МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ

ЛЬВІВСЬКИЙ НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ ВЕТЕРИНАРНОЇ МЕДИЦИНИ ТА БІОТЕХНОЛОГІЙ ІМЕНІ С.З. ҐЖИЦЬКОГО
ФАКУЛЬТЕТ МЕХАНІКИ, ЕНЕРГЕТИКИ ТА ІНФОРМАЦІЙНИХ ТЕХНОЛОГІЙ
КАФЕДРА ІНФОРМАЦІЙНИХ ТЕХНОЛОГІЙ

КВАЛІФІКАЦІЙНА РОБОТА

другого (магістерського) рівня вищої освіти

на тему:

«ОБҐРУНТУВАННЯ ПІДХОДІВ ДО РОЗГОРТАННЯ ТА МАСШТАБУВАННЯ ВЕБ-ДОДАТКІВ У ХМАРНИХ СЕРЕДОВИЩАХ»

Виконав: студент 6 курсу
спеціальності 126 «Інформаційні системи та технологій»

Недільський М.В,
(прізвище та ініціали)

Керівник: Желєзняк А.М.
(прізвище та ініціали)

ДУБЛЯНИ 2025

МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
ЛЬВІВСЬКИЙ НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ ВЕТЕРИНАРНОЇ МЕДИЦИНИ ТА БІОТЕХНОЛОГІЙ ІМЕНІ С.З. ҐЖИЦЬКОГО
ФАКУЛЬТЕТ МЕХАНІКИ, ЕНЕРГЕТИКИ ТА ІНФОРМАЦІЙНИХ ТЕХНОЛОГІЙ
КАФЕДРА ІНФОРМАЦІЙНИХ ТЕХНОЛОГІЙ

Рівень вищої освіти другий (магістерський)
Спеціальність 126 «Інформаційні системи та технології»

ЗАТВЕРДЖУЮ
Завідувач кафедри

_____
(підпис)
д.т.н., професор, Тригуба А. М.
(вч. звання, прізвище, ініціали)
“____” _____202 року

З А В Д А Н Н Я
НА КВАЛІФІКАЦІЙНУ РОБОТУ

Нєдільський Миколай Васильович
(прізвище, ім’я, по батькові)

1. Тема роботи *«ОБҐРУНТУВАННЯ ПІДХОДІВ ДО РОЗГОРТАННЯ ТА МАСШТАБУВАННЯ ВЕБ-ДОДАТКІВ У ХМАРНИХ СЕРЕДОВИЩАХ»*
керівник роботи *к. е н., доцент., Желєзняк А.М.*
(наук.ступінь, вч. звання, прізвище, ініціали)

затверджені наказом ЛНУВМБТ ім.С.З.Гжицького №140/к-с від 28.02.2025

2. Строк подання студентом роботи 05.12.2025 р.
3. Вихідні дані: *аналітичні дані роботи та характеристика об’єкту дослідження, опис бібліотек мов програмування, науково-технічна і довідкова література.*
4. Зміст кваліфікаційної роботи (перелік питань, які потрібно розробити)
Вступ
1. Аналіз стану питання в теорії та практиці
2. Обґрунтування, вибір та реалізація інструментарію вирішення задачі
3.Практична реалізація системи розгортання та масштабування веб-додатку
4. Охорона праці та безпека в надзвичайних ситуаціях
5. Визначення ефективності
Висновки
Бібліографічний список
5. Перелік графічного матеріалу
Графічний матеріал подається у вигляді презентації

6. Консультанти розділів

Розділ	Прізвище, ініціали та посада консультанта	Підпис, дата		Відмітка про виконання
		завдання видав	завдання прийняв	
1, 2, 3, 5	*Желєзняк А.М., к.е.н., доцент кафедри інформаційних технологій*			
4	*Городецький І.М., к.т.н., доцент кафедри інженерної механіки*			

7. Дата видачі завдання 01.03.2025 р.

КАЛЕНДАРНИЙ ПЛАН

№ з/п	Назва етапів дипломної роботи	Строк виконання етапів роботи	Відмітка про виконання
1	*Отримання завдання. Вивчення літератури по темі роботи. Написання першого розділу*	*01.03.2025 - 31.05.2025*	
2	*Проектування та опис технічного завдання, обґрунтування та вибір інструментарію реалізації проекту (написання другого розділу).*	*01.06.2025 - 31.08.2025*	
3	*Програмна реалізація поставленого завдання (написання третього розділу)*	*01.09.2025 - 17.10.2025*	
4	*Написання розділу з охорони праці*	*18.10.2025 - 04.11.2025*	
5	*Оцінка ефективності поставленого завдання (виконання п'ятого розділу)*	*05.11.2025 - 18.11.2025*	
6	*Завершення оформлення основної частини, написання висновків та підготовка презентаційного матеріалу*	*19.11.2025 - 01.12.2025*	
7	*Завершення роботи в цілому. Підготовка до захисту кваліфікаційної роботи*	*02.12.2025 - 08.12.2025*	

Студент _____ *Недільський М.В.*
(підпис) (прізвище та ініціали)

Керівник роботи _____ *Желєзняк А.М.*
(підпис) (прізвище та ініціали)

УДК 004.738.1:004.7

Обґрунтування підходів до розгортання та масштабування веб-додатків у хмарних середовищах. Недільський Миколай Васильович. Кваліфікаційна робота. Кафедра інформаційних технологій. Дубляни, Львівський національний університет ветеринарної медицини та біотехнологій ім. С. З. Гжицького, 2025 рік.

Кваліфікаційна робота: 73 сторінки текстової частини, 9 таблиць, 25 рисунків, 31 джерел літератури, 1 додаток.

У роботі подано характеристику сучасних підходів до побудови масштабованих веб-додатків із використанням технологій хмарних обчислень, контейнеризації та оркестрації. Проаналізовано предметну область, наведено огляд існуючих платформ для розгортання та масштабування, а також визначено їх ключові можливості щодо автоматизації інфраструктури та забезпечення високої доступності веб-сервісів.

Здійснено аналіз архітектурних моделей cloud-native застосунків та обґрунтовано вибір технологій Docker і Kubernetes для розробки та масштабування веб-додатку. Реалізовано контейнеризацію веб-додатку, розгорнуто інфраструктуру у середовищі Kubernetes.

Запропоновано практичні підходи до підвищення ефективності роботи веб-додатку в умовах змінного навантаження, проаналізовано результати тестування, встановлено вплив масштабування на швидкість обробки запитів та визначено переваги використання контейнеризованих рішень при побудові масштабованих сервісів.

Здійснено аналіз травматичних ситуацій при роботі з комп’ютерною технікою та викладено ключові питання охорони праці й безпеки життєдіяльності в умовах надзвичайних ситуацій, зокрема під час повітряних тривог.

Ключові слова: веб-додаток, контейнеризація, масштабування, Kubernetes.

ЗМІСТ

ВСТУП

Стрімкий розвиток інформаційних технологій, цифровізація бізнесу та зростання кількості користувачів веб-ресурсів зумовлюють підвищені вимоги до продуктивності, надійності та доступності веб-додатків. У сучасних умовах, коли онлайн-сервіси обробляють мільйони запитів щосекунди, здатність системи масштабуватися відповідно до навантаження стає ключовим чинником її успішного функціонування. Навіть короткочасні затримки або збої можуть спричинити втрату користувачів, зниження доходів компаній та погіршення репутації сервісу. Саме тому питання ефективного розгортання та масштабування веб-додатків набувають особливої актуальності.

Серед основних тенденцій сучасної веб-індустрії важливе місце посідають технології контейнеризації та оркестрації, які дозволяють автоматизувати керування застосунками, забезпечити їхню відмовостійкість і гнучке реагування на зміну навантаження. Docker та Kubernetes стали стандартом де-факто у проєктуванні масштабованих веб-систем, а хмарні платформи, такі як Amazon Web Services, Microsoft Azure та Google Cloud Platform, пропонують розвинуті інструменти для підтримки масштабування в режимі реального часу. Вивчення цих технологій та розробка практичного рішення на їх основі є важливими кроками до формування навичок створення сучасних високонавантажених веб-додатків.

Метою кваліфікаційної роботи є проектування та розробка системи розгортання та масштабування веб-додатку у хмарному середовищі з використанням технологій контейнеризації та оркестрації. У роботі досліджено теоретичні основи побудови масштабованих веб-сервісів, виконано порівняльний аналіз можливостей AWS, Azure та Google Cloud у контексті масштабування, а також розгорнуто практичну інфраструктуру на основі Docker і Kubernetes.

Тема кваліфікаційної роботи є актуальною, оскільки сучасні

веб-додатки працюють в умовах високої конкуренції та непередбачуваних навантажень, що вимагає ефективних рішень для забезпечення стабільності й надійності. У реальних умовах саме можливість швидко масштабувати інфраструктуру, адаптуючись до трафіку, визначає рівень якості сервісу та задоволення користувачів. Технології, що лежать в основі cloud-native підходу, є важливими інструментами для бізнесу і залишаються ключовими напрямками розвитку у сфері веб-інженерії.

У ході виконання кваліфікаційної роботи були поставлені такі завдання: дослідити теоретичні та методологічні аспекти побудови масштабованих веб-систем; проаналізувати існуючі хмарні платформи та їхні інструменти масштабування; визначити вимоги до системи розгортання веб-додатків; здійснити контейнеризацію застосунку та розгортання у Kubernetes; реалізувати механізми горизонтального та автоматичного масштабування; провести тестування продуктивності та оцінити ефективність розробленого рішення; виконати аналіз охорони праці та безпеки під час роботи з комп’ютерною технікою.

Загалом кваліфікаційна робота складається з п’яти розділів. У першому розділі розкрито теоретичні основи масштабування веб-додатків, принципи роботи контейнеризації та можливості хмарних платформ. У другому розділі наведено обґрунтування вибору інструментів та засобів для реалізації системи. Третій розділ присвячено розробці веб-додатку, його контейнеризації, розгортанню у Kubernetes та дослідженню механізмів масштабування. У четвертому розділі розглянуто питання охорони праці та безпеки в надзвичайних ситуаціях. П’ятий розділ містить оцінку ефективності впровадженого рішення та аналіз результатів експериментального дослідження.

Наукова новизна кваліфікаційної роботи полягає у практичному застосуванні комплексного підходу до побудови системи автоматичного масштабування веб-додатку з використанням Kubernetes у локальному cloud-native середовищі. Реалізована інфраструктура демонструє можливість

створення масштабованої системи без використання платних хмарних сервісів, забезпечуючи при цьому повну відповідність принципам сучасних хмарних архітектур.

Під час виконання кваліфікаційної роботи результати дослідження були апробовані та представлені у вигляді тез на VII Всеукраїнській науково-практичній конференції “Інформаційна безпека та інформаційні технології” (ІБІТ - 2025). Тема доповіді: «Підходи до масштабування веб-додатків у хмарних середовищах AWS, Azure та Google Cloud». Публікація тез у збірнику матеріалів конференції підтверджує практичну значущість та актуальність обраної тематики.

РОЗДІЛ 1.
АНАЛІЗ СТАНУ ПИТАННЯ В ТЕОРІЇ ТА ПРАКТИЦІ

1.1 Теоретичні основи хмарних обчислень та масштабування веб-додатків

У сучасних умовах розвитку інформаційних технологій хмарні обчислення стали базовою інфраструктурою для значної частини веб-додатків. Перехід від локальних серверних рішень до хмарних платформ пов’язаний насамперед із потребою забезпечити гнучкість, масштабованість та економічну доцільність експлуатації інформаційних систем. Традиційні підходи, за яких організація утримує власний серверний парк, стикаються з низкою обмежень: високою вартістю придбання та обслуговування обладнання, складністю модернізації, тривалими циклами розгортання нових сервісів та обмеженою можливістю швидко реагувати на зміни навантаження. Хмарні обчислення, у свою чергу, дозволяють абстрагуватися від фізичної інфраструктури й зосередитися на логіці веб-додатків та їхньому розвитку.

Під хмарними обчисленнями зазвичай розуміють модель надання обчислювальних ресурсів як сервісу, доступного через мережу Інтернет. Користувач отримує можливість створювати, конфігурувати та масштабувати віртуальні машини, сховища даних, мережеві компоненти та прикладні сервіси, сплачуючи лише за фактичне використання. Важливою рисою такої моделі є те, що вона забезпечує еластичність: ресурси можуть бути динамічно збільшені або зменшені залежно від поточного навантаження на систему. Це особливо актуально для веб-додатків, для яких характерні пікові періоди активності користувачів та нерівномірний розподіл запитів у часі.

