Міністерство освіти і науки України
Харківський національний університет радіоелектроніки

Факультет комп'ютерних наук
(повна назва)

Кафедра програмної інженерії
(повна назва)

КВАЛІФІКАЦІЙНА РОБОТА
Пояснювальна записка

рівень вищої освіти другий (магістерський)

Дослідження методів оптимізації продуктивності хмарних веб-додатків контейнеризованих у Kubernetes
(тема)

Виконав:
здобувач 2 року навчання
групи ІПЗм-23-1

Дмитро ВОРОНА
(Власне ім'я, ПРІЗВИЩЕ)

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

Керівник проф. Галуза О.А.
(посада, Власне ім'я, ПРІЗВИЩЕ)

Допускається до захисту
Зав. кафедри ____________ Кирило СМЕЛЯКОВ
(підпис) (Власне ім'я, ПРІЗВИЩЕ)

2025 р.

Харківський національний університет радіоелектроніки

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

ЗАТВЕРДЖУЮ:
Зав. кафедри ________
(підпис)
«___» _________ 2025 р.

ЗАВДАННЯ
НА КВАЛІФІКАЦІЙНУ РОБОТУ

студентові Вороні Дмитру Олександровичу
(прізвище, ім'я, по батькові)

1. Тема роботи «Дослідження методів оптимізації продуктивності хмарних веб-додатків контейнеризованих у Kubernetes»

Затверджена наказом по університету від 15.04. 2024р. № 290 Ст

2. Термін подання студентом роботи до екзаменаційної комісії 23.06.2025

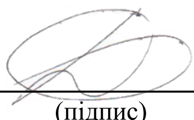
3. Вихідні дані до роботи опис досліджуваних методів оптимізації продуктивності хмарних веб-додатків контейнеризованих у Kubernetes, мови програмування Golang, середовища розробки IntelliJ IDEA 2025

4. Перелік питань, що потрібно опрацювати в роботі
мета роботи, аналіз предметної галузі, постановка задачі, створення програмної системи та підготовка до експерименту, проведення експериментів та аналіз отриманих результатів

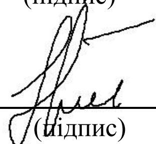
КАЛЕНДАРНИЙ ПЛАН

№	Назва етапів роботи	Термін виконання етапів роботи	Примітка
1	Отримання завдання	16.04.2025	*виконано*
2	Аналіз предметної галузі	21.04.2025-05.05.2025	*виконано*
3	Постановка задачі	05.05.2025-12.05.2025	*виконано*
4	Створення програмної системи та підготовка до експерименту	12.05.2025-25.05.2025	*виконано*
5	Підготовка до апробації результатів дослідження. Публікація матеріалів	25.05.2025-05.06.2025	*виконано*
6	Проведення експериментів та аналіз отриманих результатів	05.06.2026-16.06.2025	*виконано*
7	Підготовка пояснювальної записки	16.06.2025-17.06.2025	*виконано*
8	Підготовка презентації та доповіді	17.06.2025-18.06.2025	*виконано*
9	Перевірка на плагіат	16.06.2025	*виконано*
10	Нормоконтроль	18.06.2025	*виконано*
11	Рецензування	18.06.2025	*виконано*
12	Попередній захист	20.06.2025	*виконано*
13	Занесення диплома в електронний архів	23.06.2025	*виконано*
14	Допуск до захисту у зав. кафедри	23.06.2025	*виконано*

Дата видачі завдання 16.04.2025р.

Студент (ка)  _____ Дмитро ВОРОНА

(підпис)

Керівник роботи  _____ проф. Олексій ГАЛУЗА

(підпис) (посада, Власне ім’я, ПРІЗВИЩЕ)

РЕФЕРАТ / ABSTRACT

Пояснювальна записка містить: 67 с., 20 рис., 4 табл., 10 джерел, 4 додатка.

ДОСЛІДЖЕННЯ, КОНТЕЙНЕР, ОПТИМІЗАЦІЯ, ХМАРА, AZURE, CERT MANAGER, CONTINUOUS DEVELOPMENT, CONTINUOUS INTEGRATION, DOCKER, GRAFANA, KUBERNETES, PROMETHEUS.

Об'єктом дослідження є хмарні веб-додатки контенеризовані у Kubernetes.

Метою роботи є аналіз проблемної галузі та проведення дослідження методів оптимізації продуктивності хмарних веб-додатків контейнеризованих у Kubernetes

Під час дослідження були реалізовані програмні системи за допомогою мови Golang [1-2], Docker контейнерізації та Kubernetes для оркестрації. Для розробки використовувалися наступні технології: Azure Kubernetes Service, Azure Container Registry та Azure Virtual Network.

Також було використано Prometheus та Grafana для моніторінгу та збору метрик. Був використано Cert manager для інтеграції сервісу сертифікації на Kubernetes кластері.

AZURE, CERT MANAGER, CLOUD, CONTAINER, CONTINUOUS DEVELOPMENT, CONTINUOUS INTEGRATION, DOCKER, GRAFANA, KUBERNETES, OPTIMIZATION, PROMETHEUS, RESEARCH.

The object of the research is cloud-based web applications containerized in Kubernetes.

The purpose of this work is to analyze the problem domain and conduct a study of methods for optimizing the performance of cloud-based web applications containerized in Kubernetes.

During the research, software systems were implemented using the Golang programming language [1-2], Docker for containerization, and Kubernetes for

orchestration. The development utilized the following technologies: Azure Kubernetes Service, Azure Container Registry, and Azure Virtual Network.

Additionally, Prometheus and Grafana were used for monitoring and metrics collection. Cert Manager was employed to integrate the certification service into the Kubernetes cluster.

Завідувачу кафедри
ПІ__________________
(скорочена назва кафедри)
проф. Кирилу СМЕЛЯКОВУ_
(вчене звання, сласне ім'я, прізвище)

ЗАЯВА
щодо самостійності виконання кваліфікаційної роботи та можливості її публікації (та/або публікації анотації кваліфікаційної роботи) в електронному архіві відкритого доступу ElAr KhNURE

Я, _______Ворона Дмитро Олександрович___________________________
(прізвище, ім'я, по батькові)

здобувач вищої освіти на другому (магістерському) рівні вищої освіти академічної групи_______ІПЗм-23-1____

кафедра__________________ програмної інженерії _________________________,
(повна назва кафедри)

заявляю: моя кваліфікаційна робота на тему _____Дослідження методів оптимізації продуктивності хмарних веб-додатків контейнеризованих у Kubernetes__,
(назва роботи)

що буде представлена в екзаменаційну комісію для публічного захисту, виконана самостійно, в ній не містяться елементи плагіату і вона може бути опублікована в репозиторії "ElArKhNURE". Погоджуюся з авторським договором, відповідно до Положення про репозиторій ХНУРЕ "ElArKhNURE". Всі запозичення з друкованих та електронних джерел мають відповідні посилання.

Я ознайомлений (а) з вимогами академічної доброчесності, згідно з якими виявлення плагіату є підставою для відмови в допуску кваліфікаційної роботи до захисту та застосування дисциплінарних заходів.

Дата
17.06.2025

Підпис

ЗМІСТ

Вступ 8
1 Аналіз предметної галузі 9
1.1. Огляд предметної галузі 9
1.1.2 Специфіка застосування контейнеризації в сучасних веб-додатках 9
1.2. Існуючі підходи та їх обмеження 10
1.2.1 Аналіз існуючих методів оптимізації продуктивності у хмарних середовищах 10
1.2.2 Переваги та недоліки використання Kubernetes для контейнеризації 11
1.3 Огляд й аналіз літературних, наукових джерел 13
1.3.1 Огляд основних джерел 13
1.3.2 Аналіз літератури 14
1.3.3 Оцінка актуальності та новизни 16
1.3.4 Висновки з огляду 17
2 Постановка задачі 18
3 Створення програмної системи та підготовка до експерименту 21
3.1 Архітектура та проектування системи 21
3.2 Оптимізація системи 29
4 Проведення експериментів та аналіз отриманих результатів 33
4.1 Підготовка до проведення експериментів 33
4.2 Метод оптимізації за допомогою горизонтального автоскейлінгу 36
4.3 Метод оптимізації за допомогою вертикального автоскейлінгу 40
4.4 Метод оптимізації за допомогою мікросервісів 43
Висновки 49
Перелік джерел посилання 51
Додаток А Слайди презентації 54
Додаток Б Апробація результатів роботи 62
Додаток В Звіт з результатами перевірки на унікальність тексту в базі ХНУРЕ 65
Додаток Г Експертний висновок результатів перевірки кваліфікаційної роботи 67

ВСТУП

Сучасна індустрія розробки програмного забезпечення активно впроваджує хмарні технології для забезпечення високої доступності, масштабованості та продуктивності веб-додатків. Одним з ключових інструментів для ефективного управління хмарними додатками є Kubernetes — платформа для оркестрації контейнерів, що забезпечує автоматизацію розгортання, масштабування та управління контейнеризованими додатками.

Із зростанням обсягів даних і складності додатків зростає необхідність у пошуку ефективних методів оптимізації продуктивності хмарних веб-додатків. Це включає використання ресурсів обчислювальної інфраструктури, мінімізацію затримок у обробці запитів та забезпечення стабільності роботи додатків при пікових навантаженнях.

Метою даної кваліфікаційної роботи є дослідження методів оптимізації продуктивності веб-додатків, розгорнутих у контейнерах під управлінням Kubernetes. У роботі буде розглянуто існуючі підходи до налаштування ресурсів, оптимізації мережевих запитів, використання горизонтального та вертикального масштабування, а також методи моніторингу і профілювання додатків. Основні результати теоретичної і практичної складової проекту були надіслані до «XXIX міжнародний молодіжний форум «Радіоелектроніка та молодь у XXI столітті» [3]

1 АНАЛІЗ ПРЕДМЕТНОЇ ГАЛУЗІ

1.1. Огляд предметної галузі

1.1.1 Загальні тенденції розвитку хмарних технологій

Хмарні технології відіграють ключову роль у сучасній IT-індустрії, забезпечуючи ефективне управління ресурсами, високий рівень доступності та масштабованості. Сьогодні організації все частіше використовують мультихмарні середовища, залучаючи кілька хмарних провайдерів для підвищення гнучкості, безпеки та мінімізації ризиків залежності від одного постачальника. Паралельно з цим контейнеризація стає стандартом для розробки та розгортання хмарних додатків. Технології, такі як Docker, і платформи оркестрації на зразок Kubernetes, забезпечують високу ефективність і масштабованість таких систем.

Автоматизація процесів також є однією з ключових тенденцій, що включає впровадження DevOps-підходів та автоматизацію управління інфраструктурою, сприяючи швидкому випуску оновлень і зниженню витрат на обслуговування. Окрім цього, зростає фокус на безпеку: розробляються нові методи захисту хмарних середовищ, включаючи шифрування даних, контроль доступу та моніторинг активності. Ще однією важливою тенденцією є переходи до серверлес архітектури, які дозволяють розробникам зосередитися на логіці додатків, делегуючи управління інфраструктурою провайдерам.

Узгоджено з цими напрямками, хмарні технології продовжують інтегруватися у всі сфери діяльності, надаючи підприємствам і розробникам інструменти для підвищення продуктивності, оптимізації витрат та впровадження інноваційних рішень.

1.1.2 Специфіка застосування контейнеризації в сучасних веб-додатках

Контейнеризація стала ключовим інструментом у розробці та розгортанні сучасних веб-додатків. Її основна перевага полягає в ізоляції середовищ виконання, що дозволяє забезпечити однакові умови для роботи додатків у процесі розробки, тестування та продакшн-розгортання. Це суттєво знижує ймовірність виникнення проблем, пов'язаних із різницею у конфігураціях. За допомогою контейнерів можна

легко упаковувати додаток разом із його залежностями в єдину розгортальну одиницю, що спрощує управління і забезпечує високу портативність між різними середовищами.

У сучасних веб-додатках контейнери широко використовуються для мікросервісної архітектури, де кожна функція або сервіс виконується в окремому контейнері. Це забезпечує гнучкість у масштабуванні та розподілі ресурсів, дозволяючи збільшувати або зменшувати кількість екземплярів конкретних мікросервісів залежно від навантаження. Оркестраційні платформи, такі як Kubernetes, надають можливості для автоматичного масштабування, балансування навантаження та управління відмовами, що робить контейнеризацію особливо привабливою для створення високонавантажених веб-додатків.

Попри всі переваги, контейнеризація має і свої виклики. Оптимальне управління ресурсами, наприклад CPU та пам'яттю, є критичним завданням, оскільки недостатнє виділення ресурсів може призвести до зниження продуктивності, тоді як надлишкове використання збільшує витрати. Важливим аспектом є також безпека контейнеризованих додатків, включаючи управління вразливостями у базових образах та налаштування правил доступу.

Контейнеризація суттєво змінила підхід до розробки веб-додатків, зробивши їх більш гнучкими, масштабованими та легшими в обслуговуванні. Водночас успішне впровадження контейнеризації вимагає врахування низки технічних та організаційних аспектів, що становить основу для подальших досліджень і вдосконалень.

