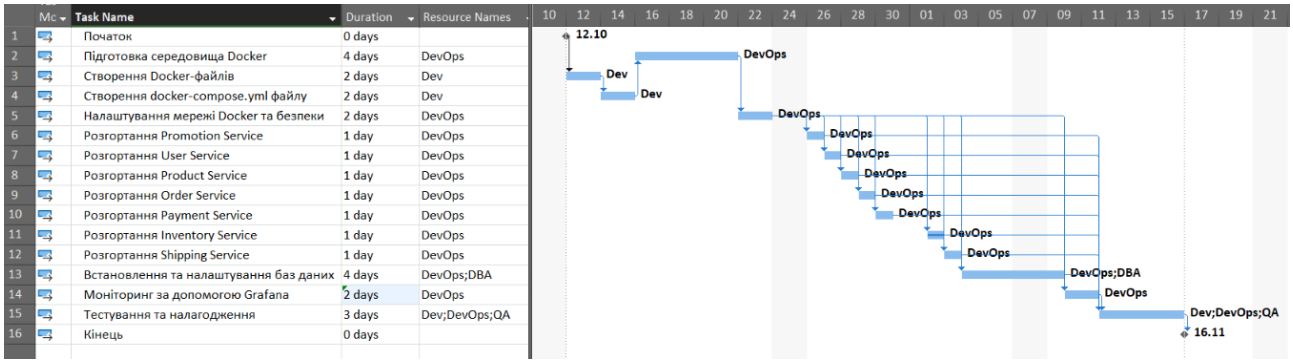

Рисунок 4.17 – Діаграма Ганта для розгортання ПЗ за допомогою Docker

Загальний час розгортання за допомогою Docker складає 36 людино-день. Це на 50% швидше у порівнянні з традиційним підходом. Використання Docker значно зменшує час налаштування середовища, оскільки контейнери можна швидко та легко розгортати, копіювати та масштабувати.

В таблиці 4.3 представлено порівняння ресурсовитрат при розгортанні ПЗ.

Таблиця 4.3 – Порівняння ресурсовитрат при розгортанні ПЗ.

	Традиційне розгортання	Розгортання за допомогою Docker
Час розгортання	71 людино-день	36 людино-день
Кількість ресурсів	Потребує більше фізичних серверів або віртуальних машин, оскільки кожен компонент системи зазвичай розміщується на окремій машині для забезпечення надійності та продуктивності.	Ефективніше використовує ресурси за рахунок контейнеризації, дозволяючи розміщувати декілька контейнерів на одному фізичному сервері або віртуальній машині, зменшуючи витрати на апаратне забезпечення.
Людські ресурси	Потребує більше часу і зусиль з боку DevOps спеціалістів, розробників та DBA для налаштування, тестування та підтримки системи	Спрощує процес налаштування та розгортання, автоматизуючи багато задач, що зменшує необхідність у великій кількості інженерів

Розгортання мікросервісного eCommerce проєкту за допомогою Docker значно ефективніше з точки зору часу та ресурсів у порівнянні з традиційними методами. Використання Docker дозволяє скоротити час розгортання на 50%, зменшити витрати на апаратне забезпечення та зменшити необхідність у людських ресурсах, що робить цей підхід більш привабливим для сучасних IT-проєктів.

Docker забезпечує гнучкість і масштабованість, що є критичним для сучасних eCommerce проєктів, які повинні адаптуватися до змін в попиті і умовах ринку. Контейнеризація полегшує управління залежностями та забезпечує ізоляцію середовищ, що зменшує ризики виникнення конфліктів між різними компонентами системи.

Крім того, розгортання за допомогою Docker дозволяє легко інтегрувати інструменти моніторингу, такі як Grafana, які забезпечують прозорість і контроль за станом системи в реальному часі. Це допомагає швидко виявляти та вирішувати проблеми, підвищуючи надійність і стабільність роботи всієї системи.

Таким чином, розгортання за допомогою Docker не тільки спрощує процес, але й забезпечує більш надійне, масштабоване і ефективне використання ресурсів, що є вирішальним фактором для успішного впровадження і підтримки мікросервісних архітектур в IT-проєктах.