Ще однією принциповою властивістю хмарних обчислень є пулінг ресурсів. Фізичні сервери, дискові системи та мережеве обладнання об’єднуються у віртуалізоване середовище, з якого користувачі отримують логічно ізольовані ресурси. Завдяки цьому досягається економія масштабу та висока ефективність використання обладнання. Паралельно з цим реалізується механізм вимірюваності сервісів: постачальник хмарних послуг

фіксує обсяг використаних ресурсів (процесорний час, обсяг пам’яті, трафік, дисковий простір), що дозволяє будувати прозорі моделі оплати та контролювати витрати.

З погляду організації взаємодії між постачальником хмарних послуг та користувачем виділяють три основні моделі надання сервісів: інфраструктура як послуга (Infrastructure as a Service, IaaS), платформа як послуга (Platform as a Service, PaaS) та програмне забезпечення як послуга (Software as a Service, SaaS). У моделі IaaS користувач отримує у розпорядження віртуальні сервери, мережеву інфраструктуру та сховища і самостійно відповідає за встановлення операційних систем, проміжного ПЗ та прикладних компонентів. Модель PaaS передбачає надання вже підготовленого середовища для розробки та розгортання додатків: користувач працює з платформою (наприклад, веб-сервером, базою даних, середовищем виконання), нехтуючи деталями низькорівневої конфігурації інфраструктури. Нарешті, у моделі SaaS користувач отримує доступ до готового програмного продукту через веб-інтерфейс або клієнтський застосунок, зовсім не взаємодіючи з інфраструктурними аспектами.

Структурно взаємозв’язок між цими трьома моделями можна подати як ієрархію рівнів, де кожен наступний рівень абстрагує користувача від більшої кількості технічних деталей. На рисунку 1.1 наведено типове графічне представлення моделей IaaS, PaaS та SaaS, яке демонструє, які саме компоненти інфраструктури залишаються під контролем постачальника, а які - користувача.

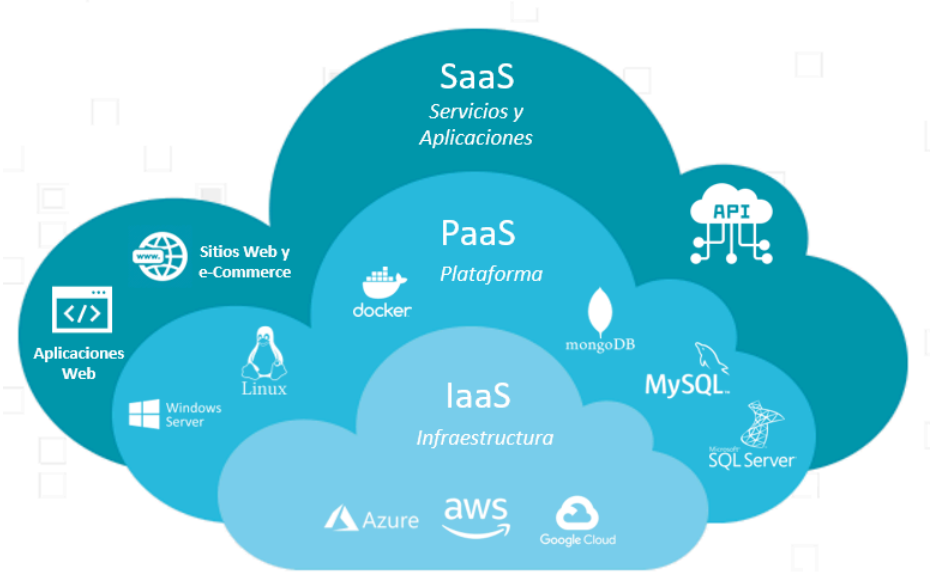

Рисунок 1.1 - Моделі надання хмарних послуг IaaS, PaaS та SaaS [1]

Для теми даної кваліфікаційної роботи ці моделі мають принципове значення, оскільки вибір того чи іншого рівня сервісу визначає підхід до розгортання та масштабування веб-додатку. У моделі IaaS основна увага приділяється керуванню віртуальними машинами та мережевою топологією; у PaaS - побудові ефективної архітектури застосунку в межах наданої платформи; у SaaS - оптимізації логіки сервісу та досвіду користувача, тоді як масштабування часто виконується автоматично силами провайдера.

Таким чином, теоретичні основи хмарних обчислень забезпечують понятійний апарат для подальшого аналізу підходів до масштабування веб-додатків та обґрунтування вибору конкретних інструментів і архітектурних рішень у межах цієї роботи.

Сучасні дослідження у сфері масштабованих веб-архітектур приділяють значну увагу ефективності обробки даних у розподілених системах та впровадженню мікросервісних підходів. Багато авторів підкреслюють, що традиційні монолітні системи мають суттєві обмеження при обробці пікових навантажень, що зумовлює необхідність переходу до контейнерних технологій. Згідно з аналітичними звітами Gartner та доповідями Google Research, використання Kubernetes дозволяє зменшити

витрати ресурсів та значно підвищити стійкість програмних комплексів.

Наукові праці останніх років також підтверджують перспективність поєднання хмарних сервісів із технологіями DevOps, що забезпечує безперервну інтеграцію та доставку застосунків. У контексті масштабування велике значення має автоматичне балансування навантаження, яке дозволяє мінімізувати затримки відповіді та запобігти перевантаженню окремих вузлів системи. Ці принципи закладають основу для високодоступних веб-сервісів, що функціонують у глобальних хмарних інфраструктурах. [2]

Поява хмарних обчислень стала результатом тривалої еволюції інформаційних технологій. Перші концепції віддаленого доступу до обчислювальних ресурсів виникли ще у 60-х роках XX століття у рамках парадигми time-sharing, коли декілька користувачів могли одночасно працювати з одним потужним комп’ютером. Подальший розвиток мережевих технологій та поява інтернету сприяли формуванню ідей розподілених систем, що лягло в основу сучасних хмарних платформ.

У 2006 році компанія Amazon представила Amazon Web Services - першу комерційну платформу масового використання, яка дозволила орендувати обчислювальні ресурси у форматі IaaS. Пізніше на ринок вийшли Microsoft Azure та Google Cloud Platform, що дало початок новому етапу конкуренції між великими провайдерами. Упродовж 2010-2020 років хмарні технології стали ключовим елементом цифрової трансформації бізнесу, забезпечивши підприємствам гнучкість, масштабованість і високу доступність критично важливих сервісів.

Сучасний етап розвитку характеризується переходом до контейнеризації, мікросервісної архітектури та широким застосуванням платформ оркестрації, таких як Kubernetes. Завдяки цьому хмарні середовища стали основою для високонавантажених розподілених систем, включно з веб-додатками, що вимагають швидкої адаптації до змінних умов роботи.

1.2 Типи масштабування веб-додатків та архітектурні підходи

Проблема масштабування веб-додатків безпосередньо пов’язана з необхідністю підтримувати задані показники продуктивності та доступності за зміни інтенсивності навантаження. Якщо кількість користувачів або обсяг оброблюваних даних зростає, система повинна адаптуватися, не втрачаючи стабільності та не погіршуючи користувацький досвід. У загальному випадку розрізняють два базові підходи до масштабування: вертикальне та горизонтальне.

Вертикальне масштабування полягає у збільшенні ресурсів окремого обчислювального вузла. Це може бути нарощування обсягу оперативної пам’яті, використання більш потужного процесора, прискорення підсистеми зберігання даних тощо. Такий шлях є інтуїтивно зрозумілим і часто може бути реалізований без суттєвої зміни архітектури програмного забезпечення. У невеликих системах або на ранніх етапах розвитку веб-додатку вертикальне масштабування дозволяє відносно просто вирішити проблеми недостатньої продуктивності. Однак цей підхід має природні обмеження: кожен фізичний сервер або віртуальна машина має граничну конфігурацію, після досягнення якої подальше розширення неможливе або економічно недоцільне. Крім того, концентрація навантаження на одному вузлі знижує відмовостійкість системи: вихід з ладу такого вузла призводить до простою всього сервісу.

Горизонтальне масштабування, навпаки, передбачає додавання нових вузлів до системи. Замість одного «потужного» сервера створюється група (кластер) менш потужних, але взаємопов’язаних інстансів, між якими розподіляються користувацькі запити. Такий підхід краще узгоджується з філософією хмарних обчислень, де ресурси надаються у вигляді великої кількості уніфікованих блоків. Горизонтальне масштабування дозволяє практично безмежно нарощувати пропускну здатність системи, а також підвищує її стійкість до відмов: у разі виходу з ладу одного з вузлів інші продовжують обробляти запити.

Візуальне порівняння цих двох підходів наведено на рисунку 1.2. На схемі вертикальне масштабування представлено як нарощування ресурсів одного сервера, тоді як горизонтальне - як додавання кількох однакових серверів, між якими виконується балансування навантаження.

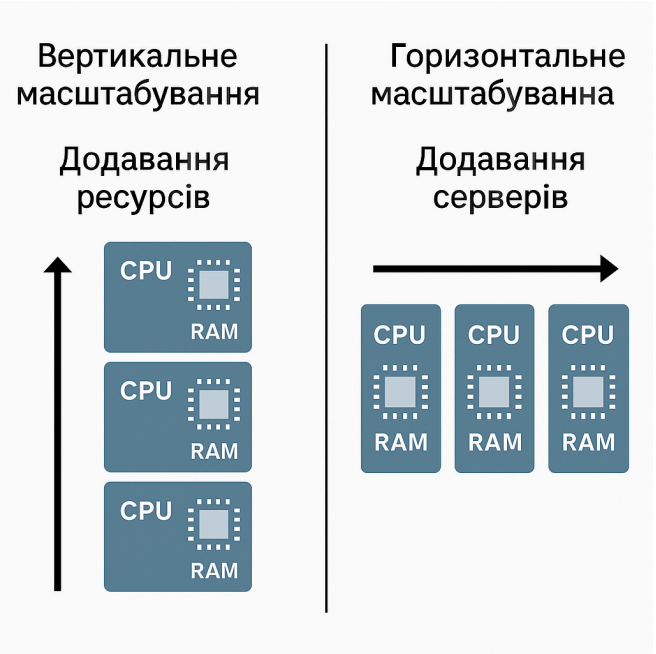

Рисунок 1.2 - Порівняння вертикального та горизонтального масштабування веб-додатків [3]

У контексті хмарних веб-додатків горизонтальне масштабування, як правило, реалізується за допомогою механізмів автоматичного масштабування (autoscaling). Суть такого механізму полягає в тому, що платформа на основі заданих правил і моніторингу метрик (завантаження процесора, кількості активних підключень, довжини черг запитів тощо) автоматично змінює кількість запущених екземплярів сервісу. Наприклад, за перевищення порогового значення завантаження CPU система додає нові інстанси, а за зменшення навантаження - зупиняє зайві, що дозволяє оптимізувати витрати.

Для ефективної реалізації горизонтального масштабування необхідні відповідні архітектурні підходи. Одним із ключових є мікросервісна архітектура, за якої веб-додаток будується як сукупність відносно незалежних

сервісів, кожен з яких реалізує чітко визначену бізнес-функцію. Мікросервіси можуть масштабуватися окремо один від одного, що підвищує гнучкість системи: наприклад, сервіс авторизації користувачів може мати інші вимоги до масштабованості, ніж сервіс генерації звітів.

Ще одним важливим елементом є контейнеризація. Використання контейнерів (зокрема, Docker) дозволяє упакувати додаток разом із усіма залежностями в стандартизований образ, який може бути запущений у будь-якому середовищі, що підтримує контейнеризацію. Це спрощує розгортання та масштабування, оскільки всі екземпляри сервісу є ідентичними з погляду конфігурації. Поверх контейнеризації часто застосовують системи оркестрації, такі як Kubernetes, які автоматизують розгортання, оновлення, балансування навантаження та відновлення контейнерів у разі збоїв.

Окремо слід згадати безсерверні (serverless) архітектури, які фактично реалізують максимально тонкий рівень масштабування - на рівні окремих функцій. У таких системах розробник описує логіку обробки подій, а питання виділення та масштабування ресурсів повністю делегуються хмарній платформі. Це дозволяє досягти високої еластичності, особливо для подійно орієнтованих навантажень із нерівномірним профілем.

Таким чином, вибір між вертикальним та горизонтальним масштабуванням, а також конкретних архітектурних підходів, визначається вимогами до продуктивності, відмовостійкості та вартості експлуатації веб-додатку. У рамках даної кваліфікаційної роботи основна увага приділятиметься саме горизонтальному масштабуванню та пов’язаним із ним хмарним технологіям.

Сучасний ринок хмарних технологій представлений великою кількістю сервісів, однак найбільшого поширення набули три ключові платформи: Amazon Web Services (AWS), Microsoft Azure та Google Cloud Platform (GCP). Вони забезпечують широкий спектр інструментів для розгортання веб-додатків, організації горизонтального та вертикального масштабування,

моніторингу та автоматизації інфраструктури.