1.2. Існуючі підходи та їх обмеження

1.2.1 Аналіз існуючих методів оптимізації продуктивності у хмарних середовищах

Методи оптимізації продуктивності у хмарних середовищах охоплюють широкий спектр технологій і стратегій. Одним із найважливіших є оптимізація використання ресурсів, зокрема за допомогою правильного налаштування лімітів і запитів для контейнерів у Kubernetes. Це дозволяє забезпечити баланс між

продуктивністю додатків і витратами на інфраструктуру. Іншою поширеною практикою є використання горизонтального і вертикального автоскейлінгу, що дозволяє автоматично адаптувати кількість ресурсів відповідно до поточного навантаження. Горизонтальне масштабування є однією з основних переваг мікросервісної архітектури, яке дозволяє підвищити продуктивність системи за рахунок додавання нових екземплярів сервісів без зміни їхньої внутрішньої структури. Це особливо важливо для веб-додатків, які працюють в умовах змінного навантаження

Ще одним аспектом є оптимізація мережевих запитів, зокрема використання CDN для зниження затримок і балансувальників навантаження для розподілу трафіку між різними вузлами. Методи кешування, такі як Redis, також допомагають суттєво зменшити час відповіді на запити. Для підвищення стабільності та доступності систем активно впроваджуються стратегії ізоляції збоїв, такі як використання кількох зон доступності (Availability Zones) і автоматичне відновлення в разі збоїв.

Однак існуючі підходи мають свої обмеження. Наприклад, автоматичне масштабування не завжди може враховувати специфіку навантаження, що може призвести до непередбачуваних затримок. Деякі методи, такі як налаштування ресурсів, потребують глибокого аналізу і профілювання додатків, що може бути трудомістким і потребувати значних знань. Крім того, оптимізація зазвичай залежить від конкретної хмарної платформи, що обмежує її універсальність.

Оптимізація продуктивності у хмарних середовищах є багатогранною задачею, яка вимагає комплексного підходу, враховуючи як технічні аспекти, так і економічні обмеження. Подальше вдосконалення існуючих методів і розробка нових рішень залишаються актуальними викликами.

1.2.2 Переваги та недоліки використання Kubernetes для контейнеризації

Платформа Kubernetes пропонує численні переваги для контейнеризації, завдяки яким вона стала стандартом для управління контейнеризованими додатками. Однією з ключових переваг є автоматизація процесів розгортання,

масштабування та управління додатками. Kubernetes дозволяє динамічно адаптувати кількість екземплярів додатка до змін у навантаженні завдяки використанню механізмів горизонтального та вертикального масштабування. Це забезпечує ефективніше використання ресурсів та знижує витрати на інфраструктуру.

Крім того, Kubernetes надає інструменти для забезпечення надійності та доступності додатків. Механізми відновлення після збоїв (self-healing) автоматично перезапускають несправні контейнери або переміщують їх на інші вузли (nodes) кластера. Також система підтримує балансування навантаження між сервісами, що дозволяє рівномірно розподіляти трафік і уникати перевантаження окремих компонентів.

Ще однією перевагою є портативність і можливість використання мультихмарних середовищ. Kubernetes дозволяє розгортати додатки в різних хмарах або локальних середовищах, забезпечуючи однаковий рівень керованості незалежно від інфраструктури. Завдяки цьому організації можуть уникати залежності від одного провайдера та підвищувати стійкість своїх систем.

Однак використання Kubernetes має і певні недоліки. Платформа є досить складною для освоєння та вимагає значних зусиль для налаштування. Наприклад, налаштування ресурсних квот для контейнерів або конфігурація мережевих політик потребує глибокого розуміння її функціоналу. Іншим викликом є високі витрати на підтримку інфраструктури Kubernetes, особливо для малих організацій, які не мають достатньо ресурсів для її оптимізації та управління.

Крім того, безпека контейнеризованих додатків у Kubernetes вимагає особливої уваги. Необхідно налаштовувати політики доступу, регулярно перевіряти контейнери на вразливості та забезпечувати ізоляцію між різними компонентами.

Kubernetes пропонує значні переваги для контейнеризації додатків, але одночасно потребує відповідального підходу до її впровадження та управління. Платформа найбільше підходить для середовищ із високими вимогами до

масштабованості та надійності, однак її складність і вартість можуть стати бар’єром для широкого впровадження в менших організаціях.

1.3 Огляд й аналіз літературних, наукових джерел

1.3.1 Огляд основних джерел

Одним із основних аспектів дослідження є аналіз наукових джерел, які висвітлюють актуальні аспекти оптимізації продуктивності у хмарних середовищах та контейнеризації. Для огляду було обрано шість ключових джерел, які надають всебічний огляд існуючих підходів, методів та викликів, пов’язаних із використанням Kubernetes для управління контейнеризованими додатками.

У статті "Kubernetes and Docker Load Balancing: State-of-the-Art Techniques and Challenges"[4] представлено огляд сучасних методів балансування навантаження у контейнеризованих середовищах, що використовують Kubernetes та Docker. Особливу увагу приділено викликам, таким як управління динамічними змінами навантаження та забезпечення стабільності при масштабуванні.

Джерело "Enhanced Visibility for Real-time Monitoring and Alerting in Kubernetes by Integrating Prometheus, Grafana, Loki, and Alerta"[5] описує інтеграцію інструментів для моніторингу та візуалізації, що значно покращують ефективність управління продуктивністю. Автори акцентують увагу на важливості реального часу для виявлення проблем та запобігання збоїв.

У роботі "Kubernetes in Microservices"[6] висвітлюється використання Kubernetes у мікросервісній архітектурі. Автори досліджують специфіку масштабування мікросервісів, забезпечення їхньої взаємодії та мінімізацію витрат на обчислювальні ресурси.

Стаття "Optimizing Container Management with Kubernetes on Linux: Key Strategies and Common Obstacles"[7] зосереджується на ключових стратегіях оптимізації управління контейнерами в операційних системах Linux. У дослідженні також описані основні перешкоди, такі як недостатнє налаштування ресурсів або складності інтеграції з інфраструктурою.

Джерело "Navigating the Landscape of Kubernetes Security Threats and Challenges"[8] присвячене безпеці у Kubernetes. У ньому детально аналізуються загрози, пов'язані з некоректним налаштуванням доступу, вразливістю образів контейнерів та потенційними атаками на кластер.

У роботі "Containerization in Cloud Computing: Comparing Docker and Kubernetes for Scalable Web Applications"[9] автори проводять порівняння Docker та Kubernetes з точки зору їхньої здатності забезпечити масштабованість веб-додатків. У дослідженні зроблено акцент на інтеграцію цих технологій та їхній вплив на продуктивність і витрати.

Для вибору джерел застосовувалися такі критерії, як актуальність, авторитетність, об'єктивність та достовірність інформації. Усі джерела є рецензованими науковими роботами, опублікованими у 2022–2024 роках, що гарантує сучасний підхід до аналізу проблеми. Вони висвітлюють як фундаментальні аспекти, так і практичні виклики, що робить їх цінним внеском у контекст даного дослідження. Узагальнення представлених результатів дозволяє сформувати комплексне уявлення про стан і перспективи оптимізації хмарних веб-додатків у середовищі Kubernetes.

1.3.2 Аналіз літератури

Сучасні дослідження у сфері хмарних технологій і контейнеризації акцентують увагу на важливості використання оркестраційних платформ, таких як Kubernetes, для забезпечення ефективності, безпеки та продуктивності додатків. У процесі аналізу наукової літератури було розглянуто основні концепції, теорії та моделі, що стосуються оптимізації продуктивності у хмарних середовищах, із особливим фокусом на контейнеризацію.

Дослідження "Kubernetes and Docker Load Balancing: State-of-the-Art Techniques and Challenges" (2024) окреслює сучасні техніки балансування навантаження у середовищах Kubernetes та Docker, зокрема автоматизацію розподілу запитів і використання алгоритмів машинного навчання для прогнозування навантаження. У цьому контексті акцент зроблено на виклики,

пов’язані з динамічністю хмарних середовищ, та запропоновано стратегії для їх подолання.

Робота "Enhanced Visibility for Real-time Monitoring and Alerting in Kubernetes by Integrating Prometheus, Grafana, Loki, and Alerta" (2023) досліджує інтеграцію інструментів для моніторингу та аналізу продуктивності. Основна увага приділена забезпеченню прозорості роботи кластерів у реальному часі, що дозволяє швидко реагувати на збої та знижувати час простою. Автори оцінюють ефективність підходу як високу, особливо для великих інфраструктур.

У статті "Kubernetes in Microservices" (2022) розглянуто використання Kubernetes у мікросервісних архітектурах. Автори акцентують увагу на здатності Kubernetes забезпечувати незалежне масштабування сервісів, а також на методах покращення комунікації між мікросервісами. У результаті аналізу було визначено ключові переваги платформи для мікросервісних додатків, а також вказано на можливості для вдосконалення.

Дослідження "Optimizing Container Management with Kubernetes on Linux: Key Strategies and Common Obstacles" (2024) зосереджується на стратегіях оптимізації управління контейнерами, таких як ефективне використання ресурсів, впровадження політик планування та усунення технічних обмежень операційної системи. Основні висновки підкреслюють важливість належного налаштування ресурсів та інтеграції Kubernetes із хмарними платформами.

Стаття "Navigating the Landscape of Kubernetes Security Threats and Challenges" (2023) охоплює аспекти безпеки, включаючи аналіз вразливостей у контейнерах, налаштування доступу та управління сертифікатами. Автори наголошують на потребі у вдосконаленні систем безпеки для забезпечення захисту від атак на рівні кластерів.

Нарешті, у роботі "Containerization in Cloud Computing: Comparing Docker and Kubernetes for Scalable Web Applications" (2022) проведено порівняння Docker і Kubernetes з точки зору забезпечення масштабованості додатків. Автори оцінюють їхню ефективність для різних типів робочих навантажень, зокрема підкреслюють, що Kubernetes має переваги у випадках складної інфраструктури.

Огляд показав, що більшість досліджень акцентують увагу на практичних аспектах використання Kubernetes, враховуючи оптимізацію продуктивності, безпеку та управління ресурсами. Використані методи, такі як моніторинг у реальному часі, автоматичне масштабування і політики розподілу ресурсів, є ключовими для сучасних хмарних додатків. Однак результати також свідчать про існування низки викликів, які потребують подальших досліджень, зокрема у сфері інтеграції та адаптації Kubernetes до специфічних потреб організацій.

1.3.3 Оцінка актуальності та новизни

Оцінка наукових джерел, що стосуються оптимізації продуктивності контейнеризованих веб-додатків у Kubernetes, дозволяє визначити сучасний стан проблеми та виявити прогалини для подальшого дослідження. У роботі "Kubernetes and Docker Load Balancing: State-of-the-Art Techniques and Challenges" акцент зроблено на сучасних методах балансування навантаження, що підтверджує актуальність теми в умовах зростаючих обсягів трафіку. Дослідження "Enhanced Visibility for Real-time Monitoring and Alerting in Kubernetes" демонструє нові інтеграції для підвищення ефективності моніторингу, що є ключовим аспектом у забезпеченні стабільності системи.

Праця "Kubernetes in Microservices" розкриває особливості використання Kubernetes у мікросервісній архітектурі, що підкреслює його важливість для сучасних розробок. Дослідження "Optimizing Container Management with Kubernetes on Linux" акцентує увагу на стратегіях оптимізації управління контейнерами, що робить його цінним джерелом для цієї роботи. Новизна у сфері безпеки розкрита в статті "Navigating the Landscape of Kubernetes Security Threats", яка підкреслює важливість протидії сучасним кіберзагрозам.

Таким чином, огляд літератури свідчить про те, що проблема оптимізації продуктивності контейнеризованих додатків є актуальною та багатогранною. Розглянуті джерела забезпечують різносторонній підхід до вивчення теми та вказують на ключові аспекти для подальшого дослідження.

1.3.4 Висновки з огляду

На основі проведеного огляду літератури можна зробити кілька ключових висновків, які суттєво впливають на розвиток теми дослідження. По-перше, сучасні дослідження підтверджують, що контейнеризація, зокрема за допомогою Kubernetes, залишається критично важливим компонентом для управління хмарними веб-додатками. Методи, описані у джерелах, забезпечують підвищення продуктивності, зменшення затримок і покращення ефективності використання ресурсів. Наприклад, інтеграція інструментів моніторингу, таких як Prometheus та Grafana, значно спрощує управління системами та знижує ризики збоїв.

Попри значні досягнення, існує низка невирішених проблем, що стосуються як оптимізації використання ресурсів, так і підвищення безпеки в контейнеризованих середовищах. Більшість робіт зосереджені на окремих аспектах, таких як балансування навантаження або захист від кіберзагроз, але бракує інтегрованого підходу, який охоплює всі етапи життєвого циклу додатків. Прогалини також спостерігаються у вирішенні проблем з автоматичним масштабуванням і адаптацією до змін у робочих навантаженнях.

Ці висновки підтверджують необхідність подальших досліджень, які б розширили існуючі підходи до оптимізації та вирішили ключові виклики, зазначені в оглянутих джерелах. Зокрема, важливо інтегрувати сучасні рішення у єдину модель, яка буде ефективною для різних типів додатків. Таким чином, дослідження, представлене у цій науково-дослідній роботі, заповнює ці прогалини, роблячи внесок у вдосконалення контейнеризації у хмарних веб-додатках.

2 ПОСТАНОВКА ЗАДАЧІ

Метою дослідження є розробка та аналіз методів оптимізації продуктивності хмарних веб-додатків, контейнеризованих у Kubernetes. Це передбачає вивчення існуючих підходів до налаштування ресурсів, впровадження масштабування та профілювання додатків, а також інтеграцію інструментів моніторингу для підвищення ефективності використання обчислювальної інфраструктури. Особливу увагу приділено виявленню чинників, що впливають на продуктивність додатків, і пошуку способів їх усунення.

Дослідження спрямоване на визначення найкращих практик для розробників і адміністраторів, які дозволяють забезпечити високу доступність, стабільність і якість обслуговування додатків у хмарному середовищі.