4.4 Ризики методу розгортання інфраструктури мікросервісного eCommerce проєкту за допомогою Docker відносно традиційного розгортанн

Розгортання інфраструктури для eCommerce проєктів за допомогою методів традиційного та контейнерного розгортання мають суттєві відмінності. Використання контейнеризації, зокрема Docker, пропонує значні переваги, але також супроводжується певними ризиками. Цей розділ аналізує ризики

використання Docker для розгортання інфраструктури мікросервісного eCommerce проєкту порівняно з традиційним підходом.

Одним з ключових параметрів для оцінки ефективності обох підходів є час розгортання.

$$T_{\text{трад.}} = 71 \text{ днів}$$

$$T_{Docker} = 36 \text{ днів}$$

Скорочення часу розгортання при використанні Docker вказує на значне прискорення процесів і підвищення ефективності. Проте, прискорення може супроводжуватись різними ризиками, які потребують детального аналізу та управління. В таблиці 4.4 представлено порівняння ризиків при розгортанні ПЗ.

Таблиця 4.4 – Порівняння ризикв при розгортанні ПЗ.

Параметр	Традиційне розгортання	Розгортання за допомогою Docker
Час розгортання	71 день	36 дкень
Гнучкість	Низька	Висока
Складність	Середня	Висока
Можливість ізоляції	Низька	Висока
Залежність від платформи	Висока	Низька
Управління ресурсами	Меньш ефективна	Високо ефективне
Витрати на інфраструктуру	Високі	Низькі

Розрахуємо індекс ризику розгортання інфраструктури. Формула індексу ризику розгортання інфраструктури розраховується наступним чином:

$$R = \frac{T \times D}{G}, \quad (4.1)$$

де R – індекс ризику для традиційного розгортання,

T – час розгортання (дні),

D – показник залежність від платформи,

G –показник гнучкості.

Показники залежності від платформи та гнучкості отримано шдяхом аналізу проєктів, інструментам аналізу, консультації з іншими високорівневими спеціалістами та власним досвідом і експертній оцінці.

Розрахуємо індекс ризику для традиційного підходу:

$$R_{\text{трад}} = \frac{T_{\text{трад}} \times D_{\text{трад}}}{G_{\text{трад}}} = \frac{71 \times 1.5}{O.7} = 152.14 \quad (4.2)$$

Розрахуємо індекс ризику для Docker:

$$R_{Docker} = \frac{T_{Docker} \times D_{Docker}}{G_{Docker}} = \frac{36 \times 2}{1.8} = 152.14 \quad (4.3)$$

Метод розгортання за допомогою Docker значно знижує час розгортання та підвищує гнучкість, але супроводжується збільшенням складності управління. Традиційне розгортання має свої переваги у простоті та зрозумілості процесів, але пов’язане з більшими витратами та меншою гнучкістю. Для успішного впровадження методу Docker необхідно ретельно управляти ризиками, використовуючи відповідні інструменти оркестрації та моніторингу контейнерів. На основі розрахованих індексів ризику, Docker демонструє менший ризик у порівнянні з традиційним підходом, що вказує на його перевагу за умови належного управління процесами.

ВИСНОВКИ

В ході роботи було проведено дослідження методів контейнеризації, що включало детальний аналіз Docker та його застосування для розгортання оточення програмного забезпечення. Під час аналізу було виявлено ключові переваги контейнеризації, такі як швидкість створення, переносимість між середовищами, а також можливість ізолювати та стандартизувати різноманітні середовища.

Особлива увага була приділена дослідженню мікросервісної архітектури в eCommerce проєктах та ролі Docker у розгортанні мікросервісів. Виявлено, що Docker забезпечує ефективне управління мікросервісами, їх ізоляцію, незалежність та дозволяє забезпечити швидкий, безперебійний розвиток впровадження нового функціоналу.