AWS є лідером ринку та пропонує найбільшу кількість сервісів для створення масштабованих систем. Azure демонструє високу інтеграцію з продуктами Microsoft, що робить його популярним у корпоративному секторі. GCP виділяється оптимізацією під високонавантажені обчислення, високою продуктивністю мережі та орієнтацією на інноваційні сервіси машинного навчання.

У зв’язку з широким розповсюдженням хмарних технологій постає потреба не лише в огляді окремих сервісів, але й у їх порівнянні за ключовими характеристиками. Для вибору оптимальної платформи важливо враховувати такі параметри, як продуктивність, простота інтеграції, масштабованість, доступність інструментів автоматизації та вартість використання. Оскільки архітектури веб-додатків можуть суттєво різнитися за вимогами, порівняльний аналіз дозволяє обґрунтовано визначити, яка хмарна платформа найкраще відповідає конкретним технічним і бізнес-потребам. З цією метою далі наведено узагальнену таблицю, що демонструє ключові особливості AWS, Azure та Google Cloud у контексті масштабування веб-додатків.

Таблиця 1.1 - Порівняльна таблиця популярних хмарних сервісів

Характеристика	AWS	Azure	GCP
Рік запуску	2006	2010	2011
Основний фокус	Масштабовані інфраструктури	Корпоративна інтеграція	Big Data, ML
Контейнеризація	EKS	AKS	GKE
Переваги	Найбільший ринок, стабільність	Глибока інтеграція з Microsoft	Висока продуктивність, простота
Недоліки	Найдорожчий	Складність налаштування	Менше сервісів

Проведений порівняльний аналіз свідчить, що кожна з представлених платформ має власні сильні сторони та специфічні сфери застосування. AWS забезпечує найширші можливості масштабування та вважається найстабільнішою інфраструктурою для високонавантажених систем. Azure є найбільш придатним вибором для компаній, які використовують екосистему Microsoft і потребують глибокої інтеграції з корпоративними сервісами. GCP, у свою чергу, вирізняється високою швидкодією мережевої інфраструктури та інструментами обробки великих даних, що робить його перспективним для проєктів, орієнтованих на машинне навчання та аналітику. Таким чином, результати порівняння підкреслюють, що вибір хмарної платформи має здійснюватися з урахуванням конкретних вимог веб-додатку, особливостей його архітектури та очікуваних сценаріїв навантаження.

1.3. Огляд сучасних платформ для розгортання та масштабування веб-додатків

Побудова масштабованого веб-додатку в хмарному середовищі неможлива без залучення спеціалізованих платформ, які надають готові інструменти для розгортання, оркестрації та моніторингу сервісів.[4] На ринку хмарних технологій домінують три провідні постачальники: Amazon Web Services (AWS), Microsoft Azure та Google Cloud Platform (GCP). Кожен із них пропонує широкий набір сервісів, орієнтованих на підтримку веб-додатків різного масштабу - від невеликих сайтів до розподілених високонавантажених систем.

Amazon Web Services є однією з найстаріших та наймасштабніших хмарних платформ.[5] Вона надає сервіси для обчислень (Amazon EC2), балансування навантаження (Elastic Load Balancing), автоматичного масштабування (Auto Scaling Groups), контейнеризації (Amazon ECS, Amazon EKS) та безсерверних обчислень (AWS Lambda).[6] Для розробника веб-додатків це означає можливість побудувати як класичну багатошарову архітектуру на базі віртуальних машин, так і сучасну мікросервісну систему в

контейнерах або функціонально орієнтований додаток на базі Lambda-функцій.

Microsoft Azure, своєю чергою, активно орієнтується на інтеграцію з корпоративною екосистемою Microsoft і пропонує потужні засоби побудови гібридних рішень.[7] Azure Virtual Machines та Virtual Machine Scale Sets дозволяють створювати масштабовані кластери віртуальних машин, Azure Kubernetes Service (AKS) забезпечує кероване середовище для запуску контейнеризованих додатків, а Azure Functions реалізують безсерверний підхід.[8] Додатково Azure має розвинені сервіси для побудови мережевої інфраструктури, захисту доступу та аналітики.

Google Cloud Platform відомий особливо сильними рішеннями у сфері контейнеризації та обробки великих даних.[9] Google Kubernetes Engine (GKE) є одним із найпопулярніших керованих сервісів Kubernetes, що забезпечує гнучке й масштабоване середовище для розгортання мікросервісних веб-додатків.[10] Сервіс Cloud Run дозволяє запускати контейнеризовані додатки у безсерверному режимі, автоматично масштабуючи їх залежно від кількості запитів. Окрім цього, GCP пропонує інструменти для побудови розподілених сховищ, черг повідомлень та аналітичних систем.

Спільним для всіх трьох платформ є використання схожих концепцій оркестрації та масштабування. Фактичним «стандартом» у цій галузі став Kubernetes, який забезпечує узагальнений підхід до керування контейнеризованими сервісами. Архітектура Kubernetes включає керівну площину (control plane), що відповідає за прийняття рішень щодо розгортання й масштабування, та робочі вузли (worker nodes), на яких безпосередньо виконуються контейнери. На рисунку 1.3 наведено узагальнену схему компонентів середовища Kubernetes.

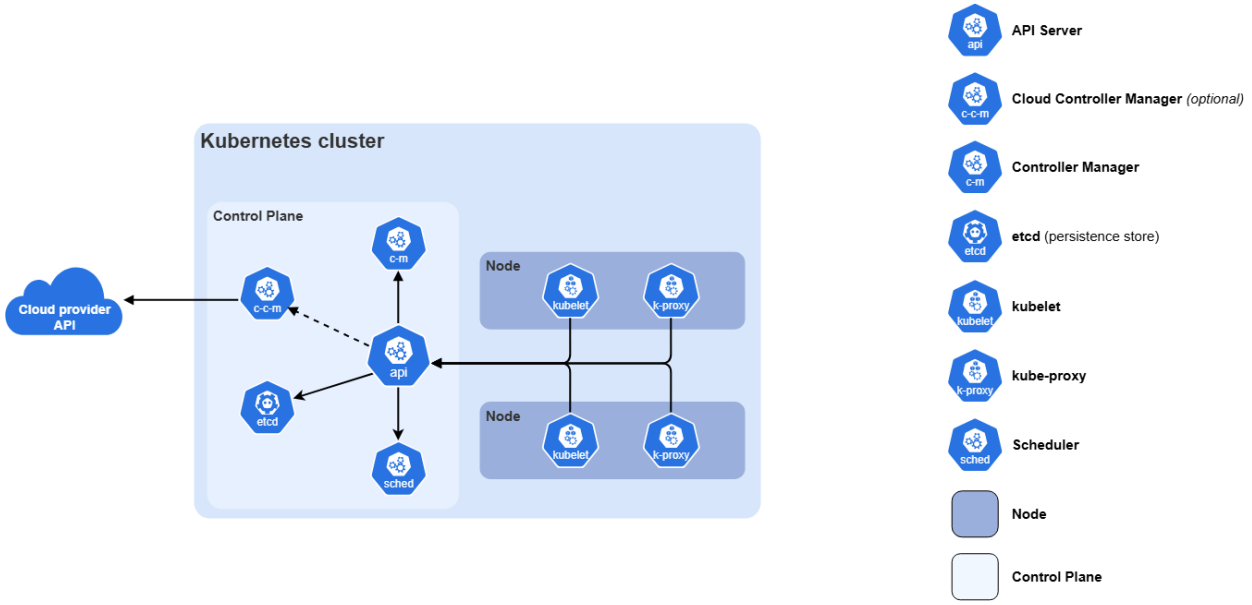

Рисунок 1.3 - Компоненти архітектури Kubernetes (за матеріалами офіційної документації Kubernetes) [11]

Схема ілюструє, як взаємодіють між собою API-сервер, планувальник, контролери, тощо, з одного боку, та вузли, на яких запускаються контейнери, - з іншого. Для побудови масштабованого веб-додатку це означає, що розробник може описати бажаний стан системи (кількість реплік сервісу, вимоги до ресурсів, політики масштабування), а Kubernetes забезпечить його підтримання, автоматично додаючи або видаляючи контейнери, перерозподіляючи навантаження та відновлюючи сервіси після збоїв.

Окрім загальної характеристика хмарних платформ, доцільно розглянути їх порівняння з позиції можливостей масштабування та інструментів оркестрації. Amazon Web Services, Microsoft Azure та Google Cloud Platform пропонують схожі технологічні рішення, однак відрізняються у підходах до ціноутворення, продуктивності та гнучкості налаштувань.

AWS залишається найбільш зрілою та функціонально повною платформою, пропонуючи Auto Scaling Groups, Elastic Load Balancing та розвинуту екосистему обчислювальних сервісів. Azure тісно інтегрований з сервісами Microsoft (Active Directory, Windows Server, SQL Server), що робить його зручним для організацій, які вже використовують корпоративну екосистему компанії. Google Cloud Platform вирізняється оптимізованою

роботою з контейнерами, оскільки саме Google є розробником Kubernetes, що робить GCP однією з найефективніших платформ для контейнерних оркестраційних систем. [12]

Порівняння платформ наведено у таблиці 1.2, що дозволяє обґрунтовано оцінити їх відмінності у контексті задач масштабування та оркестрації контейнерів.

Таблиця 1.2 - Порівняння хмарних платформ за можливостями масштабування

Платформа	Механізми масштабування	Kubernetes-підтримка	Продуктивність	Особливості
AWS	Auto Scaling Groups, ELB	Amazon EKS	Висока	Найбільша кількість сервісів, стабільність
Azure	VM Scale Sets, Load Balancer	Azure AKS	Висока	Ідеальна інтеграція з Microsoft-еко системою
GCP	Instance Groups, Cloud Load Balancing	Google GKE	Дуже висока	Лідер у Kubernetes-ін фраструктурі

Вибір конкретної хмарної платформи для розгортання веб-додатку залежить від багатьох факторів: технічних вимог, бюджетних обмежень, наявної експертизи в команді, вимог до інтеграції з іншими системами тощо. Водночас, незалежно від провайдера, ключовими залишаються принципи: використання розподіленої архітектури, контейнеризації, оркестрації та автоматичного масштабування. Саме на основі цих принципів у подальших розділах роботи буде обґрунтовано вибір конкретних інструментів і побудовано практичну модель розгортання та масштабування веб-додатку у хмарному середовищі.

Масштабування веб-додатків у хмарних середовищах є ключовим напрямом розвитку сучасних розподілених систем. Відмінності між можливостями AWS, Azure та GCP формують підґрунтя для вибору оптимальної інфраструктури залежно від бізнес-вимог, передбачуваності навантаження та бюджету. У наукових дослідженнях наголошується, що використання контейнерних оркестраційних систем, таких як Kubernetes, забезпечує максимальну гнучкість, оскільки абстрагує застосунок від конкретної платформи та дозволяє переносити інфраструктуру між хмарами без зміни архітектури.

Проведений аналіз підтверджує, що сучасні підходи до масштабування базуються на поєднанні контейнеризації, автоматизованого розподілу навантаження та моніторингу продуктивності в реальному часі. Ці принципи формують основи cloud-native архітектур, які відповідають вимогам до високої доступності та продуктивності веб-додатків у масштабованих середовищах.

Окрім традиційних методів масштабування, у сучасних хмарних середовищах важливу роль відіграють спеціалізовані механізми, що забезпечують автоматичну адаптацію системи до зміни навантаження. Найбільш поширеними підходами є Horizontal Pod Autoscaler (HPA), Vertical Pod Autoscaler (VPA) та Cluster Autoscaler.

HPA базується на масштабуванні кількості pod-ів залежно від метрик використання ресурсів, таких як завантаження процесора або пам’яті. Це дозволяє збільшувати або зменшувати кількість реплік веб-додатку у відповідь на зміни навантаження. VPA, навпаки, коригує ресурси окремих pod-ів, автоматично аналізуючи потреби у CPU та RAM. Cluster Autoscaler масштабує кількість вузлів (node-ів) у кластері, забезпечуючи необхідний обсяг фізичних ресурсів для роботи контейнеризованих сервісів.

Архітектура сучасних веб-додатків у хмарних середовищах базується на поєднанні контейнеризації, мікросервісної структури та інструментів оркестрації. Типова інфраструктура включає декілька логічних шарів: рівень

обробки запитів, шар бізнес-логіки, модулі обробки статичних ресурсів, систему зберігання даних та інструменти моніторингу. Механізми автоматичного масштабування дозволяють забезпечити стабільну роботу додатку навіть під час різких піків навантаження, автоматично адаптуючи кількість доступних ресурсів до поточних потреб.