Для досягнення поставленої мети необхідно вирішити такі завдання:

– провести детальний аналіз існуючих підходів до оптимізації продуктивності веб-додатків у контейнеризованих середовищах, орієнтуючись на Kubernetes;

– дослідити методи профілювання додатків і моніторингу ресурсів, такі як використання інструментів Prometheus, Grafana та інших рішень для аналізу продуктивності;

– виявити основні фактори, що впливають на продуктивність веб-додатків, та оцінити їхній вплив у різних сценаріях навантаження;

– розробити та протестувати кілька підходів до оптимізації налаштувань ресурсів контейнерів, масштабування та балансування навантаження;

– провести порівняльний аналіз результатів оптимізації за запропонованими методиками, визначивши найефективніші з них;

– розробити рекомендації щодо впровадження найкращих практик для підвищення продуктивності хмарних веб-додатків у контейнеризованих середовищах.

Ці завдання забезпечують систематичний підхід до досягнення мети дослідження та сприяють виявленню інноваційних рішень у галузі оптимізації

продуктивності хмарних систем. Визначення завдань дослідження безпосередньо впливає на досягнення кінцевих результатів і відображає логічну послідовність дій, необхідних для вирішення основної проблеми. Кожне завдання розглядається як окремий етап, який робить свій внесок у загальний успіх проекту, а їх виконання забезпечує досягнення мети дослідження.

Кожне завдання узгоджується із відповідними очікуваними результатами, створюючи чіткий взаємозв'язок між теоретичними, практичними та кінцевими етапами дослідження. Це забезпечує досягнення мети проєкту та створення цінності для широкого кола користувачів.

Для досягнення мети дослідження обрано низку методів і підходів, які забезпечують систематичний і комплексний аналіз продуктивності хмарних веб-додатків у контейнеризованих середовищах. Основним критерієм вибору методів є їх відповідність актуальним завданням дослідження, ефективність у реальних сценаріях застосування та здатність адаптуватися до динамічних умов роботи сучасних хмарних платформ.

Перш за все, використання Kubernetes як платформи для оркестрації контейнерів обумовлено її широким розповсюдженням і функціональністю, що дозволяє ефективно управляти розгортанням, масштабуванням і ресурсами додатків. Kubernetes надає вбудовані механізми для реалізації горизонтального та вертикального масштабування, управління мережевими запитами, а також моніторингу стану контейнерів, що робить його ідеальним інструментом для дослідження.

Для аналізу та оптимізації продуктивності обрано інструменти моніторингу та профілювання, такі як Prometheus, Grafana і Loki. Ці інструменти забезпечують збір та візуалізацію метрик у режимі реального часу, дозволяючи виявляти вузькі місця у роботі додатків і приймати рішення щодо оптимізації. Використання цих рішень виправдане їхньою інтеграцією з Kubernetes і здатністю працювати в масштабованих середовищах.

Обмеження дослідження включають залежність результатів від обраної платформи Kubernetes і специфіки інфраструктури, що використовується. Однак

обрані методи дозволяють мінімізувати ці обмеження завдяки універсальності інструментів і можливості адаптації до різних умов.

3 СТВОРЕННЯ ПРОГРАМНОЇ СИСТЕМИ ТА ПІДГОТОВКА ДО ЕКСПЕРИМЕНТУ

3.1 Архітектура та проектування системи

Для дослідження методів оптимізації продуктивності хмарних веб-додатків контейнеризованих у Kubernetes, було обрано створити монолітний веб-додаток для інтернет-магазину у якості першої ітерації програмного продукту. Після застосування методів оптимізації, початкова архітектура програмного продукту зміниться.

Усі функції системи, включаючи бізнес-логіку, інтерфейс користувача, управління базою даних та інтеграцію зі сторонніми сервісами, реалізовані як єдиний програмний модуль (див. рис. 3.1.1).

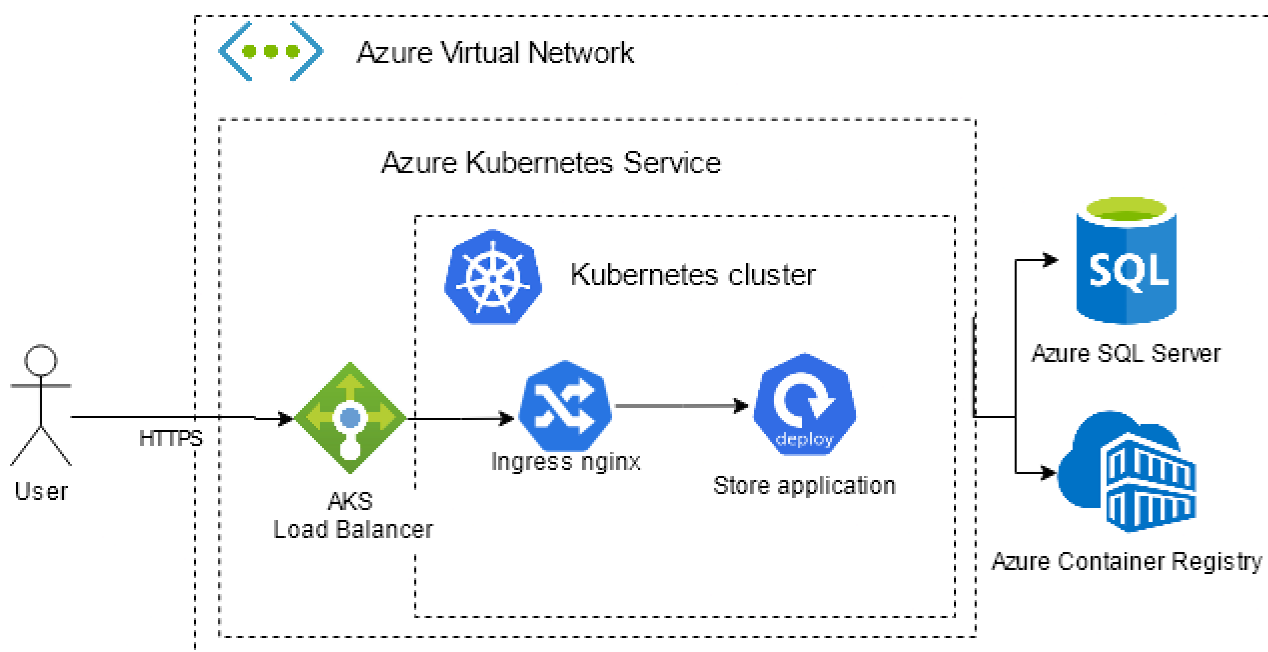


Рисунок 3.1.1 – Початкова архітектура монолітного додатку (рисунок створено самостійно)

Веб-додаток спілкуються з Azure Virtual Network через DNS домен, відправляючи запити для обробки. Azure Virtual Network відповідає за фільтрацію цих запитів, допускаючи тільки ті, які походять з авторизованих IP-адрес.

Після того, як запити перевірені Azure Virtual Network, вони надходять до AKS Load Balancer, що є частиною Azure Kubernetes Service. Load Balancer забезпечує спілкування з Ingress nginx.

Ingress nginx аналізує отримані запити і перенаправляє їх до серверу веб-додатку для обробки та виконання.

Веб-додаток використовує Azure Container Registry та Azure SQL Server, обидва розташовані в межах однієї Azure Virtual Network. Azure Container Registry служить місцем зберігання для нових Docker образів. Azure SQL Server, з іншого боку, використовується як база даних для веб-додатку, забезпечуючи надійне та гнучке зберігання даних.

Основними компонентами монолітної архітектури є інтерфейс користувача, серверна частина, база даних та механізми інтеграції. Інтерфейс користувача дозволяє клієнтам переглядати товари, додавати їх до кошика та оформляти замовлення. Серверна частина виконує функції обробки запитів та взаємодії з базою даних. База даних зберігає інформацію про товари, користувачів та замовлення. Інтеграція зі сторонніми API забезпечує взаємодію з платіжними системами.

Розробка монолітної системи базується на використанні сучасних технологій, що сприяють ефективному створенню, тестуванню та розгортанню програмного забезпечення. Серверна частина реалізована мовою Golang, який забезпечує зручність побудови хмарних додатків.

Для створення інтерактивного користувацького інтерфейсу застосовується React.js, що полегшує взаємодію користувачів із системою. Контейнеризація здійснюється за допомогою Docker, що забезпечує простоту розгортання системи у різних середовищах.

Use Case діаграма демонструє основні взаємодії користувачів із системою (див. рис 3.1.2).

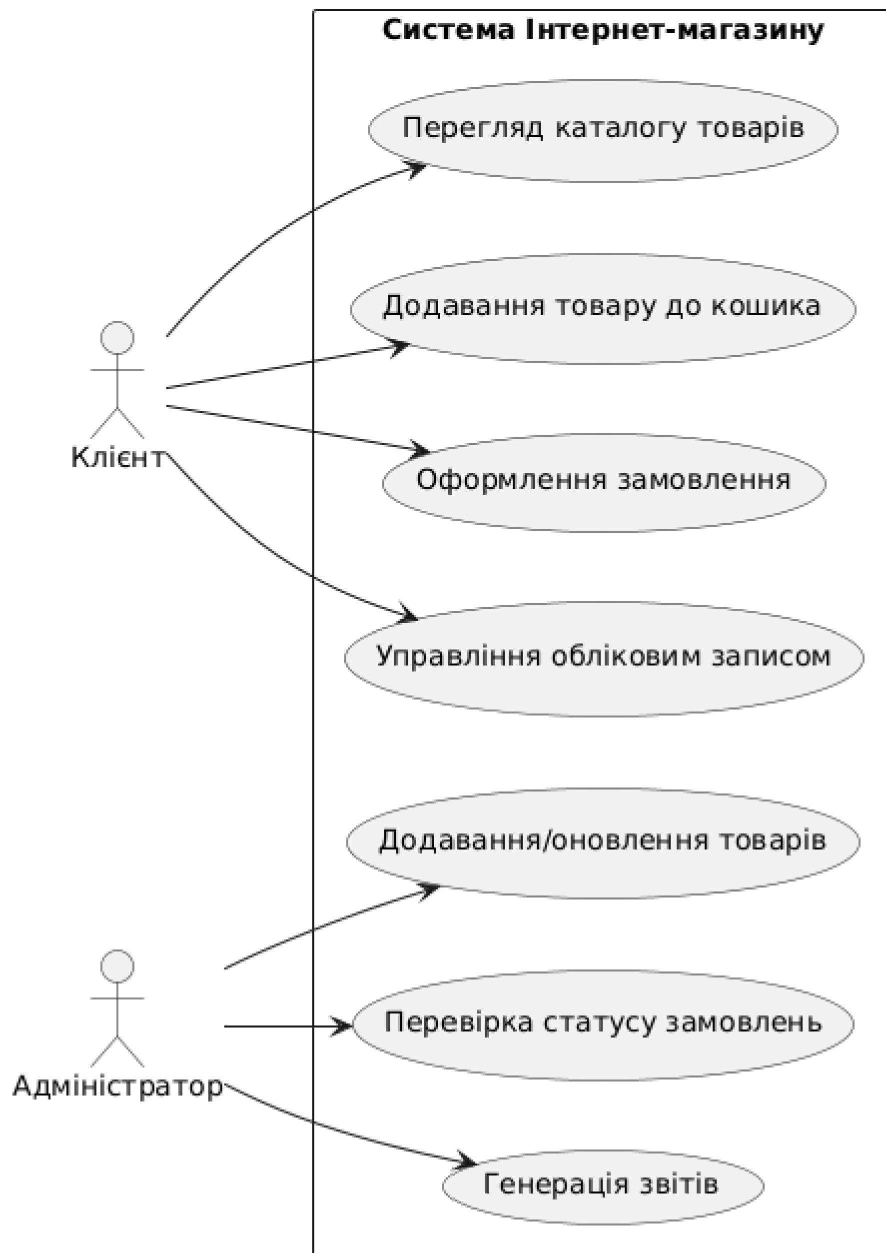


Рисунок 3.1.2 – Use case діаграма (рисунок створено самостійно)

Користувачі:

– перегляд каталогу товарів;

– додавання товарів до кошика;

– оформлення замовлення;

– управління обліковим записом.

Адміністратори:

– управління товарами;

– перевірка статусів замовлень;

– генерація звітів.

Діаграма класів (UML) описує основні класи системи та їх взаємодії (див. рис. 3.1.3).