Експериментальна перевірка підтвердила, що впровадження Docker у eCommerce проєктах з мікросервісною архітектурою суттєво підвищує ефективність розгортання та управління програмним забезпеченням. Застосування Docker дозволяє спростити процес масштабування, забезпечити ефективне управління залежностями та конфігураціями, що призвело до зменшення часу розгортання, витрат ресурсів та кількості помилок у програмному забезпеченні.

Результати кваліфікаційної роботи представлені у матеріалах 28 міжнародного молодіжний форуму «Радіоелектроніка і молодь у XXI столітті», 2024 р.

ПЕРЕЛІК ДЖЕРЕЛ ПОСИЛАННЯ

1. 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 2022-12-19 | Journal article, DOI: 10.15407/vidbir2022.50.078.

2. Методичні вказівки щодо розробки та оформлення кваліфікаційної роботи (для студентів усіх форм навчання другого (магістерського) рівня вищої освіти спеціальності 122 Комп'ютерні науки освітньопрофесійної програми «Управління проєктами в галузі інформаційних технологій») / Упоряд.: Петров К.Е., Левикін В.М., Чалий С.Ф., Євланов М.В., Саєнко В.І., Міхнов Д.К., Міхнова А.В., Чала О.В. – Харків: ХНУРЕ, 2021. – 30 с.

3. ДСТУ 3008:2015. Державний стандарт України. Інформація та документація. Звіти у сфері науки і техніки. Структура та правила оформлення. – К.: ДП «УкрНДНЦ», 2016. – 31 с.

4. ДСТУ 8302:2015. Бібліографічне посилання. Загальні положення та правила складання. / Видання офіційне. – К.: ДП «УкрНДНЦ», 2016 – 20 с.

5. Best Practices for Container: вебсайт URL: https://cloud.google.com/ blog/products/containers-kubernetes (дата звернення 15.01.2024 по 25.05.2024)

6. Docker Documentation: вебсайт URL: https://docker-docs.uclv.cu/ (дата звернення 17.01.2024 по 25.05.2024)

7. Podman Documentation: вебсайт URL: https://docs.podman.io/en/v3.0/ Reference.html (дата звернення 18.01.2024 по 25.05.2024)

8. Kubernetes Documentation: вебсайт URL: https://kubernetes.io/docs/home (дата звернення 18.01.2024 по 25.05.2024)

9. Прес Р. Д. АНАЛІЗ МЕТОДІВ КОНТЕЙНЕРИЗАЦІЇ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ : матеріали XXVIII Міжнар. молодіж. форуму, м. Харків, 17 трав. 2024 р. Харків, 2024. С. 248-249.

10. Коваленко О.С., Добровська Л. М., Проєктування інформаційних

систем: Загальні питання теорії проєктування ІС, Електронне мережне навчальне видання, 2020. – 192с.

11. С. О. Цибульник, К. С. Барандич, Технології розроблення програмного забезпечення Частина 1, 2022. – 270с.

12. В.В.Завгородній, К.М.Ялова., Конспект лекцій з дисципліни «Архітектура та проєктування програмного забезпечення» , 2019.– 144с.

13. Docker overview documentation: вебсфйт URL: https://docs.docker.com/get-started/overview/ (дата звернення 25.04.2024 по 25.05.2024)

14. The Classic Enterprise E-Commerce Platforms: вебсайт URL: https://fabric.inc/blog/commerce/enterprise-ecommerce-platforms (дата звернення 15.05.2024 по 25.05.2024)

15. Docker integration: вебсайт URL: https://docs.datadoghq.com/containers/docker/data_collected/ (дата звернення 12.05.2024 по 25.05.2024)

16. James Turnbull, The Docker Book: Containerization Is the New Virtualization James Turnbull, 2014 – 344с.

17. Swarm mode overview: вебсайт URL: https://docs.docker.com/engine/swarm/ (дата звернення 13.05.2024 по 25.05.2024)

18. How microservices benefit the organization: вебсайт URL: https://www.ibm.com/topics/microservices (дата звернення 14.05.2024 по 25.05.2024)

19. Using RBAC Authorization: вебсайт URL: https://kubernetes.io/docs/reference/access-authn-authz/rbac/ дата звернення 15.05.2024 по 25.05.2024)