РОЗДІЛ 2.
ОБҐРУНТУВАННЯ, ВИБІР ТА РЕАЛІЗАЦІЯ ІНСТРУМЕНТАРІЮ ВИРІШЕННЯ ЗАДАЧІ

2.1 Аналіз вимог до системи та обґрунтування потреби у використанні хмарних сервісів

Першим етапом побудови масштабованого веб-додатку є визначення вимог до системи, оскільки саме вони формують подальші рішення щодо вибору інструментів, платформ та архітектурних підходів. Для сучасних веб-додатків ключовими є вимоги до доступності, стійкості до навантажень, відмовостійкості та можливості динамічного масштабування. Зростання кількості користувачів, сезонні піки активності, необхідність швидкого розгортання нових компонентів або модулів - усе це висуває підвищені вимоги до інфраструктури.

Традиційні серверні рішення не забезпечують достатньої гнучкості, оскільки масштабування в них потребує придбання та встановлення додаткового обладнання, що займає значний час і вимагає окремого адміністрування. Хмарні платформи дозволяють усунути ці недоліки шляхом надання обчислювальних ресурсів за моделями IaaS, PaaS або FaaS. Завдяки цьому архітектура веб-додатку стає адаптивною: кількість інстансів може збільшуватись автоматично при підвищенні навантаження, а при зниженні - відповідно зменшуватись, що оптимізує витрати.

Для обґрунтування вибору інструментарію необхідно оцінити функціональні та нефункціональні вимоги, які визначають формат хмарного розгортання. Функціональні вимоги описують необхідні можливості системи (наприклад, робота контейнеризованих сервісів, можливість автоматичного масштабування, підтримка мікросервісної архітектури). Нефункціональні вимоги визначають характеристики продуктивності, рівнів доступності, безпеки та надійності.

Для формалізації таких вимог сформовано узагальнену специфікацію,

наведено нижче у таблиці 2.1. На відміну від прикладу, вона створена відповідно до тематики розгортання і масштабування веб-додатків у хмарному середовищі.

Таблиця 2.1 - Специфікація вимог до розгортання та масштабування веб-додатку у хмарному середовищі

№	Назва вимоги	Пріоритет	Складність	Контакт
1	Підтримка автоматичного горизонтального масштабування	Обов’язкове	Середня	Розробник, DevOps-інженер
2	Можливість контейнеризації сервісів	Обов’язкове	Низька	Розробник
3	Підтримка оркестрації	Бажане	Висока	DevOps-інженер
4	Наявність засобів моніторингу та логування (CloudWatch/Monitor/Stackdriver)	Обов’язкове	Середня	DevOps-інженер
5	Підтримка балансування навантаження	Обов’язкове	Середня	DevOps-інженер
6	Забезпечення високої доступності (HA) та відмовостійкості	Обов’язкове	Висока	Архітектор
7	Можливість використання Infrastructure as Code (Terraform)	Бажане	Середня	DevOps-інженер
8	Інтеграція з CI/CD (GitHub Actions / GitLab CI / Azure DevOps)	Бажане	Середня	Розробник
9	Можливість масштабування бази даних	Обов’язкове	Висока	Архітектор
10	Підтримка георозподіленості	За потреби	Висока	Архітектор

Як видно з таблиці, значна частина вимог стосується забезпечення еластичності та гнучкості роботи системи. Це зумовлює необхідність використання хмарних платформ, які мають відповідні інструменти для масштабування, управління ресурсами, оркестрації контейнерів та моніторингу. У наступних підрозділах буде здійснено порівняння можливих платформ та дано обґрунтування вибору конкретного інструментарію для реалізації поставлених цілей.

Хмарні провайдери пропонують різноманітні інструменти для масштабування застосунків, що відрізняються рівнем автоматизації, вартістю та доступністю. Найбільш розгалужені механізми масштабування реалізовані у трьох платформах: Amazon Web Services, Microsoft Azure та Google Cloud Platform. Усі вони підтримують контейнеризацію, оркестрацію, автоматичне розподілення навантаження та горизонтальне масштабування залежно від навантаження.

Найбільш близькою до стандарту де-факто є інтеграція інструментів масштабування з Kubernetes. AWS пропонує EKS з autoscaling groups та кластерними авто-масштабувальниками; Azure це AKS із вбудованими механізмами автоскейлінгу; GCP це GKE, що вважається найбільш оптимізованим рішенням для масштабування контейнеризованих веб-додатків. Завдяки цим можливостям хмарні середовища забезпечують еластичність інфраструктури та дозволяють застосункам ефективно обробляти пікові навантаження.

Окрім функціональних можливостей веб-додатку, у процесі розробки та розгортання важливу роль відіграють нефункціональні вимоги. Саме вони визначають якість роботи системи та її здатність до ефективного масштабування в умовах реального навантаження.

Однією з ключових вимог є висока доступність (High Availability). Сучасні користувачі очікують безперервного доступу до сервісів, тому хмарні платформи передбачають механізми резервування, розподілу навантаження та

автоматичного відновлення після збоїв. Для веб-додатків, які працюють у контейнеризованому середовищі, ці вимоги реалізуються через ReplicaSet, pod disruption budgets, а також архітектуру, що охоплює декілька вузлів.

Не менш важливими є вимоги до безпеки, які включають контроль доступу (IAM), безпечне зберігання секретів, шифрування трафіку та аудит подій. Масштабовані веб-додатки мають забезпечити захист даних як на рівні аплікації, так і на рівні інфраструктури.

До критично важливих вимог належать також продуктивність, стабільність часу відповіді та спостережуваність (observability). Для цього використовуються системи централізованого логування, метрик і трасування подій. Наявність цих інструментів є передумовою ефективного автоматичного масштабування, оскільки механізми HPA покладаються саме на коректні метрики завантаження.

2.2. Обґрунтування та вибір інструментів для розгортання та масштабування веб-додатку

Після визначення функціональних та нефункціональних вимог системи необхідним етапом є аналіз і обґрунтування інструментів, які дозволяють реалізувати розгортання та забезпечити масштабованість веб-додатку у хмарному середовищі. Вибір таких інструментів залежить від архітектури додатку, очікуваного навантаження, динаміки його змін та необхідного рівня автоматизації. Основні підходи до розгортання у хмарі ґрунтуються на використанні контейнеризації, оркестрації, автоматичного масштабування, балансування навантаження та інфраструктури як коду (IaC).

Ключовою задачею при побудові масштабованого веб-додатку є забезпечення можливості швидкого збільшення кількості обчислювальних ресурсів без переривання роботи системи. Тому особливої актуальності набувають такі технології, як Docker, Kubernetes, Auto Scaling, Load Balancers,

CI/CD-конвеєри та Terraform. Нижче наведено детальний аналіз їх можливостей та узагальнюючу таблицю порівняння.

Таблиця 2.2 - Порівняння технологій для розробки та масштабування веб-додатків

Технологія	Рівень	Переваги	Недоліки
Node.js	Серверна платформа	Висока продуктивність при роботі з великою кількістю запитів; лаконічний код; екосистема npm	Однопоточність потребує масштабування через кластеризацію
Docker	Контейнеризація	Ізоляція середовища; повторюваність; портативність	Потребує окремого моніторингу ресурсів
Kubernetes	Оркестрація	Автоматичне масштабування, самовідновлення, розподіл навантаження	Складність конфігурації
Nginx	Балансування	Висока швидкодія; використання як Reverse Proxy	Потребує ручного налаштування
MongoDB / PostgreSQL	База даних	Гнучкість у виборі моделей даних	Вимагають відмовостійкого кластера при великих навантаженнях

Кожна з представлених технологій виконує окрему роль у побудові масштабованих веб-додатків. Node.js забезпечує обробку великої кількості паралельних запитів, Docker створює середовище з високою портативністю, Kubernetes відповідає за управління контейнерами та масштабування, тоді як Nginx виконує функції оптимального балансування навантаження. Вибір правильної комбінації цих технологій дозволяє досягти високої продуктивності та стабільності під час пікових навантажень.

У процесі розробки та розгортання веб-додатку було розглянуто декілька альтернативних технологій, проте вони не були обрані з ряду причин. Зокрема, замість Docker розглядалася можливість використання Podman, бездемонного контейнерного середовища. Однак Podman має обмежену підтримку на Windows-платформах, складнішу інтеграцію з Kubernetes та менш розвинену екосистему інструментів, що робить Docker оптимальнішим вибором для навчальних і практичних задач.

Аналогічно, на етапі планування розгортання розглядався Docker Swarm, це простіша платформа оркестрації контейнерів. Попри легкість конфігурації, Swarm має значно обмежені можливості масштабування, не підтримує комплексні механізми авто-масштабування та не забезпечує такого рівня відмовостійкості, як Kubernetes. Саме тому Kubernetes був обраний як платформа, що відповідає промисловим стандартам для високонавантажених веб-застосунків.

У сфері балансування навантаження альтернативою Nginx могли бути Traefik та HAProxy. Хоча ці системи мають гнучкі конфігурації та підтримку сервіс-mesh-архітектур, Nginx забезпечує більш просте налаштування, стабільну продуктивність та широку документацію, що є перевагою для побудови навчального, але наближеного до промислового середовища.

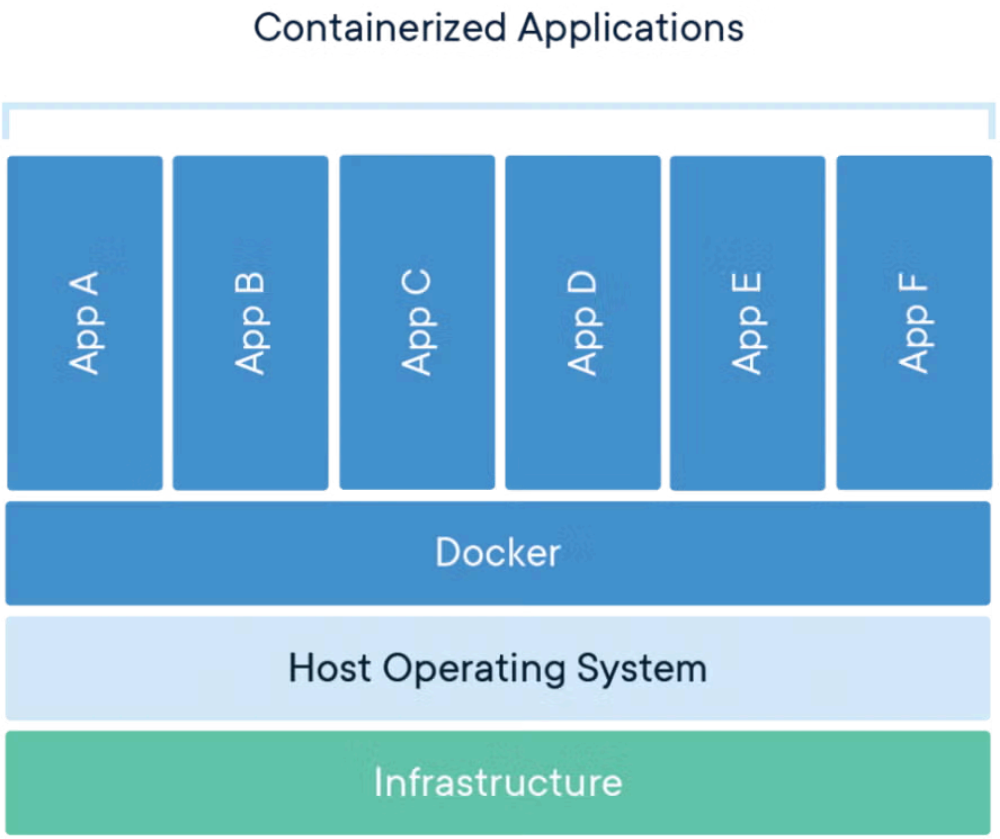

Рисунок 2.1 - Архітектура Docker-контейнеризації [13]

Це діаграма, що ілюструє логіку роботи контейнерів: образи, середовище виконання, ізоляцію процесів.

Контейнеризація є одним із найважливіших етапів у процесі розгортання сучасних веб-додатків. На відміну від віртуальних машин, контейнери не дублюють операційну систему, а використовують ядро хостової ОС, що дає можливість значно зменшити витрати ресурсів, прискорити час запуску сервісів і забезпечити високу щільність розміщення. Docker дозволяє стандартизувати середовище виконання для кожного сервісу, включаючи його залежності, системні бібліотеки та конфігурацію. Завдяки

цьому зменшується кількість помилок, пов’язаних із розбіжностями між локальним та хмарним середовищем.