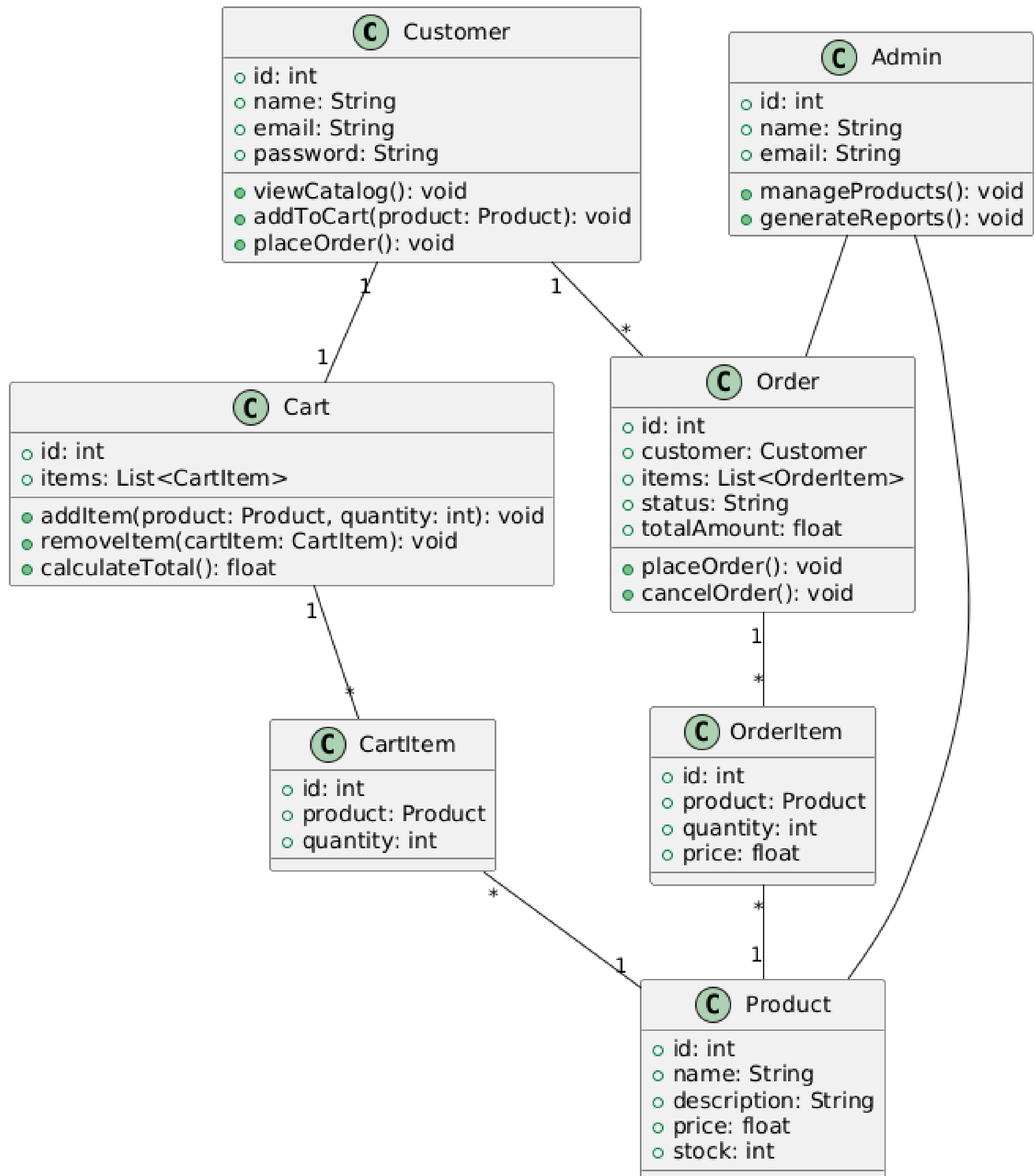


Рисунок 3.1.3 – Діаграма класів (UML) (рисунок створено самостійно)

До класів належать клас товару (Product), який містить атрибути, такі як назва, ціна та опис, клас користувача (User), що включає дані про імена, адреси та

облікові записи, а також класи замовлення (Order) і кошика (Cart), що забезпечують функціональність замовлення та збереження товарів. Опишемо кожний клас.

Клас Customer представляє користувачів магазину.

Атрибути:

– id: унікальний ідентифікатор клієнта;

– name: ім'я клієнта;

– email: адреса електронної пошти клієнта;

– password: пароль для авторизації.

Методи:

– viewCatalog(): перегляд каталогу товарів;

– addToCart(product: Product): додавання товару до кошика;

– placeOrder(): оформлення замовлення.

Клас Product описує товари, доступні в магазині.

Атрибути:

– id: унікальний ідентифікатор товару;

– name: назва товару;

– description: опис товару;

– price: ціна товару;

– stock: кількість товару на складі.

Клас Cart моделює кошик, до якого клієнти додають товари.

Атрибути:

– id: унікальний ідентифікатор кошика;

– items: список об’єктів класу CartItem, які представляють товари в кошику.

Методи:

– addItem(product: Product, quantity: int): додавання товару до кошика з вказаною кількістю;

– removeItem(cartItem: CartItem): видалення товару з кошика;

– calculateTotal(): розрахунок загальної суми товарів у кошику.

Клас CartItem представляє товар у кошику.

Атрибути:

– id: унікальний ідентифікатор товару в кошику;
– product: об’єкт класу Product;
– quantity: кількість товару.

Клас Order моделює замовлення, оформлене клієнтом.

Атрибути:

– id: унікальний ідентифікатор замовлення;
– customer: об’єкт класу Customer, який оформив замовлення;
– items: список об’єктів класу OrderItem, які представляють товари в замовленні;
– status: статус замовлення (наприклад, "оформлено", "відправлено", "доставлено");
– totalAmount: загальна сума замовлення.

Методи:

– placeOrder(): оформлення замовлення;
– cancelOrder(): скасування замовлення.

Клас OrderItem описує товар, що входить до замовлення.

Атрибути:

– id: унікальний ідентифікатор товару в замовленні;
– product: об’єкт класу Product;
– quantity: кількість товару;
– price: ціна товару.

Клас Admin представляє адміністратора системи, який управляє товарами та замовленнями.

Атрибути:

– id: унікальний ідентифікатор адміністратора;
– name: ім'я адміністратора;
– email: адреса електронної пошти адміністратора.

Методи:

– manageProducts(): управління товарами (додавання, редагування, видалення);

– generateReports(): генерація звітів.

Взаємозв’язки між класами:

– Customer — Cart: кожен клієнт має один кошик, де зберігаються обрані товари;

– Cart — CartItem: кошик містить багато товарів (CartItem), кожен з яких пов’язаний із певним товаром (Product);

– Customer — Order: кожен клієнт може оформити кілька замовлень;

– Order — OrderItem: замовлення містить товари (OrderItem), які також пов’язані з класом Product;

– Admin — Product, Admin — Order: адміністратор керує товарами та замовленнями через відповідні методи.

Діаграма класів монолітної архітектури чітко демонструє місця потенційної оптимізації через декомпозицію. Наприклад, класи Order, Cart, і Product можуть бути розділені на окремі мікросервіси для кращої масштабованості.

Схема бази даних складається з таблиць для зберігання інформації про товари, користувачів, замовлення та кошик (див. рис. 3.1.4).

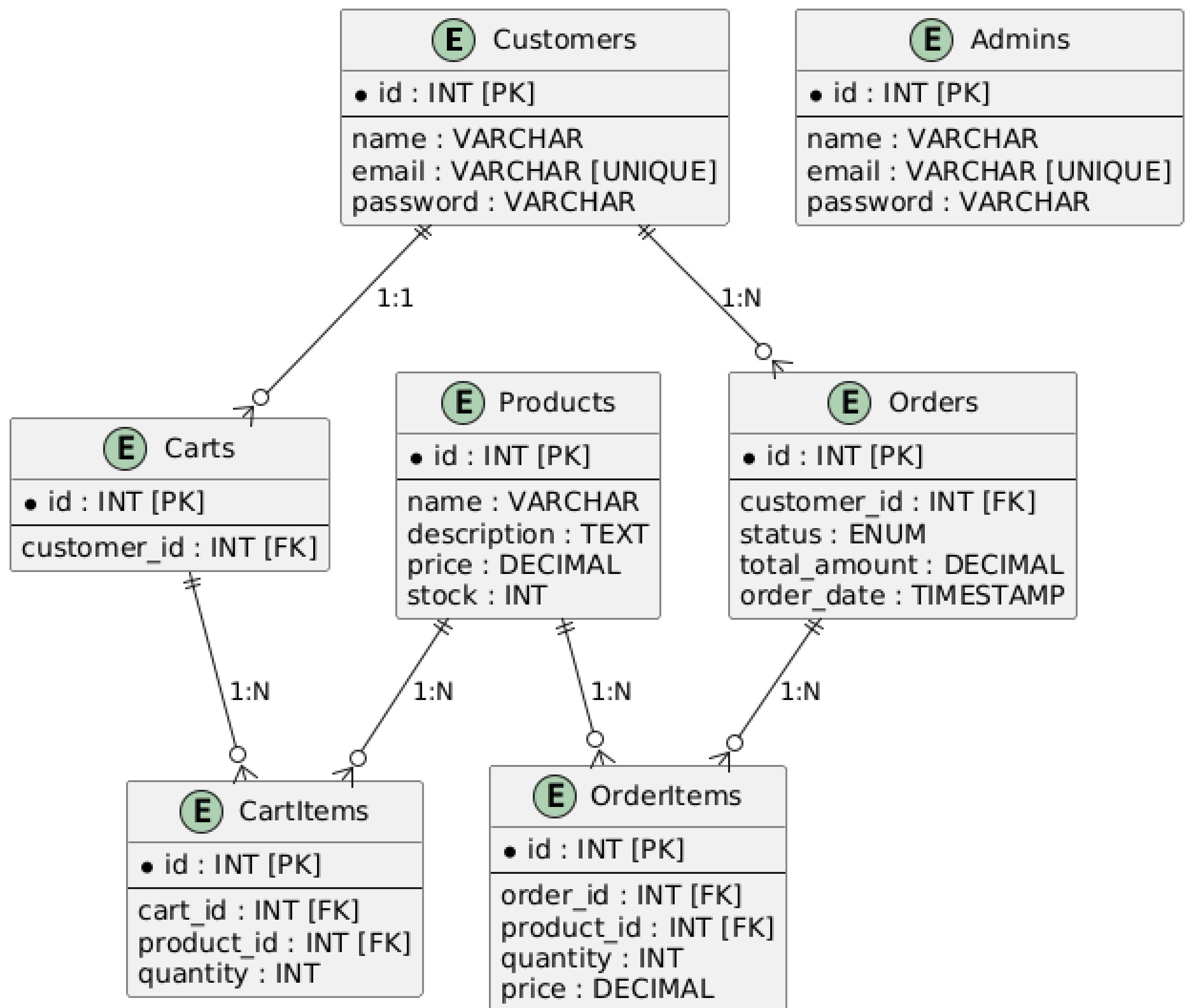


Рисунок 3.1.4 – Схема бази даних (рисунок створено самостійно)

Схема бази даних схожа на діаграму класів (UML) за структурою. Атрибути та взаємозв’язки даних однакові на обох діаграмах.

Монолітна архітектура має свої обмеження. Зокрема, будь-які зміни в одній частині додатку можуть призвести до впливу на інші компоненти, що ускладнює масштабування та тестування. Також, під час пікових навантажень можуть виникати проблеми з продуктивністю, оскільки всі функції додатку працюють в одному середовищі.

Доцільно буде оптимізувати шляхом переходу до мікросервісної архітектури, що дасть змогу забезпечити масштабованість, гнучкість і стабільність роботи додатку.

3.2 Оптимізація системи

Одним із головних переваг переходу до мікросервісної архітектури є розподіл навантаження між окремими сервісами. Функціональність інтернет-магазину можна розподілити між сервісами для управління товарами, обробки замовлень, роботи кошика та аутентифікації користувачів (див. рис. 3.2.1). Це дозволяє масштабувати кожен сервіс окремо залежно від його навантаження, що особливо корисно в періоди пікових розпродажів, коли потреба в обробці замовлень значно зростає.

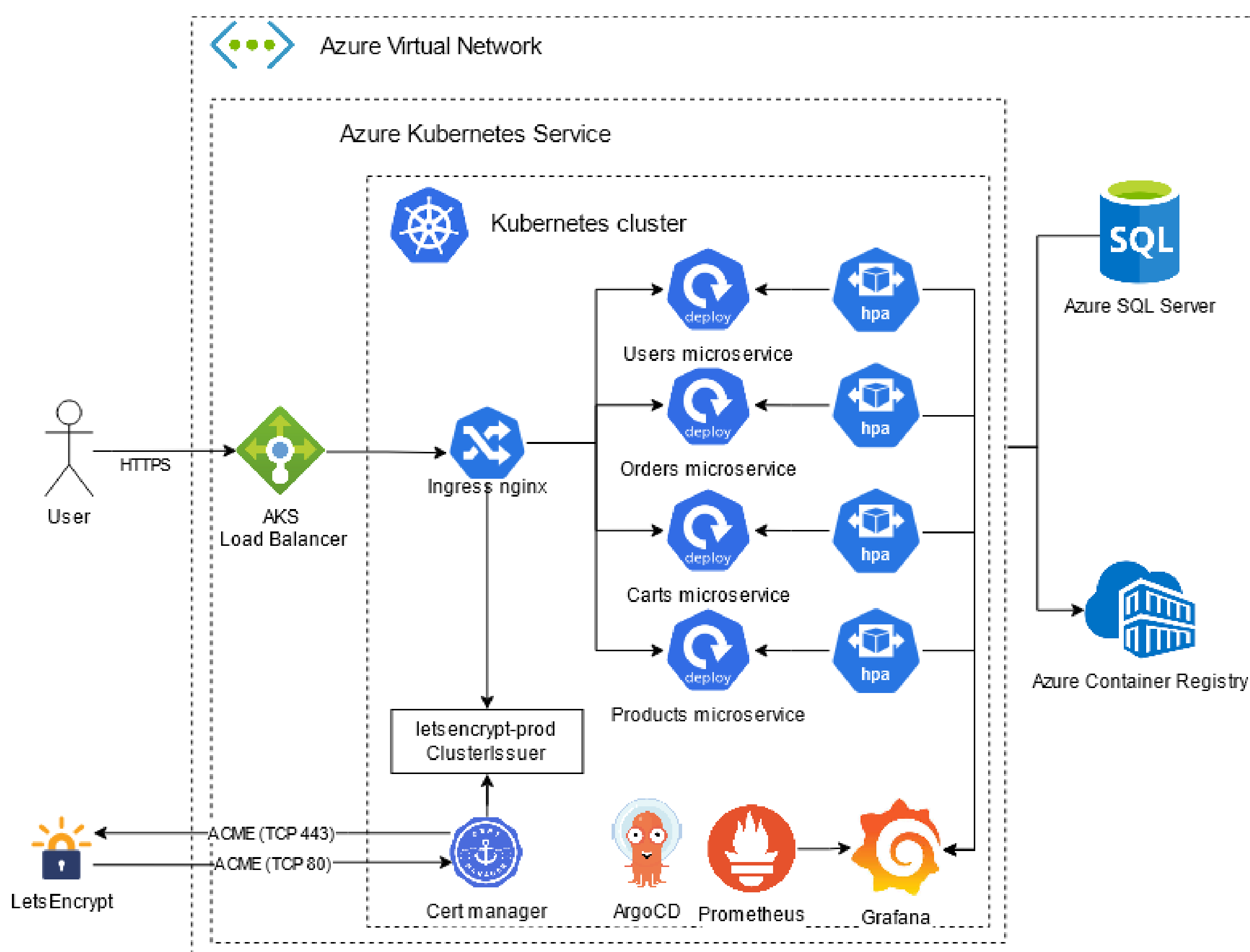


Рисунок 3.2.1 – Високорівнева архітектура мікросервісної інфраструктури (рисунок створено самостійно)

При переході до мікросервісної архітектури буде доречно інтегрувати систему сертифікації Cert-manager для захисту запитів мікросервісів. Cert-manager

відіграє критичну роль у системі, автоматизуючи управління сертифікатами TLS/SSL. Ці сертифікати є ключовими компонентами для забезпечення безпечного з'єднання між клієнтами (мобільним та веб додатками) та серверною частиною, розміщеною в Kubernetes кластері. Постійне оновлення і підтримка цих сертифікатів є важливим аспектом безпеки, а Cert-manager автоматизує цей процес, що значно спрощує життєвий цикл управління сертифікатами.

Ingress ресурс в Kubernetes кластері безпосередньо взаємодіє з ClusterIssuer Cert-manager. ClusterIssuer у Cert-manager - це сутність, яка дозволяє віддавати сертифікати на весь кластер, а не просто на окремі простори імен (namespaces).

Ця конфігурація дозволяє гарантувати, що Ingress завжди має актуальні та валідні сертифікати. Це забезпечує високий рівень безпеки для всіх вхідних з'єднань до системи.

Важливо зазначити, що у даному проекті Cert-manager використовує LetsEncrypt як авторитетний центр видачі сертифікатів (CA). LetsEncrypt - це вільна, автоматизована, та відкрита служба видачі SSL/TLS сертифікатів для безпечного з'єднання.

Інтеграція з LetsEncrypt дозволяє Cert-manager автоматично видавати та оновлювати SSL/TLS сертифікати для Ingress ресурса у Kubernetes кластері. Таким чином, використання ClusterIssuer у співпраці з Ingress ресурсом є ключовою частиною стратегії безпеки, що забезпечує безпечні та автоматизовані оновлення сертифікатів для всіх вхідних з'єднань.

На кластері розташований Prometheus, система моніторингу та збору метрик, яка працює в парі з Grafana. Grafana - це платформа для візуалізації, моніторингу та аналізу даних, яка дозволяє створювати інтерактивні інформаційні панелі на основі даних з різних джерел Співробітництво Prometheus та Grafana забезпечує збір і агрегацію метрик в реальному часі, що дозволяє відстежувати стан ресурсів Kubernetes, діагностувати проблеми і аналізувати тенденції у використанні ресурсів на протязі часу.

ArgoCD - це інструмент для автоматизованого розгортання за допомогою методології GitOps. Він розташований на кластері і дозволяє автоматизувати процес розгортання додатків у Kubernetes.

Мікросервісна архітектура також сприяє підвищенню стійкості системи. У моноліті збій одного компонента може вплинути на всю систему, тоді як у мікросервісній структурі помилка одного сервісу ізольована та не порушує роботу інших. Наприклад, несправність у сервісі аналітики не вплине на обробку замовлень або роботу кошика.

Ще одним важливим аспектом оптимізації є гнучкість у виборі технологій зберігання даних. Кожен сервіс може використовувати базу даних, найбільш відповідну його завданням.

Перехід до мікросервісної архітектури також спрощує розробку та розгортання системи. Кожен сервіс розробляється незалежно, що дозволяє швидко оновлювати або додавати новий функціонал без необхідності змінювати всю систему. Наприклад, зміни у логіці роботи кошика можна впровадити, не торкаючись інших компонентів.

Використання асинхронної взаємодії між сервісами через gRPC, знижує затримки та підвищує ефективність системи. Обробка платежів може виконуватися у фоновому режимі, що зменшує навантаження на основний потік операцій.

Кожен мікросервіс має свої політики доступу, що зменшує ризики компрометації даних. Сервіс аутентифікації може забезпечувати шифрування паролів і токенів доступу, не впливаючи на інші сервіси.

До кожного мікросервісу також можливо додати HPA (Horizontal Pod Autoscaler), який забезпечить горизонтальне масштабування окремо для кожного сервісу. Горизонтальне масштабування передбачає збільшення кількості екземплярів одного й того ж сервісу. Наприклад, якщо сервіс обробки замовлень починає перевантажуватися через велику кількість запитів, можна запустити кілька ідентичних екземплярів цього сервісу, які будуть працювати паралельно. Балансувальник навантаження розподіляє запити між цими екземплярами, забезпечуючи рівномірну обробку запитів.

Узагальнюючи, перехід до мікросервісної архітектури створює умови для ефективного управління ресурсами, швидкого впровадження інновацій і підтримки стабільної роботи системи навіть у періоди високого навантаження. Це рішення є ідеальним для масштабних проєктів, таких як інтернет-магазини, які вимагають високої продуктивності та надійності.

4 ПРОВЕДЕННЯ ЕКСПЕРИМЕНТІВ ТА АНАЛІЗ ОТРИМАНИХ РЕЗУЛЬТАТІВ

4.1 Підготовка до проведення експериментів

За допомогою додатку OpenLens, можна проаналізувати стан кластеру з розгорнутими деплойментами додатку (див. рис. 4.1.1)

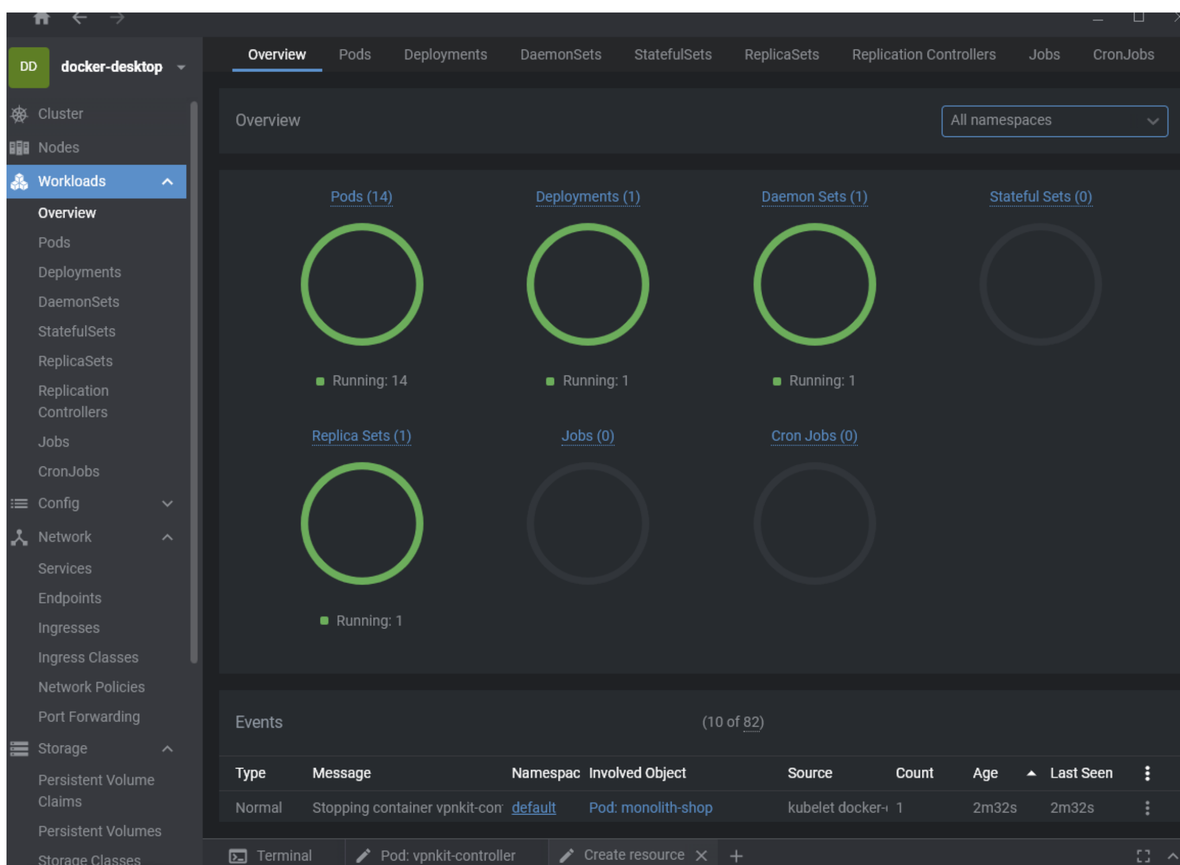


Рисунок 4.1.1 – Загальний стан кластеру (рисунок створено самостійно)

Монолітний додаток представлено у вигляді одного поду “monolith-shop” у просторі імен “monolith-shop” (див. рис. 4.1.2):

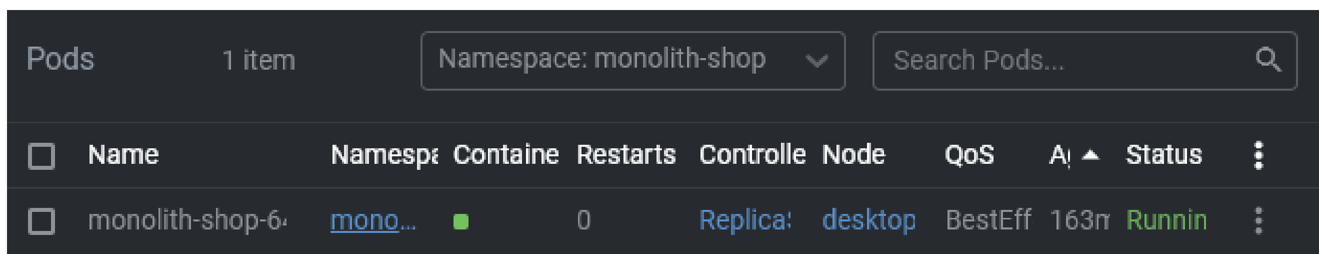


Рисунок 4.1.2 – Розгорнутий монолітний додаток (рисунок створено самостійно)

З метою імітації реального навантаження було використано інструмент hey, який дозволяє генерувати велику кількість HTTP-запитів на заданий маршрут. Під час тестування надсилалося до 1000 одночасних запитів на маршрути /products, /order та /cart. Це дозволило перевірити стійкість додатку до навантаження, а також оцінити вплив на ресурси.

У процесі оцінювання методів оптимізації було обрано п'ять основних критеріїв (див. табл. 4.1.1), які найкраще відображають реальні вимоги до сучасних хмарних застосунків. Кожному критерію було присвоєно ваговий коефіцієнт, що відображає його пріоритетність у загальній системі оцінювання.

Масштабованість (Scalability). Цей критерій отримав найвищий ваговий коефіцієнт — 0.35, оскільки здатність системи масштабуватись відповідно до навантаження є критично важливою для стабільної роботи у хмарному середовищі. Підвищення кількості користувачів, запитів або оброблюваних даних не повинно призводити до зниження продуктивності. Масштабованість оцінюється за здатністю системи адаптивно збільшувати або зменшувати кількість інстансів сервісів.

Надійність (Reliability). Ваговий коефіцієнт — 0.25. Цей критерій відображає здатність системи залишатися функціональною та доступною навіть у разі часткових відмов або перевищення навантаження. Надійність є важливою для забезпечення безперервності бізнес-процесів, особливо у розподілених мікросервісних архітектурах.

Гнучкість підтримки (Maintainability). Ваговий коефіцієнт — 0.15. Гнучкість підтримки визначає легкість внесення змін до коду, налаштувань чи конфігурації сервісів. У контексті DevOps-практик це дозволяє швидко впроваджувати оновлення, виправлення помилок або нові функціональні можливості. Висока підтримуваність є ключовою для скорочення часу обслуговування та підвищення продуктивності команд розробки.

Вартість ресурсів (Resource Cost). Ваговий коефіцієнт — 0.15. Під цим критерієм мається на увазі ефективність використання апаратних ресурсів (CPU, пам'ять), що безпосередньо впливає на вартість експлуатації інфраструктури.

Оптимізація споживання ресурсів дозволяє зменшити витрати без втрати продуктивності.

Час розгортання (Deployment Time). Ваговий коефіцієнт — 0.10. Цей показник є особливо актуальним у середовищах із частими релізами та CI/CD-пайплайнами. Чим швидше можна розгорнути оновлення — тим менший ризик виникнення збоїв і тим вищою є реакція на бізнес-потреби. Незважаючи на нижчий ваговий коефіцієнт, цей критерій відіграє суттєву роль у забезпеченні гнучкості.