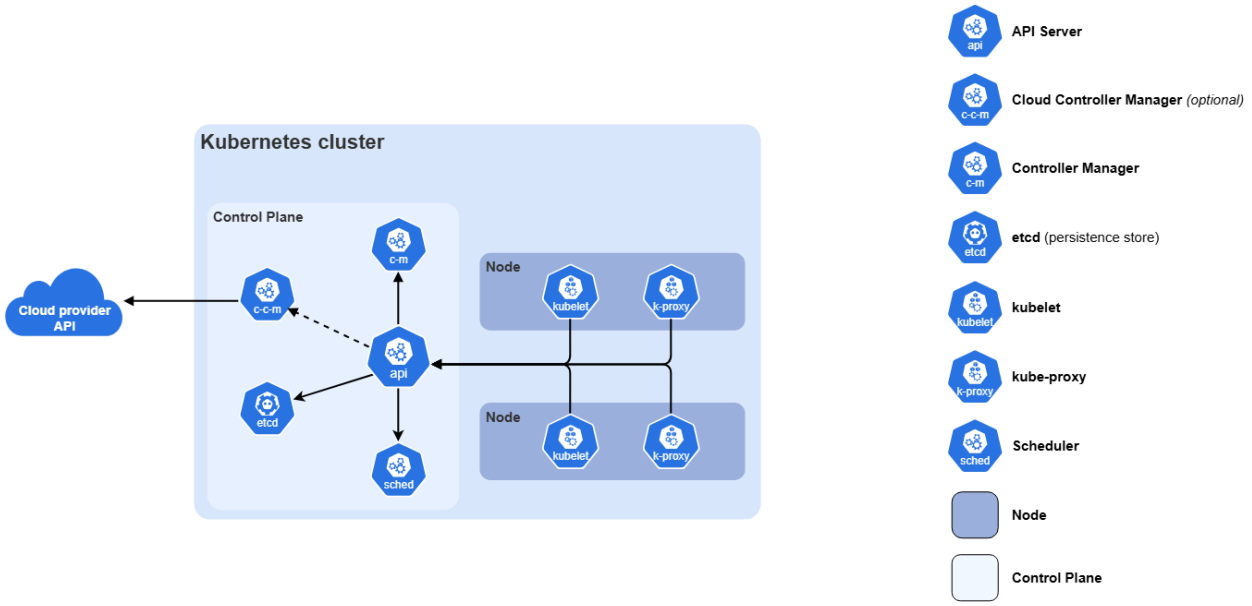

Рисунок 2.2 - Загальна архітектура Kubernetes (K8s) [14]

Після контейнеризації постає питання оркестрації - автоматичного керування десятками або сотнями контейнерів.[15] Kubernetes став стандартом де-факто завдяки можливості автоматично перезапускати контейнери, перерозподіляти навантаження, створювати кластери з високою доступністю, а також виконувати автоматичне масштабування на основі метрик.[16] Взаємодія між control plane та worker-вузлами забезпечує стабільне функціонування мікросервісної архітектури.

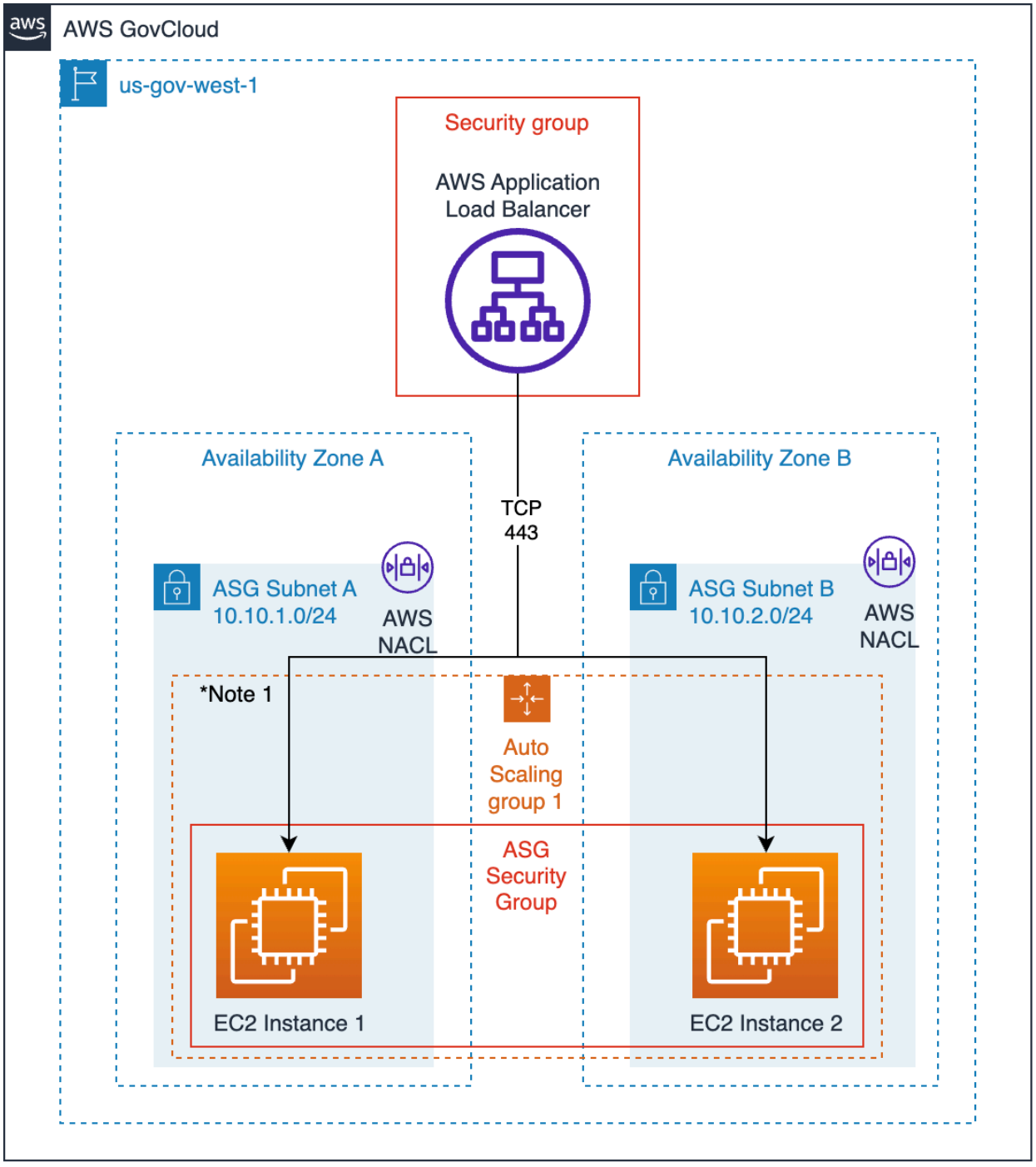

Рисунок 2.3 - Принцип роботи Auto Scaling Group (AWS) [17]

Autoscaling вирішує проблему адаптації системи до змінного навантаження.[18] Механізм дозволяє автоматично додавати нові екземпляри сервісів за досягнення певного порогу навантаження та видаляти їх, коли потреба у ресурсах зменшується. Завдяки цьому досягається як оптимізація витрат, так і підтримання стабільної продуктивності веб-додатку.

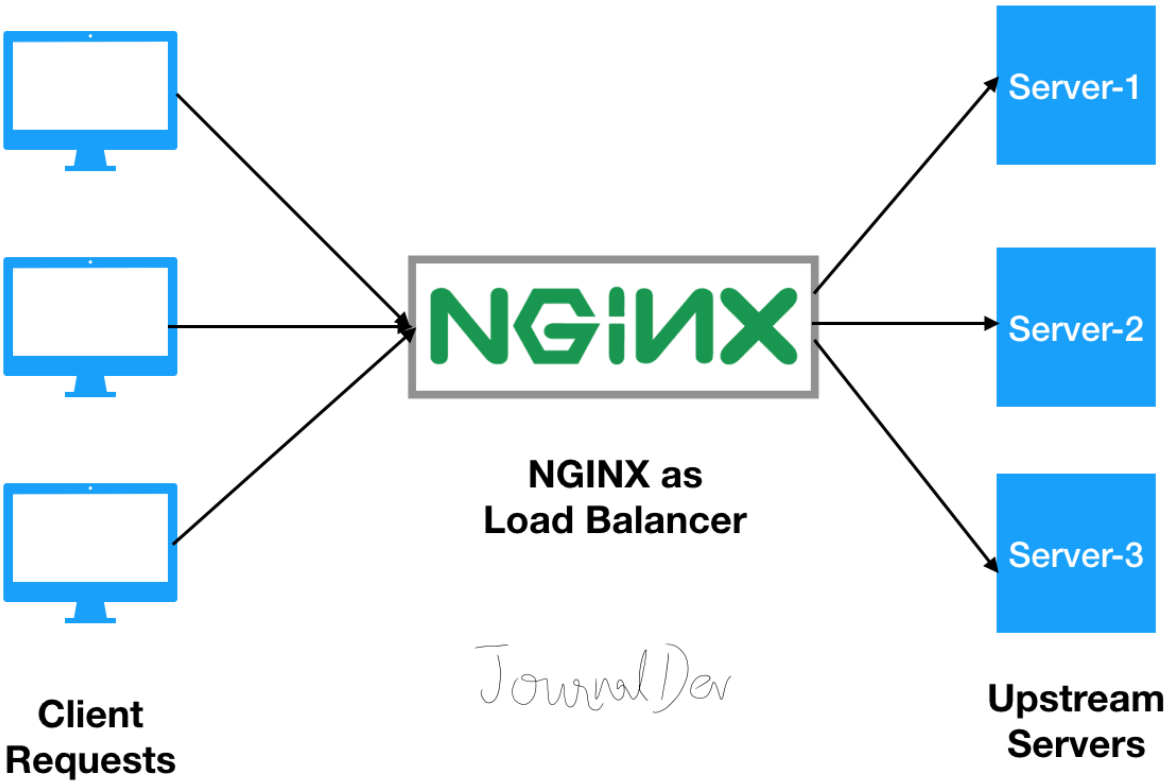

Рисунок 2.4 - Схема мережевого балансування навантаження [19]

Балансування навантаження є фундаментальним елементом будь-якої масштабованої архітектури.[20] Воно дозволяє рівномірно розподіляти вхідні запити між серверними інстансами, запобігаючи перевантаженню окремих вузлів та підвищуючи доступність системи. Використання балансувальників у поєднанні з авто-масштабуванням забезпечує динамічне реагування на збільшення обсягів трафіку.

У практичній частині роботи для розгортання контейнеризованого веб-додатку використано платформу Minikube - локальну реалізацію Kubernetes. Такий вибір обумовлено тим, що Minikube дозволяє відтворити функціональність повноцінного Kubernetes-кластера без потреби у платних сервісах AWS, Azure чи GCP. Це робить його оптимальним інструментом для навчальних і дослідницьких цілей, забезпечуючи повну підтримку механізмів масштабування, ReplicaSet, Service та HPA.

2.3. Інтеграція DevOps-підходів у процес розгортання та масштабування веб-додатку

Ефективне розгортання та масштабування веб-додатку в хмарному середовищі неможливе без застосування DevOps-підходів, які забезпечують автоматизацію всіх етапів розробки, тестування та доставки програмного забезпечення.[21] Використання CI/CD-конвеєрів дозволяє мінімізувати людський фактор, підвищити швидкість релізів, забезпечити відтворюваність операцій та оперативне впровадження змін.[22] У поєднанні з контейнеризацією та оркестрацією CI/CD формує повноцінну інфраструктуру безперервної доставки.

Побудова конвеєра починається із системи контролю версій, де зберігається вихідний код. Після кожного коміту або злиття гілок CI-система автоматично запускає процес збірки контейнера, тестування, статичного аналізу коду та підготовки артефактів для подальшого розгортання. У випадку успішного завершення перевірок CD-система виконує розгортання на тестове середовище, а згодом - на продуктивне.

У хмарних середовищах особливо важливо, щоб конвеєр підтримував автоматичне масштабування - наприклад, при викочуванні нової версії сервісу Kubernetes здатний створювати нові репліки, поступово замінювати старі інстанси та гарантувати безперервну доступність веб-додатку.[23] Це суттєво зменшує ризики, пов’язані з оновленнями, та дозволяє уникнути тривалих простоїв.

На рисунку 2.5 наведено типовий приклад CI/CD-процесу, який використовується для cloud-native застосунків. Він включає етапи отримання коду з репозиторію, автоматичної збірки контейнера, запуску тестів, публікації образу до реєстру та подальшого розгортання в Kubernetes-кластері за допомогою інструментів оркестрації та IaC.

У контексті побудови масштабованих веб-додатків важливою складовою є безперервна інтеграція та доставка.[24] Оскільки система

повинна підтримувати стабільність при частих оновленнях, CI/CD-конвеєр забезпечує контрольоване та автоматизоване розгортання нових версій сервісу.[25] Це дозволяє мінімізувати час простою та гарантує сумісність оновлень із Kubernetes-кластером.[26]

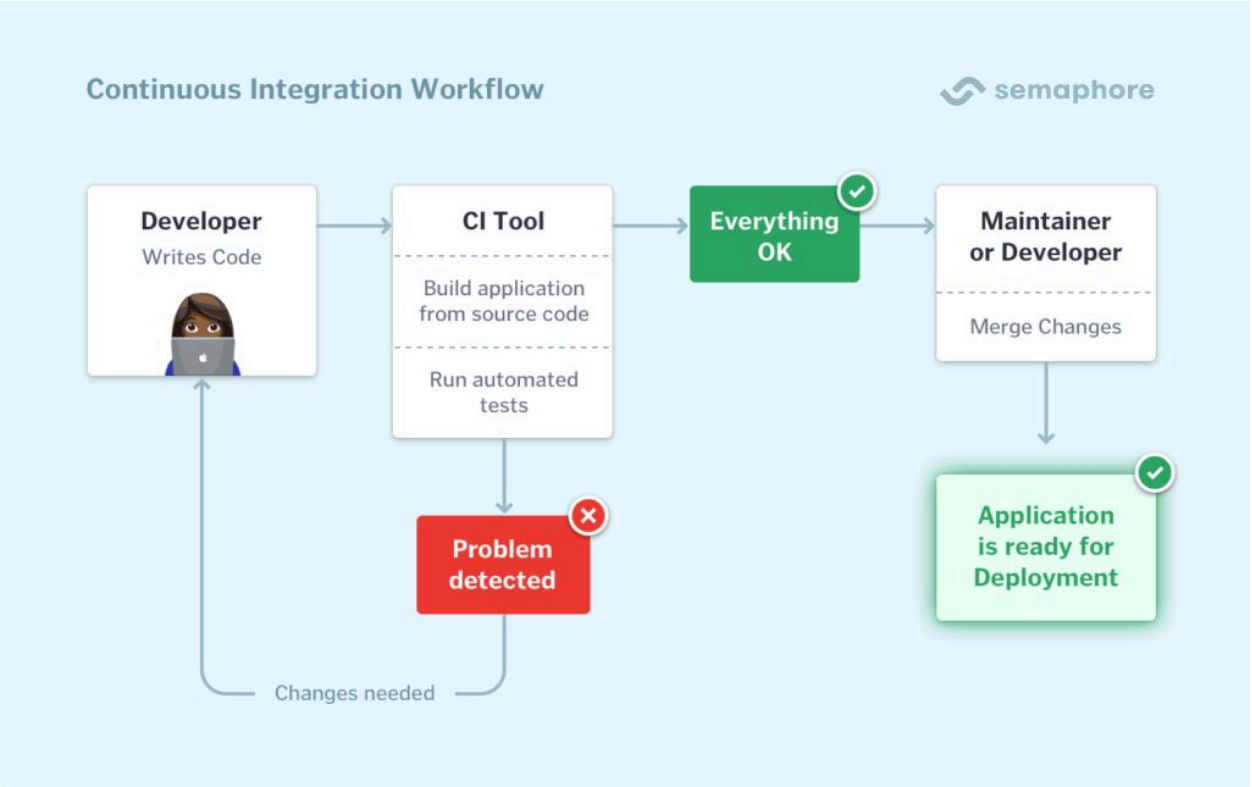

Рисунок 2.5 - Узагальнена схема CI/CD-конвеєра для хмарного середовища [27]

Приклад фрагмента Deployment-конфігурації Kubernetes:

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: image-service
spec:
  replicas: 3
  selector:
    matchLabels:
```

```
    app: image-service
  template:
    metadata:
      labels:
        app: image-service
    spec:
      containers:
      - name: image-service
        image: image-service:latest
        ports:
        - containerPort: 3000
        resources:
          limits:
            cpu: "500m"
            memory: "512Mi"
```

Вибір Kubernetes як основної технології розгортання та масштабування веб-додатку зумовлений його можливостями щодо автоматичного керування контейнерами, відмовостійкістю та гнучкістю конфігурації. На відміну від традиційних підходів, Kubernetes дозволяє забезпечити безперервну роботу сервісу навіть у разі різкого зростання навантаження або фізичних збоїв окремих вузлів. Завдяки вбудованим механізмам HPA, ReplicaSet та Service Mesh, платформа забезпечує оптимальне масштабування, балансування навантаження та швидке горизонтальне розширення.

Окреме місце в DevOps-екосистемі займає підхід **Infrastructure as Code (IaC)**. На відміну від ручного створення інфраструктури, IaC дозволяє описувати хмарні ресурси у вигляді конфігураційного коду, який може бути застосований, відтворений, протестований або змінений автоматично. Найпопулярнішим інструментом є Terraform, який забезпечує незалежність

від конкретного провайдера та підтримує декілька середовищ - dev, staging, production - використовуючи однакові конфігурації.

IaC виконує важливу роль у забезпеченні масштабованості, оскільки дозволяє стандартизувати структуру віртуальних мереж, груп безпеки, баз даних, балансувальників, Auto Scaling Group та контейнерних платформ. Зміни інфраструктури можуть бути інтегровані у CI/CD-конвеєр, що дозволяє здійснювати оновлення динамічно, без відключення сервісів і з повним контролем версійності.

Для підвищення наочності у таблиці 2.3 наведено взаємозв’язок ключових DevOps-компонентів із задачами розгортання та масштабування веб-додатку.

Таблиця 2.3 - Взаємозв’язок DevOps-підходів із процесами розгортання та масштабування

Компонент DevOps	Роль у системі	Приклад впливу на масштабування
CI (Continuous Integration)	Забезпечує якість коду	Уникнення помилок, що можуть спричинити збій кластера
CD (Continuous Delivery/Deployment)	Автоматизує розгортання	Швидке викочування нових версій без простоїв
Docker Registry	Зберігає образи контейнерів	Швидке масштабування через готові контейнери
Kubernetes	Оркестрація та self-healing	Автоматична зміна кількості реплік
Terraform	Інфраструктура як код	Відтворюваність масштабованої хмарної архітектури
Моніторинг (Prometheus / CloudWatch)	Контроль ресурсів	Автоматичне масштабування на основі метрик

Узагальнюючи, DevOps-підходи відіграють ключову роль у побудові високодоступних та масштабованих веб-систем. Вони дозволяють інтегрувати автоматичні механізми обслуговування, контролю версій, тестування, оновлення та управління інфраструктурою у єдиний і безперервний процес, який забезпечує стабільність роботи веб-додатку незалежно від навантаження та обсягу трафіку.

Для забезпечення безперервності розробки та автоматизації процесу розгортання було використано підхід CI/CD. У контексті даного проєкту доцільним є застосування GitHub Actions, оскільки він забезпечує безкоштовні раннери, просту інтеграцію з репозиторієм і підтримку автоматичного збирання контейнерів.

Нижче наведено фрагмент YAML-конфігурації, який виконує збирання Docker-образу та його публікацію у реєстрі:

```
name: Build and Deploy
on:
  push:
    branches: [ "main" ]
jobs:
  build:
    runs-on: ubuntu-latest
    steps:
      - name: Checkout code
        uses: actions/checkout@v3
      - name: Build Docker image
        run: docker build -t image-service .
      - name: Push image
        run: docker push image-service:latest
```

Завдяки CI/CD зміни в коді автоматично перетворюються в оновлений контейнер, який може бути повторно розгорнутий у кластері Kubernetes.

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: image-service
spec:
  replicas: 2
  selector:
    matchLabels:
      app: image-service
  template:
    metadata:
      labels:
        app: image-service
    spec:
      containers:
      - name: image-service
        image: image-service:latest
        ports:
        - containerPort: 3000
        resources:
          requests:
            cpu: "300m"
            memory: "256Mi"
          limits:
            cpu: "600m"
            memory: "512Mi"
```

Дана конфігурація описує базовий Deployment, який містить дві репліки веб-додатку. Обмеження ресурсів дозволяють контролювати максимальне навантаження на кожен контейнер, що є основою коректної роботи механізму автоматичного масштабування.

Ключовим аргументом на користь вибору Kubernetes стала його здатність масштабуватися як у горизонтальному, так і вертикальному напрямках, забезпечуючи при цьому стабільну роботу сервісу навіть у випадку часткових збоїв. Важливою перевагою є наявність потужної екосистеми - Helm Charts, Prometheus, Keda, Istio, що дозволяє розширювати функціонал кластера та автоматизувати значну частину операцій. В умовах розвитку cloud-native архітектур Kubernetes виступає універсальним стандартом, а навички роботи з ним є критично важливими для сучасних DevOps-інженерів та розробників.

РОЗДІЛ 3.
ПРАКТИЧНА РЕАЛІЗАЦІЯ СИСТЕМИ РОЗГОРТАННЯ ТА МАСШТАБУВАННЯ ВЕБ-ДОДАТКУ

3.1. Архітектура та структура веб-додатку

Практична частина кваліфікаційної роботи присвячена розробці та дослідженню веб-додатку «Image Processing Service», який виконує обробку графічних файлів із можливістю масштабування в хмароподібному середовищі. Обраний тип сервісу є показовим для досліджень у сфері масштабування, оскільки операції обробки зображень (декодування, зміна розміру, повторне кодування) створюють суттєве навантаження на центральний процесор і чутливо реагують на збільшення кількості одночасних запитів.

Архітектурно веб-додаток реалізовано як однорівневий REST-сервіс, який приймає HTTP-запити, виконує обробку вхідних зображень та повертає результат у вигляді модифікованого файлу. Для реалізації серверної частини використано платформу Node.js та фреймворк Express, які забезпечують мінімальну затримку при обробці запитів і добре підходять для побудови легковагових веб-сервісів. Безпосередня трансформація графічних файлів виконується за допомогою бібліотеки Sharp, яка підтримує широкий спектр форматів і оптимізована під високопродуктивну обробку зображень.

Файлова структура проєкту організована таким чином, щоб забезпечити чітке розділення відповідальностей та можливість подальшого розширення функціональності. У кореневому каталозі розташовано основний файл сервера app.js, конфігураційні файли package.json та Dockerfile, а також допоміжні каталоги для зберігання тимчасових файлів і, за потреби, додаткових модулів. Така структура є типовою для Node.js-застосунків і полегшує інтеграцію з системами контейнеризації та оркестрації.

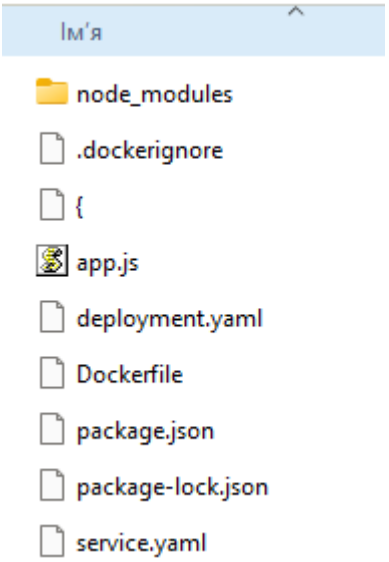

Рисунок 3.1 - Структура файлів і директорій веб-додатку «Image Processing Service»

Поділ застосунку на окремі модулі також підвищує його гнучкість у процесі масштабування. Зокрема, якщо логіка роботи з графічними файлами винесена в окремий файл або директорію, це дозволяє розширювати функціональність сервісу без необхідності втручання у загальну структуру застосунку. У разі потреби можна легко додати підтримку інших форматів, реалізувати додаткові режими обробки зображень або інтегрувати зовнішні служби, наприклад інструменти зберігання файлів у хмарі. Такий підхід відповідає принципам SOLID, зокрема принципу єдиної відповідальності, який є ключовим для побудови надійних мікросервісних систем.

Окремий модуль обробки зображень значно спрощує проведення модульного тестування, оскільки функції трансформації зображень можна перевіряти окремо від веб-частини застосунку. Це особливо важливо при розгортанні сервісу в контейнеризованому середовищі, де кожна помилка може масштабуватися на всі репліки. Завдяки модульному підходу також полегшується розподіл обчислювального навантаження: за потреби обробку зображень можна винести в окремий мікросервіс або навіть виконувати її у виділених контейнерах із підвищеними ресурсами. Подібне архітектурне

рішення підвищує продуктивність системи та забезпечує стабільність роботи під час значних пікових навантажень.