Таблиця 4.1.1 – Таблиця критеріїв оцінювання (таблиця виконана самостійно)

Критерій	Обґрунтування пріоритету	Ваговий коефіцієнт
Масштабованість (Scalability)	Дає змогу системі обробляти зростаючі обсяги трафіку або даних.	0.35
Надійність (Reliability)	Забезпечує безперервну доступність сервісу під час навантажень або збоїв.	0.25
Гнучкість підтримки (Maintainability)	Визначає легкість модифікацій та оновлень додатку.	0.15
Вартість ресурсів (Resource Cost)	Показує економічну ефективність: скільки ресурсів потрібно для досягнення стабільної роботи.	0.15
Час розгортання (Deployment Time)	Важливо для CI/CD процесів. Чим швидше — тим більше гнучкості у релізах.	0.1
РАЗОМ		1.00

Дана система критеріїв дозволяє здійснити багатокритеріальне порівняння методів оптимізації з урахуванням як технічних, так і економічних аспектів. Вагові коефіцієнти підібрані відповідно до важливості кожного параметра у сучасних

хмарних інфраструктурах. Такий підхід забезпечує об'єктивність при подальшому порівняльному аналізі HPA, VPA та мікросервісної архітектури.

4.2 Метод оптимізації за допомогою горизонтального автоскейлінгу

З метою оцінки ефективності використання горизонтального автоскейлінгу у середовищі Kubernetes було проведено стрес-тестування веб-додатку (див. рис. 4.2.1), після чого здійснено збір та аналіз ключових експлуатаційних метрик.

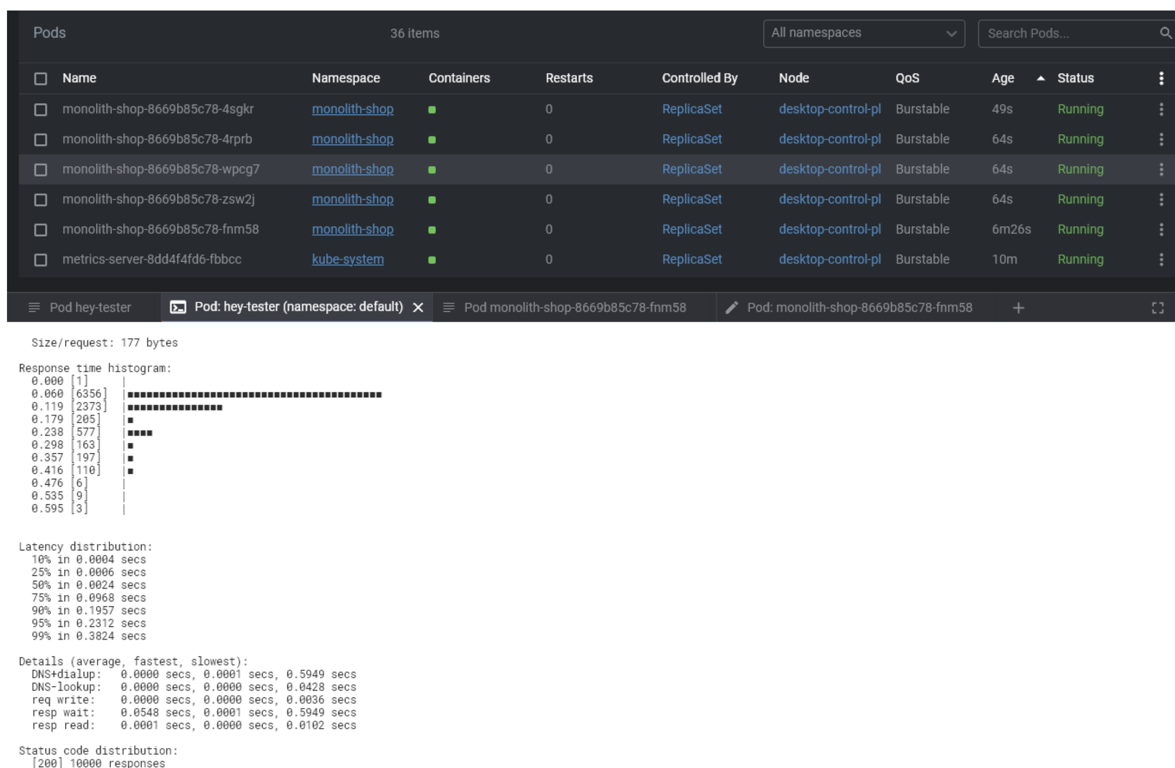


Рисунок 4.2.1 – Стрес тестування методу оптимізації горизантального автоскейлінгу (рисунок створено самостійно)

Як можна побачити з скриншоту, під час стрес тестування, поди додатку збільшились за кількістю. Дана поведінка можлива за допомогою ресурсу HPA (Horizontal Pod Autoscaler) на кластері (див. рис. 4.2.2).

Horizontal Pod Autoscalers 1 item All namespaces Search Horizontal Pod Autosc

Name	Namespace	Metrics	Min Pods	Max Pods	Replicas	Age	Status
monolith-shop	monolith-...	48% / 80%	1	100	5	8m45s	AbleToScale S

Рисунок 4.2.2 – Horizontal Pod Autoscaler (рисунок створено самостійно)

Під час стрес тесту ресурси додатку (навантаження процесору та оперативної пам’яті) змінились, що помітив HPA та почав збільшувати кількість подів на кластері. Треба зазначити, що ці метрики надсилає metrics-server – один з системних компонентів.

До дослідження було включено показники пропускної спроможності, використання процесорного часу (CPU) та використання оперативної пам’яті.

Першим та найбільш вагомим критерієм, який було обрано для аналізу, є пропускна спроможність додатку (див. рис. 4.2.3). У ході тестування було зафіксовано значне зростання швидкості обробки запитів — з початкових 88.6 B/s до 22.1 KiB/s після активації автоскейлінгу. Це свідчить про ефективну реакцію системи на зростання навантаження, що дозволяє зробити висновок про високий рівень продуктивності в даному аспекті.

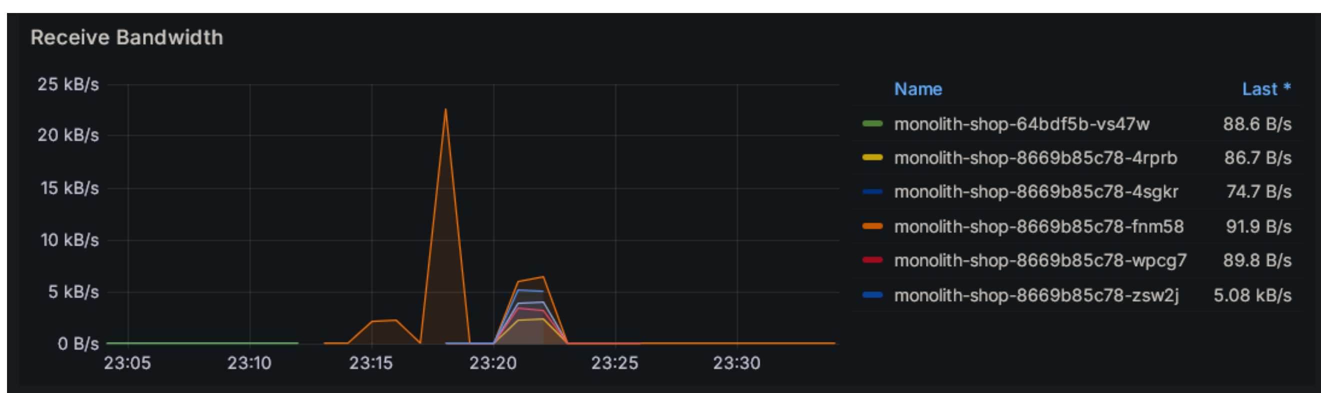


Рисунок 4.2.3 – Пропускна спроможність (рисунок створено самостійно)

Наступним критерієм оцінювання стало споживання процесорного часу(див. рис. 4.2.4). Аналіз CPU-метрик показав, що навантаження було рівномірно розподілено між новоствореними подами, а використання CPU залишалося в межах

2–3 % одного ядра. Це вказує на ефективну роботу системи автоскейлінгу, яка дозволяє уникнути перевантаження окремих компонентів.

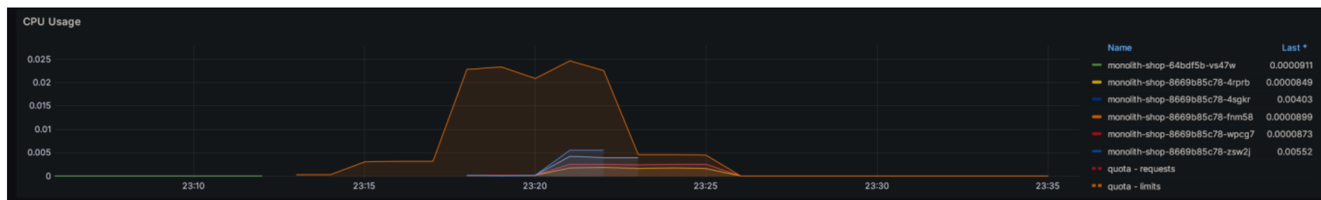


Рисунок 4.2.4 – Навантаження процесору на кластері (рисунок створено самостійно)

Також було проаналізовано використання оперативної пам’яті кожним з подів (див. рис. 4.2.5). Значення пам’яті коливалися у межах від 7 до 19 МіБ, що є прийнятним та свідчить про стабільність роботи компонентів. Важливо зазначити, що не спостерігалося різких стрибків у споживанні ресурсів навіть під час пікового навантаження.

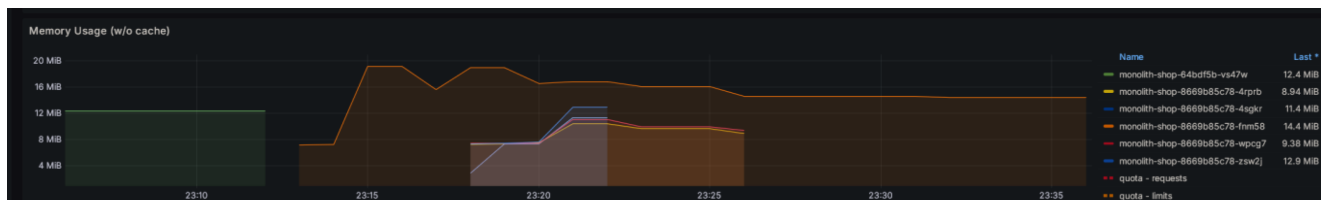


Рисунок 4.2.5 – Навантаження оперативної пам’яті (рисунок створено самостійно)

Особливу увагу було приділено стабільності роботи додатку. За час проведення тестування не було зафіксовано жодного випадку перезапуску контейнерів, що свідчить про високу надійність обраної конфігурації та стабільність механізму автоскейлінгу.

Останнім критерієм аналізу стала масштабованість, яка характеризується здатністю системи створювати додаткові репліки для балансування навантаження. У процесі тестування кількість активних подів збільшилася з одного до шести, що демонструє ефективну реакцію системи на зростання кількості запитів.

Таблиця 4.2.1 – Таблиця оцінки методу (таблиця виконана самостійно)

Критерій	Оцінка ефективності	Ваговий коефіцієнт	Зважена оцінка
Масштабованість (Scalability)	1.00	0.35	0.350
Надійність (Reliability)	0.80	0.25	0.200
Гнучкість підтримки (Maintainability)	0.70	0.15	0.105
Вартість ресурсів (Resource Cost)	0.75	0.15	0.113
Час розгортання (Deployment Time)	0.85	0.10	0.085
Разом			0.853

Метод горизонтального автоскейлінгу (HPA) демонструє високу загальну ефективність із підсумковим зваженим балом 0.853. Найвищу оцінку система отримала за масштабованість (1.00) — це основна сильна сторона HPA, оскільки механізм динамічного масштабування реплік під навантаженням суттєво підвищує адаптивність системи. Також досить добре оцінено надійність (0.80), що свідчить про здатність системи зберігати стабільність при зростанні трафіку.

Оцінки за гнучкість підтримки (0.70) та вартість ресурсів (0.75) є середніми, що пов’язано з необхідністю ручного налаштування метрик і обмеженою точністю при виборі ресурсу масштабування (CPU чи memory). Час розгортання (0.85) отримав позитивну оцінку, оскільки інтеграція HPA у CI/CD-процеси зазвичай не потребує значних зусиль.

Отже, HPA є ефективним методом для автоматизації масштабування у кластері Kubernetes, особливо у системах із динамічним навантаженням.

4.3 Метод оптимізації за допомогою вертикального автоскейлінгу

Для тестування ефективності методу оптимізації за допомогою вертикального автоскейлінгу, треба було створити VPA (Vertical Pod Autoscaler) (див. рис. 4.3.1). VPA збільшує ліміти ресурсів подів на кластері, що дозволяє нам протестувати навантаження тільки одного поду додатку під час стрес тесту. На відміну від горизонтального автоскейлінгу, кількість подів під час навантаження не буде змінюватись.