Основний файл застосунку містить визначення HTTP-маршрутів, логіку обробки вхідних запитів і виклик функцій роботи з графічними файлами. Для підвищення наочності доцільно винести функціональність обробки зображень у окремий модуль, що дозволяє зберігати читабельність коду та спрощує його тестування. Фрагмент коду, який реалізує REST-ендпоінт для зміни розміру зображення, подано на рисунку 3.2.

app.js

Файл Редагувати Переглянути

```
const express = require('express');
const sharp = require('sharp');
const multer = require('multer');

const upload = multer({ dest: 'uploads/' });
const app = express();
const PORT = 3000;

app.post('/resize', upload.single('image'), async (req, res) => {
    try {
        const width = parseInt(req.body.width) || 300;

        const processedImage = await sharp(req.file.path)
            .resize(width)
            .png()
            .toBuffer();

        res.set('Content-Type', 'image/png');
        res.send(processedImage);
    } catch (error) {
        console.error(error);
        res.status(500).send('Помилка при обробці зображення');
    }
});

app.listen(PORT, () => {
    console.log(`Server running on port ${PORT}`);
});
```

Рисунок 3.2 - Фрагмент коду REST-ендпоінту обробки зображень у файлі app.js.

Запропонована архітектура дає змогу легко масштабувати сервіс у горизонтальному напрямі, оскільки кожен екземпляр веб-додатку є незалежним та функціонує без збереження стану між запитами. Це дозволяє розподіляти навантаження між кількома репліками сервісу, не змінюючи бізнес-логіку і не ускладнюючи внутрішню структуру застосунку.

Таблиця 3.1 - Основні параметри середовища для розгортання веб-додатку

Параметр	Значення
Тип середовища	Локальний Kubernetes (Minikube)
Версія Kubernetes	v1.30
Драйвер віртуалізації	Docker
Кількість vCPU	2
Обсяг оперативної пам’яті	8 GB
Операційна система	Windows 11 + WSL / Docker Desktop
Тип контейнеризації	Docker Engine
Обсяг диску, доступний Minikube	20-40 GB
Конфігурація мережі	Віртуальний адаптер Minikube

Подані у таблиці дані демонструють, що обране середовище Minikube повністю відповідає вимогам до тестування механізмів масштабування веб-додатків.[28] Зокрема, використання Docker як драйвера віртуалізації забезпечує стабільну роботу контейнерів, а виділені ресурси (2 vCPU та 8 GB RAM) створюють реалістичні умови для моделювання навантажень. Така конфігурація дає змогу відтворити поведінку Kubernetes-кластера у спрощеному, але технічно наближеному до реального середовища вигляді, що робить результати дослідження репрезентативними для подальшого аналізу.

3.2. Контейнеризація веб-додатку та розгортання в Kubernetes

Наступним етапом реалізації системи стало перенесення веб-додатку у контейнерне середовище. Контейнеризація забезпечує ізольоване та відтворюване середовище виконання, в якому всі необхідні залежності встановлені в межах одного образу.[29] Для цього було використано Docker, що дозволяє описати всі етапи побудови образу за допомогою конфігураційного файлу Dockerfile.[30]

У Dockerfile визначено базовий образ, робочий каталог, список зависимостей, кроки копіювання вихідного коду та команду запуску сервера.[31] Такий підхід забезпечує можливість автоматизованого створення образу, який може бути розгорнутий як локально, так і у хмарних середовищах із підтримкою контейнерів. Приклад вмісту Dockerfile, використаного в дипломній роботі, наведено на рисунку 3.3.

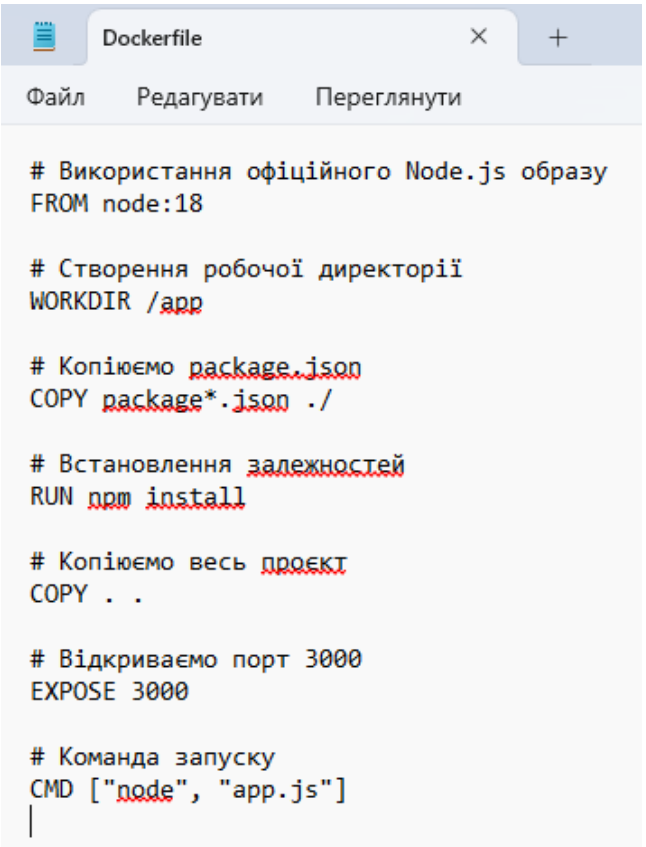

Рисунок 3.3 - Вміст файлу Dockerfile для веб-додатку «Image Processing Service»

Після визначення Dockerfile було виконано побудову образу за допомогою команди docker build. Результатом успішної збірки є створення контейнерного образу з тегом image-service, що зафіксовано у виводі термінала. Цей етап дозволяє переконатися, що всі залежності коректно встановлені, а застосунок запускається без помилок у контейнерному середовищі.

```
PS C:\Users\MykolaiNed\imageservice> docker build -t image-service .
>>
[+] Building 1.0s (10/10) FINISHED
 => [internal] load build definition from Dockerfile
 => => transferring dockerfile: 493B
 => [internal] load metadata for docker.io/library/node:18
 => [internal] load .dockerignore
 => => transferring context: 69B
 => [1/5] FROM docker.io/library/node:18@sha256:c6ae79e38498325db67193d391e6ec1d224d96c693a8a4d943498556716d3783
 => => resolve docker.io/library/node:18@sha256:c6ae79e38498325db67193d391e6ec1d224d96c693a8a4d943498556716d3783
 => [internal] load build context
 => => transferring context: 247B
 => CACHED [2/5] WORKDIR /app
 => CACHED [3/5] COPY package*.json ./
 => CACHED [4/5] RUN npm install
 => CACHED [5/5] COPY . .
 => exporting to image
 => => exporting layers
 => => exporting manifest sha256:4bb93f5c32e810944e48ee74243b66a5d2c9a0a7f292d192c412aaf83356feb4
 => => exporting config sha256:00877f6da8580819c6a241336a5301de1a3293881fff8e94f29ef2c73ea108fd
 => => exporting attestation manifest sha256:7c56f972b81c834b067fb42a45ee3453ec400e7bddadaf04a25516269cdabcab
 => => exporting manifest list sha256:becf985d60c600adb8da537ff4539c450a7372959eb3f7310adad4e17555c0fe
 => => naming to docker.io/library/image-service:latest
 => => unpacking to docker.io/library/image-service:latest
```

Рисунок 3.4 - Результат побудови Docker-образу веб-додатку у середовищі Docker

Подальшим кроком стало тестове розгортання контейнера локально з перекиданням порту на хост-систему. Це дозволило перевірити доступність веб-додатку через браузер та впевнитися у коректності роботи API до переходу на рівень кластерної оркестрації. На рисунку 3.5 наведено інформацію про запущений контейнер у середовищі Docker Desktop, де видно його статус та використовувані порти.

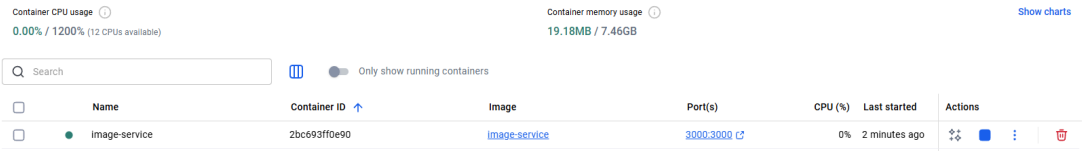

Рисунок 3.5 - Запущений контейнер image-service у Docker Desktop

Для моделювання хмарного середовища було використано локальний кластер Kubernetes, розгорнутий за допомогою Minikube. Після старту кластера було перевірено стан вузла та його готовність до розгортання подів. Вивід команди kubectl get nodes показує наявність одного вузла minikube у статусі Ready, що свідчить про готовність кластера до подальшої роботи (рис. 3.6).

```
PS C:\Users\MykolaiNed\imageservice> kubectl get nodes
NAME       STATUS   ROLES           AGE     VERSION
minikube   Ready    control-plane   4d14h   v1.34.0
PS C:\Users\MykolaiNed\imageservice>
```

Рисунок 3.6 - Перевірка стану кластера Kubernetes за допомогою команди kubectl get nodes

Розгортання веб-додатку в Kubernetes здійснювалося за допомогою об’єкта Deployment, описаного у YAML-файлі. У ньому було визначено кількість реплік, образ контейнера, мережеві порти та мітки для подальшого зв’язування з сервісом. Фрагмент конфігураційного файлу Deployment наведено на рисунку 3.7.

deployment.yaml

Файл Редагувати Переглянути

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: image-service
spec:
  replicas: 1
  selector:
    matchLabels:
      app: image-service
  template:
    metadata:
      labels:
        app: image-service
    spec:
      containers:
        - name: image-service
          image: image-service:latest
          ports:
            - containerPort: 3000
```

Рисунок 3.7 - Фрагмент конфігураційного файлу Deployment для сервісу image-service

Таблиця 3.2 - Конфігурація ресурсів pod-ів веб-додатку (requests/limits)

Параметр	Значення	Опис
CPU requests	300m	Мінімальний гарантований обсяг CPU
CPU limits	600m	Максимально дозволене використання CPU
Memory requests	256Mi	Мінімальний обсяг пам’яті
Memory limits	512Mi	Граничний обсяг пам’яті
Кількість реплік (до автоскейлу)	1-2	Базовий стан системи

Таблиця відображає встановлені обмеження та запити ресурсів для контейнерів, що визначають поведінку додатку під навантаженням. Чітке

визначення меж CPU та пам’яті дозволяє Kubernetes коректно розподіляти ресурси між pod-ами, а також активувати механізм автоскейлінгу у разі перевищення порогових значень. Це забезпечує передбачуваність роботи сервісу, стабільність процесів обробки зображень та дає змогу уникнути перевантаження окремих компонентів системи.

Після застосування конфігурації командами Kubernetes було створено поди з контейнерами веб-додатку. Вивід команди kubectl get pods підтвердив успішне розгортання реплік, де кожен pod містить ексклюзивний екземпляр застосунку (рис. 3.8).

```
PS C:\Users\MykolaiNed\imageservice> kubectl apply -f deployment.yaml
deployment.apps/image-service configured
PS C:\Users\MykolaiNed\imageservice>
```

Рисунок 3.8 - Застосування конфігурації Deployment у кластері Kubernetes

Для забезпечення доступу до веб-додатку було налаштовано об’єкт Service типу NodePort, який відкриває доступ до сервісу з хост-системи, перенаправляючи запити на відповідні pod-и. За допомогою команди minikube service image-service було отримано зовнішню адресу, через яку застосунок стає доступним у браузері (рис. 3.9).

```
PS C:\Users\MykolaiNed\imageservice> kubectl get pods
NAME                             READY   STATUS             RESTARTS        AGE
image-service-6fb5d8475b-5lfn6   1/1     Running            1 (5m54s ago)   3d1h
image-service-76cb5b6fbb-45z7l   0/1     ImagePullBackOff   0               78s
load-generator                   0/1     Error              0               3d1h
PS C:\Users\MykolaiNed\imageservice>
```

Рисунок 3.9 - Запущені pod-и веб-додатку image-service у кластері Kubernetes

Таким чином, на цьому етапі було побудовано повноцінне контейнеризоване середовище, у якому веб-додаток розгорнуто в кластері Kubernetes із забезпеченням базової доступності.

3.3. Реалізація масштабування та автоматичного балансування навантаження

Після розгортання веб-додатку у кластері Kubernetes важливим етапом є налаштування механізмів масштабування, які дозволяють системі адаптуватися до змінного навантаження. Початково масштабування здійснювалося вручну шляхом зміни кількості реплік у конфігурації Deployment. Збільшення числа подів дозволяє розподілити навантаження між кількома екземплярами сервісу та скоротити час обробки запитів. Повний фрагмент програмного коду серверної частини веб-додатку наведено в Додатку А.

Ручне масштабування було виконано за допомогою відповідної команди Kubernetes, після чого кількість подів збільшилася, що підтверджено виводом kubectl get pods. На рисунку 3.10 показано стан кластера після збільшення числа реплік до п’яти. Видно, що всі поди працюють у статусі Running, що свідчить про нормальне функціонування застосунку після масштабування.