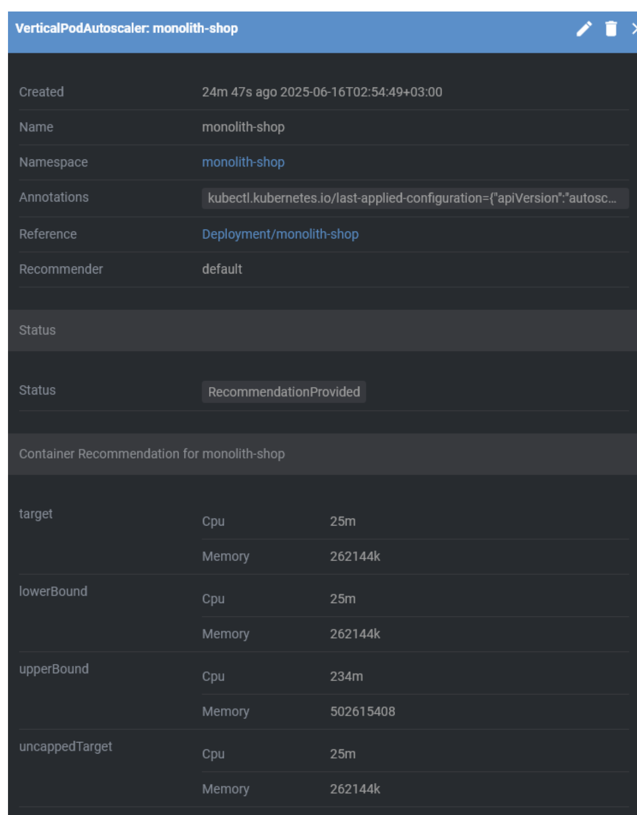


Рисунок 4.3.1 – Vertical Pod Autoscaler (рисунок створено самостійно)

У процесі тестування відстежувались ключові показники: пропускна спроможність, використання процесорного часу (CPU), споживання оперативної пам’яті, стабільність контейнерів та загальна здатність до адаптації під змінні навантаження. Результати аналізу представлені у вигляді багатокритеріальної оцінки.

Першим критерієм оцінювання є пропускна спроможність (див. рис. 4.3.2), яка відображає ефективність обробки вхідного навантаження. Під час

вертикального масштабування максимальний зафіксований показник становив 19.3 KiB/s, що співвідноситься з піковими значеннями під час горизонтального масштабування (22.1 KiB/s). Проте середнє значення було значно нижчим (~93–98 B/s), що вказує на тимчасове, а не стале зростання ефективності. Таким чином, загальна продуктивність системи за цим критерієм є середньою.

Рисунок – 4.3.2 - Пропускна спроможність (рисунок створено самостійно)

Другим показником виступає використання процесорного часу (див. рис. 4.3.3). Аналіз CPU-метрик демонструє, що після короткочасного зростання навантаження (до ~0.025 CPU) ресурси були перерозподілені, і надалі використання залишалося на рівні 0.00009–0.00015. Це свідчить про ефективну дію вертикального автоскейлінгу, який динамічно підвищив ліміти CPU відповідно до навантаження, а потім стабілізував використання. Оцінка цього критерію є високою.

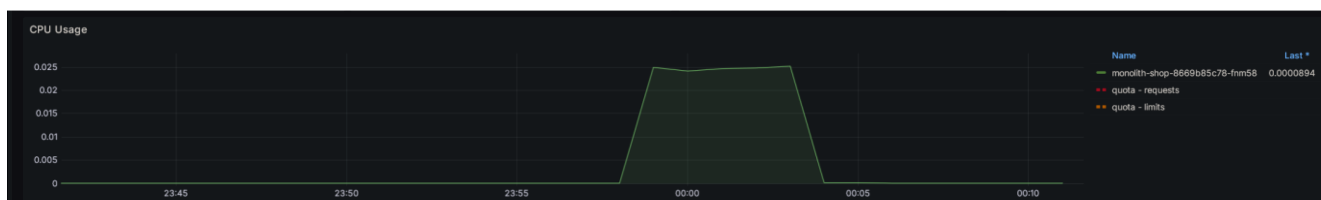


Рисунок 4.3.3 – Навантаження процесору на кластері (рисунок створено самостійно)

Наступним критерієм було споживання оперативної пам’яті (див. рис. 4.3.4). У ході тестування було зафіксовано адаптивне масштабування пам’яті з 14.4 MiB до 18.2 MiB, після чого значення стабілізувалося на рівні 14.9 MiB. Це демонструє здатність вертикального автоскейлінгу забезпечити необхідні ресурси у відповідь на навантаження без перевитрат, що також характеризується позитивно.

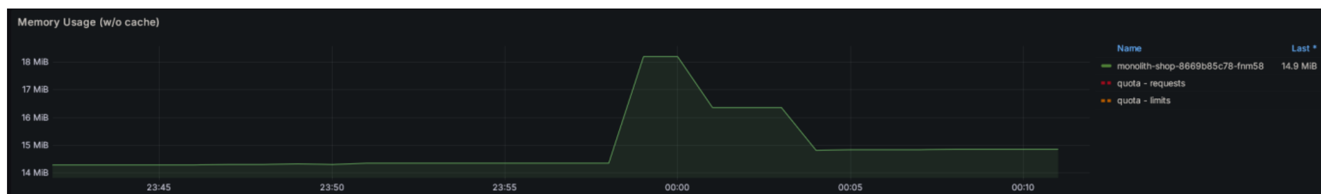


Рисунок 4.3.4 – Навантаження оперативної пам’яті (рисунок створено самостійно)

Щодо стабільності системи, протягом усього тестування не було зафіксовано жодного перезапуску контейнерів. Це свідчить про відсутність аварійного завершення процесів навіть під час пікових навантажень, що дозволяє зробити висновок про високу надійність застосованої конфігурації.

Вертикальний автоскейлінг, на відміну від горизонтального, не створює додаткових екземплярів подів. Проте система ефективно збільшувала доступні ресурси в межах одного поду, що дозволило уникнути непотрібної реплікації. Оцінювання за цим критерієм є обмеженим, але загалом позитивним.

Вертикальне автоскейлювання (VPA) отримало загальну зважену оцінку 0.746, що свідчить про помірний рівень ефективності цього методу масштабування у порівнянні з HPA (див. табл. 4.2.1).

Таблиця 4.2.1 – Таблиця оцінки методу (таблиця виконана самостійно)

Критерій	Оцінка ефективності	Ваговий коефіцієнт	Зважена оцінка
Масштабованість (Scalability)	0.65	0.35	0.228
Надійність (Reliability)	0.85	0.25	0.213

Кінець таблиці 4.2.1

Гнучкість підтримки (Maintainability)	0.60	0.15	0.090
Вартість ресурсів (Resource Cost)	0.90	0.15	0.135
Час розгортання (Deployment Time)	0.80	0.10	0.080
Разом			0.746

Найвищу оцінку VPA отримало за вартість ресурсів (0.90). Це зумовлено тим, що VPA автоматично аналізує історичне навантаження та коригує ресурси (CPU та пам'ять), що дозволяє оптимізувати витрати та зменшити кількість надлишково виділених ресурсів. Також доволі високою виявилась оцінка надійності (0.85), оскільки коректно налаштований VPA сприяє підтримці стабільної роботи додатків без перенавантажень.

Водночас, масштабованість (0.65) отримала порівняно низьку оцінку, адже VPA не змінює кількість реплік, а лише змінює розміри ресурсів окремих подів. Через це VPA менш ефективний у ситуаціях із піковим трафіком, де кількість подів критично важлива. Гнучкість підтримки (0.60) також оцінена нижче, адже зміни у ресурсах вимагають рестарту подів, що може створювати тимчасову недоступність. Час розгортання (0.80) залишається на хорошому рівні, оскільки VPA легко інтегрується з Kubernetes через відповідні контролери.

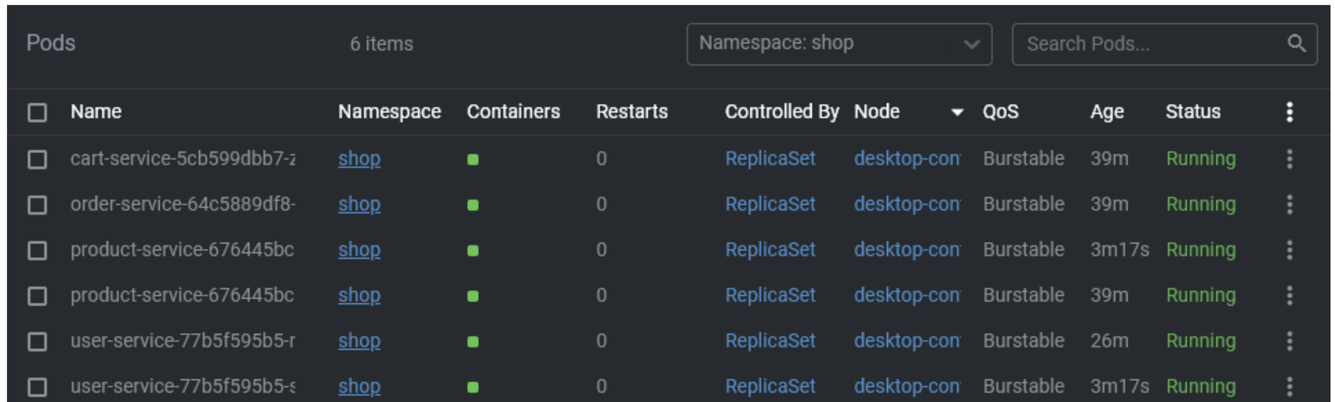
У підсумку, VPA краще підходить для довготривалого ресурсного оптимізування, але поступається HPA за гнучкістю масштабування у реальному часі.

4.4 Метод оптимізації за допомогою мікросервісів

На відміну від монолітної архітектури, кожен под у мікросервісній архітектурі представляє окремий сервіс (див. рис. 4.4.1):

- product-service – сервіс для управління товарами;

- cart-service – сервіс для управління корзинами користувачів;
- order-service – сервіс для обробки замовлень користувачів;
- user-service – сервіс автентифікації користувачів.



Pods 6 items Namespace: shop Search Pods...

Name	Namespace	Containers	Restarts	Controlled By	Node	QoS	Age	Status
cart-service-5cb599dbb7-z	shop	■	0	ReplicaSet	desktop-con	Burstable	39m	Running
order-service-64c5889df8-	shop	■	0	ReplicaSet	desktop-con	Burstable	39m	Running
product-service-676445bc	shop	■	0	ReplicaSet	desktop-con	Burstable	3m17s	Running
product-service-676445bc	shop	■	0	ReplicaSet	desktop-con	Burstable	39m	Running
user-service-77b5f595b5-r	shop	■	0	ReplicaSet	desktop-con	Burstable	26m	Running
user-service-77b5f595b5-s	shop	■	0	ReplicaSet	desktop-con	Burstable	3m17s	Running

Рисунок 4.4.1 – Розгорнутий мікросервісний додаток (рисунок створено самостійно)

Розгортання кожного сервісу у окремому поді забезпечує кращу гнучкість, масштабованість та незалежне оновлення компонентів системи. Кожен сервіс розгорнуто у власному поді, що дозволяє Kubernetes ефективно керувати їхньою життєдіяльністю, включно з автоматичним перезапуском, балансуванням навантаження, масштабуванням та оновленням. Усі сервіси розгорнуті в просторі імен “shop”.

Особливістю цього підходу є поєднання горизонтального масштабування за допомогою Horizontal Pod Autoscaler (HPA) та вертикального масштабування за допомогою Vertical Pod Autoscaler (VPA). Відповідне рішення реалізовано для кожного окремого сервісу: обробки замовлень, управління товарами, керування кошиком та авторизації користувачів. Такий підхід дозволяє досягти високої адаптивності системи до змін у навантаженні при збереженні контрольованого використання ресурсів.

Для об’єктивного порівняння ефективності мікросервісів з autoscaling було проведено моніторинг ключових експлуатаційних метрик за такими критеріями: середнє використання процесора (CPU), споживання оперативної пам’яті (RAM),

пропускна здатність мережі (Receive Bandwidth), масштабованість, гнучкість до навантаження, а також стабільність під час пікових умов.

Середнє навантаження на процесор (див. рис. 4.4.2) в сервісі cart-service було низьким (приблизно 7.3×10^{-7} cores). Аналогічні значення спостерігались і в інших сервісах, з незначним перевищенням у сервісі user-service (до 2.1×10^{-5} cores у пікові моменти). Така поведінка підтверджує ефективність горизонтального масштабування: у моменти зростання навантаження HPA додавав нові репліки, що дозволяло уникнути перевантаження окремих подів.

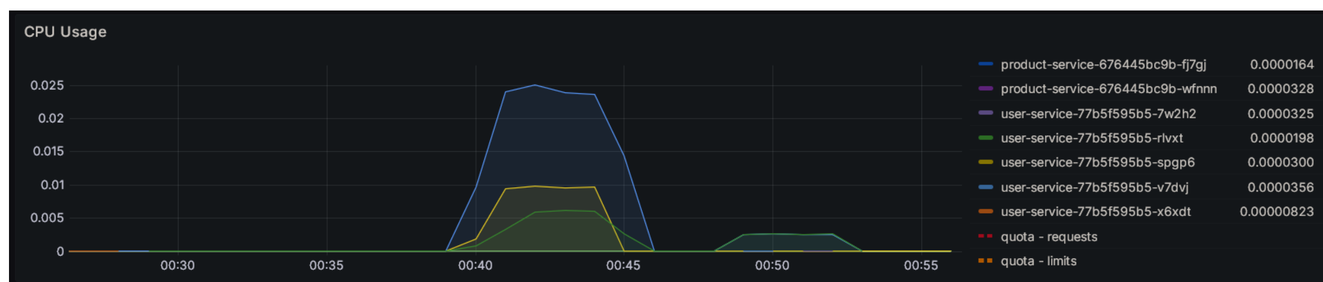


Рисунок 4.4.2 - Навантаження процесору на кластері (рисунок створено самостійно)

Щодо оперативної пам'яті (див. рис. 4.4.3), усі сервіси демонстрували стабільне використання в межах 3.2–3.5 MiB. Завдяки активованому VPA значення залишались сталими без необхідності ручного втручання. У порівнянні з попередніми методами, де пам'ять визначалась статично (особливо у монолітній архітектурі), підхід із VPA дозволив ефективно адаптувати обсяги виділеної пам'яті відповідно до фактичних потреб сервісу, без перевитрати ресурсів.

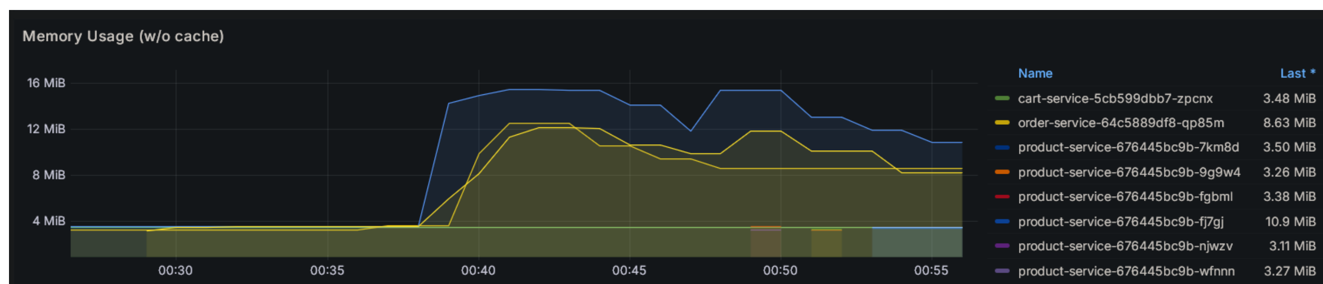


Рисунок 4.4.3 - Навантаження оперативної пам'яті (рисунок створено самостійно)

Мережеве навантаження (див. рис. 4.4.4) на більшості сервісів залишалось на нульовому або близькому до нього рівні, що пояснюється локальним характером тестування без зовнішнього трафіку. Лише у сервісі user-service були зафіксовані незначні вхідні потоки (до 5 B/s). Система з автоскейлингу справлялась із таким навантаженням без втрат або сплесків затримок.

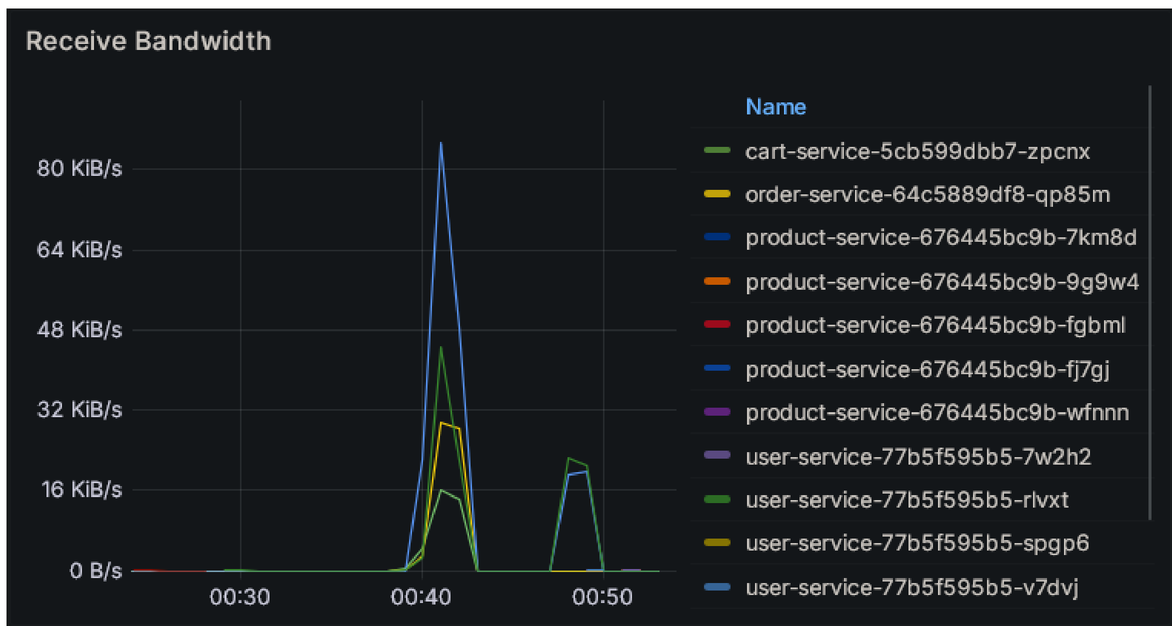


Рисунок 4.4.4 - Пропускна спроможність (рисунок створено самостійно)

Зафіксовано нуль рестартів упродовж усього періоду тестування. Це підтверджує підвищену стабільність мікросервісного підходу з автоскейлінгом.

У ході експерименту кількість подів для кожного мікросервісу зростала від 1 до 4 залежно від навантаження, що підтверджує ефективність горизонтального масштабування через HPA. Крім того, налаштування VPA дозволило кожному поду динамічно змінювати свій ресурсний профіль — CPU та memory limits/request — без перезапуску.

Загальна зважена оцінка мікросервісного підходу становить 0.926 (див. табл. 4.4.1), що є найвищим результатом серед розглянутих варіантів (HPA — 0.853, VPA — 0.746). Це свідчить про високу ефективність застосування мікросервісної архітектури для оптимізації продуктивності та масштабованості хмарного веб-додатку.

Таблиця 4.4.1 – Таблиця оцінки методу (таблиця виконана самостійно)

Критерій	Оцінка ефективності	Ваговий коефіцієнт	Зважена оцінка
Масштабованість (Scalability)	1.00	0.35	0.350
Надійність (Reliability)	0.95	0.25	0.238
Гнучкість підтримки (Maintainability)	0.90	0.15	0.135
Вартість ресурсів (Resource Cost)	0.85	0.15	0.128
Час розгортання (Deployment Time)	0.75	0.10	0.075
Разом			0.926

Мікросервіси отримали максимальну оцінку (1.00) за критерієм масштабованості, оскільки така архітектура дозволяє масштабувати кожен сервіс незалежно, відповідно до його навантаження. Це забезпечує гнучке горизонтальне масштабування, яке є особливо корисним у середовищах з динамічним трафіком.

Також було високо оцінено надійність (0.95): ізольованість компонентів означає, що збій одного сервісу не впливає на решту системи, що покращує стійкість додатку до відмов. Аналогічно, гнучкість підтримки (0.90) є ще однією сильною стороною мікросервісів — можливість незалежно оновлювати чи змінювати сервіси значно полегшує супровід системи.

Вартість ресурсів (0.85) також була оцінена високо, хоча ця архітектура може потребувати додаткових ресурсів на інфраструктуру (мережеві проксі, сервіси зв’язку, логування). Проте, завдяки точному розподілу навантаження, ресурсні витрати залишаються раціонально оптимізованими. Щодо часу розгортання (0.75)

— він нижчий через складність оркестрації кількох сервісів, однак використання CI/CD та інструментів, таких як ArgoCD або Helm, дозволяє зменшити цей недолік.

У підсумку, мікросервісна архітектура є найбільш збалансованим і перспективним методом, що поєднує масштабованість, надійність і підтримуваність, незначно поступаючись у швидкості розгортання та ресурсній економності.

ВИСНОВКИ

Під час написання кваліфікаційної роботи було досліджено методи оптимізації продуктивності хмарних веб-додатків, контейнеризованих у Kubernetes. Основна увага була приділена аналізу та оптимізації монолітної архітектури шляхом переходу до мікросервісної. Це дослідження охоплює повний цикл роботи із системою: від створення початкової версії у вигляді моноліту до її трансформації у сучасну масштабовану систему на основі мікросервісів.

На початковому етапі було розроблено монолітний додаток для інтернет-магазину, що включає основні функції, такі як каталог товарів, управління замовленнями, кошиком і обліковими записами користувачів. Ця архітектура дозволила швидко реалізувати базовий функціонал, але виявила свої недоліки при спробах забезпечити масштабованість і високу продуктивність системи. Аналіз моноліту показав, що тісний взаємозв'язок компонентів ускладнює впровадження змін, тестування та обробку високих навантажень, що робить цю архітектуру менш ефективною для складних і масштабованих проектів.

Наступним етапом дослідження стало впровадження мікросервісної архітектури як одного з найефективніших підходів до оптимізації продуктивності хмарного веб-додатку. У рамках цього переходу функціональність системи було розділено на окремі незалежні сервіси: керування товарами, обробка замовлень, робота з кошиком, автентифікація користувачів тощо. Такий розподіл дозволив кожному сервісу працювати автономно, забезпечивши можливість його незалежного масштабування, розгортання та оновлення, що підвищило гнучкість та надійність системи загалом.

Особливу увагу приділено використанню Kubernetes для оркестрації контейнерів. Ця платформа забезпечила ефективне управління розгортанням сервісів, автоматизацію масштабування та моніторинг роботи системи. Інтеграція з такими інструментами, як Prometheus і Grafana, дозволила створити прозору інфраструктуру для моніторингу та аналізу продуктивності, що сприяло швидкому виявленню та усуненню вузьких місць у системі.

У процесі оптимізації були реалізовані сучасні практики: горизонтальне масштабування сервісів за допомогою механізмів HPA (Horizontal Pod Autoscaler) та динамічне налаштування ресурсів за допомогою VPA (Vertical Pod Autoscaler).

Отримані результати свідчать, що мікросервісна архітектура значно підвищує стабільність і продуктивність системи, дозволяє адаптувати її до зростаючих навантажень і спрощує впровадження нових функцій. Перехід на цю архітектуру також зменшив залежність між компонентами системи, що покращило її стійкість до збоїв. Запропонований підхід може бути використаний у інших масштабних проектах, які потребують високої доступності та продуктивності.

Таким чином, робота стала вагомим внеском у розуміння практичних аспектів побудови масштабованих хмарних систем, демонструючи ефективність сучасних підходів і технологій у розробці веб-додатків [10].

ПЕРЕЛІК ДЖЕРЕЛ ПОСИЛАННЯ

1. Oliynyk O., Mykhnevych D., "Parallelism and concurrency in Golang. comparison with other programming languages (C#, Java, C)", International scientific journal "Internauka" №18 (2021): https://www.inter-nauka.com/issues/2021/18/7780 (дата звернення 15.06.2025)

2. Malyshchenko N., Chernonos M., Oliinyk O., "Goroutines in the concurrent programming", International scientific journal "Internauka" №20 (2020): https://www.inter-nauka.com/issues/2020/20/6726 (дата звернення 15.06.2025)

3. Ворона Д.О. Дослідження методів оптимізації продуктивності хмарних веб-додатків контейнеризованих у Kubernetes. XXIX міжнародний молодіжний форум «Радіоелектроніка та молодь у XXI столітті» м. Харків. 2025. С. 244–246. URL: https://nure.ua/konferencii-ta-workshops/mizhnarodnij-molodizhnij-forum-radioelektronika-i-molod-u-hhi-stolitti/xxix-mizhnarodnyj-molodizhnyj-forum-radioelektronika-i-molod-u-khkhi-stolitti (дата звернення 15.06.2025)

4. Kubernetes and Docker Load Balancing: State-of-the-Art Techniques and Challenges. URL: https://www.researchgate.net/publication/376593267_Kubernetes_and_Docker_Load_Balancing_State-of-the-Art_Techniques_and_Challenges (дата звернення 15.06.2025)

5. Enhanced Visibility for Real-time Monitoring and Alerting in Kubernetes by Integrating Prometheus,Prometheus, Grafana, Loki, and Alerta. URL: https://www.researchgate.net/publication/381347090_Enhanced_Visibility_for_Real-time_Monitoring_and_Alerting_in_Kubernetes_by_Integrating_PrometheusPrometheus_Grafana_Loki_and_Alerta (дата звернення 15.06.2025)

6. Kubernetes in Microservices. URL: https://www.researchgate.net/publication/365344228_Kubernetes_in_Microservices (дата звернення 15.06.2025)

7. Optimizing Container Management with Kubernetes on Linux: Key Strategies and Common Obstacles. URL: https://www.researchgate.net/publication/383467362_Optimizing_Container_Management_with_Kubernetes_on_Linux_Key_Strategies_and_Common_Obstacles (дата звернення 15.06.2025)

8. Navigating the Landscape of Kubernetes Security Threats and Challenges. URL: https://www.researchgate.net/publication/387402650_Navigating_the_Landscape_of_Kubernetes_Security_Threats_and_Challenges (дата звернення 15.06.2025)

9. Containerization in cloud computing: comparing Docker and Kubernetes for scalable web applications. URL: https://www.researchgate.net/publication/385481942_Containerization_in_cloud_computing_comparing_Docker_and_Kubernetes_for_scalable_web_applications (дата звернення 15.06.2025)

10. GitHub репозиторій додатку. URL: https://github.com/dmvorona/shop (дата звернення 15.06.2025)

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

1 Oliynyk O., Mykhnevych D., “Parallelism and concurrency in Golang. comparison with other programming languages (C#, Java, C)”, International scientific journal "Internauka" №18 (2021): https://www.inter-nauka.com/issues/2021/18/7780 (дата звернення 15.06.2025)

2 Malyshchenko N., Chernonos M., Oliinyk O., “Goroutines in the concurrent programming”, International scientific journal "Internauka" №20 (2020): https://www.inter-nauka.com/issues/2020/20/6726 (дата звернення 15.06.2025)