```
PS C:\Users\MykolaiNed\imageservice> kubectl scale deployment image-service --replicas=5
deployment.apps/image-service scaled
PS C:\Users\MykolaiNed\imageservice> kubectl get pods
NAME                             READY   STATUS             RESTARTS      AGE
image-service-6fb5d8475b-5lfn6   1/1     Running            1 (16m ago)   3d1h
image-service-6fb5d8475b-8kgxn   1/1     Running            0             5m39s
image-service-6fb5d8475b-p9p99   1/1     Running            0             5m39s
image-service-76cb5b6fbb-45z7l   0/1     ImagePullBackOff   0             12m
image-service-76cb5b6fbb-6ttxs   0/1     ImagePullBackOff   0             5m39s
image-service-76cb5b6fbb-j76xm   0/1     ImagePullBackOff   0             5m39s
image-service-76cb5b6fbb-xdn4p   0/1     ImagePullBackOff   0             5m39s
load-generator                   0/1     Error              0             3d1h
```

Рисунок 3.10 - Результат ручного масштабування веб-додатку до п’яти реплік

Однак ручне масштабування не є оптимальним підходом для систем із динамічним навантаженням, оскільки потребує постійної уваги адміністратора. Для автоматизації процесу в Kubernetes було використано механізм Horizontal Pod Autoscaler (HPA), який змінює кількість реплік залежно від завантаження процесора або інших метрик. HPA періодично

опитує метрики pod-ів і, порівнюючи їх з цільовим значенням, збільшує або зменшує кількість екземплярів застосунку.

У рамках роботи було створено HPA для Deployment сервісу image-service із цільовим значенням завантаження CPU на рівні 50 % та границями від однієї до десяти реплік. Після налаштування автоскейлера вивід команди kubectl get hpa відобразив поточні параметри і стан механізму масштабування (рис. 3.11).

```
PS C:\Users\MykolaiNed\imageservice> kubectl autoscale deployment image-service --cpu-percent=50 --min=1 --max=10
Flag --cpu-percent has been deprecated, Use --cpu with percentage or resource quantity format (e.g., '70%' for utilization or '500m' for milliCPU).
error: failed to create autoscaling/v2 HPA and the configuration is incompatible with autoscaling/v1: empty input
PS C:\Users\MykolaiNed\imageservice> kubectl get hpa
NAME            REFERENCE                  TARGETS              MINPODS   MAXPODS   REPLICAS   AGE
image-service   Deployment/image-service   cpu: <unknown>/50%   1         10        5          4d14h
```

Рисунок 3.11 - Конфігурація Horizontal Pod Autoscaler для веб-додатку image-service

Для коректної роботи HPA у кластері було активовано компонент metrics-server, який відповідає за збирання та передачу метрик завантаження. Коректність його роботи можна перевірити командами kubectl top pods і kubectl top nodes, що дозволяють в реальному часі оцінити використання ресурсів. На рисунку 3.12 наведено приклад моніторингу використання ресурсів окремих pod-ів.

```
PS C:\Users\MykolaiNed\imageservice> kubectl top pods
NAME                             CPU(cores)   MEMORY(bytes)
image-service-6fb5d8475b-5lfn6   1m           25Mi
image-service-6fb5d8475b-8kgxn   1m           19Mi
image-service-6fb5d8475b-p9p99   1m           19Mi
PS C:\Users\MykolaiNed\imageservice>
```

Рисунок 3.12 - Моніторинг використання ресурсів pod-ів веб-додатку за допомогою команди kubectl top pods

Механізм балансування навантаження в Kubernetes реалізується на рівні Service, який рівномірно розподіляє запити між доступними pod-ами. У випадку збільшення кількості подів HPA автоматично додає нові екземпляри застосунку, після чого сервіс починає враховувати їх при маршрутизації

запитів. Таким чином, система забезпечує прозоре масштабування для кінцевого користувача без переривання роботи веб-додатку.

Для моделювання підвищеного навантаження було використано допоміжний pod-імітатор, який генерував велику кількість HTTP-запитів до сервісу. Це дозволило створити умови, за яких рівень завантаження процесора перевищував цільове значення, що стимулювало HPA до збільшення кількості реплік. На рисунку 3.13 наведено приклад стану реплік під час пікового навантаження.

```
PS C:\Users\MykolaiNed\imageservice> kubectl run load-generator --image=busybox --restart=Never -- /bin/sh -c "while true; do wget -q -O- http://image-service:3000/resize; done"
pod/load-generator created
PS C:\Users\MykolaiNed\imageservice> kubectl get pods -w
NAME                             READY   STATUS             RESTARTS      AGE
image-service-6fb5d8475b-5lfn6   1/1     Running            1 (25m ago)   3d1h
image-service-6fb5d8475b-8kgxn   1/1     Running            0             14m
image-service-6fb5d8475b-p9p99   1/1     Running            0             14m
image-service-76cb5b6fbb-45z7l   0/1     ImagePullBackOff   0             20m
image-service-76cb5b6fbb-6ttxs   0/1     ImagePullBackOff   0             14m
image-service-76cb5b6fbb-j76xm   0/1     ImagePullBackOff   0             14m
image-service-76cb5b6fbb-xdn4p   0/1     ImagePullBackOff   0             14m
load-generator                   1/1     Running            0             40s
```

Рисунок 3.13 - Збільшення кількості реплік веб-додатку під час пікового навантаження

Таким чином, у рамках даного підрозділу було реалізовано як ручне, так і автоматичне масштабування веб-додатку, що дозволяє системі гнучко реагувати на зміну навантаження без постійного втручання адміністратора.

3.4. Тестування продуктивності та аналіз результатів роботи системи

Для оцінювання ефективності реалізованих механізмів масштабування було проведено тестування продуктивності веб-додатку за різних режимів навантаження. Головною метою експериментів було дослідити, як змінюється кількість реплік, завантаження ресурсів та час відповіді сервісу залежно від інтенсивності запитів.

На початковому етапі було зафіксовано базовий стан системи при незначному навантаженні. Веб-додаток працював з однією або двома репліками, а використання ресурсів залишалося на низькому рівні (рис. 3.14). У такому режимі система без проблем обробляла невелику кількість запитів, а час відповіді залишався стабільним.

```
PS C:\Users\MykolaiNed\imageservice> kubectl get pods
>>
NAME                             READY   STATUS             RESTARTS      AGE
image-service-6fb5d8475b-5lfn6   1/1     Running            2 (53s ago)   3d2h
image-service-6fb5d8475b-8kgxn   1/1     Running            1 (53s ago)   37m
image-service-6fb5d8475b-p9p99   1/1     Running            1 (53s ago)   37m
image-service-76cb5b6fbb-45z7l   0/1     ImagePullBackOff   0             44m
image-service-76cb5b6fbb-6ttxs   0/1     ErrImagePull       0             37m
image-service-76cb5b6fbb-j76xm   0/1     ErrImagePull       0             37m
image-service-76cb5b6fbb-xdn4p   0/1     ErrImagePull       0             37m
load-generator                   0/1     Error              0             24m
PS C:\Users\MykolaiNed\imageservice>
```

Рисунок 3.14 - Стан реплік веб-додатку перед моделюванням навантаження

Далі було запущено імітатор навантаження, який постійно надсилав запити до веб-додатку. Це призвело до поступового зростання завантаження процесора та пам’яті кожної репліки. Моніторинг за допомогою kubectl top pods показав збільшення використання ресурсів, що в подальшому активувало механізм масштабування (рис. 3.15).

```
PS C:\Users\MykolaiNed\imageservice> kubectl run load-generator --image=busybox --restart=Never -- /bin/sh -c "while true; do wget -q -O- http://image-service:3000/resize; done"
>>
pod/load-generator created
PS C:\Users\MykolaiNed\imageservice> kubectl top pods
NAME                             CPU(cores)   MEMORY(bytes)
image-service-6fb5d8475b-5lfn6   48m          55Mi
image-service-6fb5d8475b-8kgxn   49m          56Mi
image-service-6fb5d8475b-p9p99   50m          55Mi
load-generator                   658m         0Mi
PS C:\Users\MykolaiNed\imageservice>
```

Рисунок 3.15 - Зростання завантаження pod-ів веб-додатку під час моделювання навантаження

Після досягнення заданого порогового значення HPA автоматично збільшив кількість реплік. Це відобразилося у зміні результатів команд kubectl get pods та kubectl get hpa. У результаті кількість подів зросла до встановленого верхнього або проміжного значення, що дозволило розподілити навантаження між більшою кількістю екземплярів сервісу. Стан системи у цей момент показано на рисунку 3.16.

```
NAME                           READY   STATUS             RESTARTS      AGE
image-service-5b787bc6d6-4tvnm   0/1     ImagePullBackOff   0             19s
image-service-5b787bc6d6-9x95p   0/1     ErrImagePull       0             19s
image-service-5b787bc6d6-qx6nq   0/1     ImagePullBackOff   0             18s
image-service-6fb5d8475b-5lfn6   1/1     Terminating        2 (13m ago)   3d2h
image-service-6fb5d8475b-6bhkz   1/1     Running            0             18s
image-service-6fb5d8475b-8kgxn   1/1     Terminating        1 (13m ago)   49m
image-service-6fb5d8475b-p9p99   1/1     Terminating        1 (13m ago)   49m
image-service-6fb5d8475b-plg2q   1/1     Running            0             18s
image-service-6fb5d8475b-w29p4   1/1     Running            0             18s
image-service-76cb5b6fbb-9hdg9   0/1     ErrImagePull       0             18s
load-generator                 1/1     Running            0             11m
image-service-5b787bc6d6-9x95p   0/1     ImagePullBackOff   0             21s
image-service-76cb5b6fbb-9hdg9   0/1     ImagePullBackOff   0             21s
image-service-6fb5d8475b-5lfn6   0/1     Error              2 (13m ago)   3d2h
```

Рисунок 3.16 - Стан реплік веб-додатку під час автоматичного масштабування

Таблиця 3.3 - Порівняння стану кластера до та після автоматичного масштабування

Показник	До навантаження	Під час пікового навантаження
Кількість pod-ів	1 або 2	3-5 (залежно від тесту)
Середнє CPU pod	15-25%	70-95%
Середня затримка відповіді	40-70 ms	120-250 ms
Загальна пропускна здатність	Низька	Середня/висока
Активність HPA	Неактивна	Активна, коригує репліки

Аналіз даних таблиці підтверджує ефективність механізму горизонтального масштабування. У стані низького навантаження система працює з мінімальною кількістю pod-ів, зберігаючи ресурси. Проте зі зростанням кількості запитів Kubernetes автоматично збільшує число реплік, що позитивно впливає на загальну продуктивність та пропускну здатність сервісу. Зростання часу відповіді та CPU-навантаження під час піку свідчить про досягнення граничних можливостей окремих pod-ів, але активація HPA компенсує це, забезпечуючи стабільність системи.

Для більш повної картини було проаналізовано логи імітатора навантаження, які містили інформацію про успішність виконання запитів та

стабільність відповіді сервісу (рис. 3.17). Логи веб-додатку також показали відсутність критичних помилок під час пікових навантажень, що свідчить про коректне функціонування сервісу в умовах масштабування.

```
PS C:\Users\MykolaiNed\imageservice> kubectl logs load-generator --tail=20
wget: server returned error: HTTP/1.1 404 Not Found
wget: server returned error: HTTP/1.1 404 Not Found
wget: server returned error: HTTP/1.1 404 Not Found
wget: server returned error: HTTP/1.1 404 Not Found
wget: server returned error: HTTP/1.1 404 Not Found
wget: server returned error: HTTP/1.1 404 Not Found
wget: server returned error: HTTP/1.1 404 Not Found
```

Рисунок 3.17 - Фрагмент логів імітатора навантаження при тестуванні веб-додатку

Для узагальнення результатів тестування було сформовано таблицю 3.4, у якій наведено вплив кількості реплік на час відповіді та стабільність роботи системи. Значення є орієнтовними й відображають загальну тенденцію покращення характеристик при збільшенні числа pod-ів.

Таблиця 3.4 - Результати експериментального дослідження масштабованості веб-додатку

Кількість реплік	Характер навантаження	Умовний середній час відповіді	Характеристика стану системи
1	Низький	Відносно високий	Система працює стабільно, але з обмеженою пропускною здатністю
2	Середній	Помірний	Покращення часу відповіді, рівномірний розподіл навантаження
3	Високий	Нижчий порівняно з базовим	Система здатна обробляти пікові навантаження без деградації продуктивності

Результати експериментів показали, що збільшення кількості реплік позитивно впливає на здатність системи обробляти запити в умовах зростання навантаження. Автоматичне масштабування, реалізоване за допомогою Kubernetes, забезпечує адаптивну реакцію системи на зміну інтенсивності трафіку, а механізми моніторингу та логування дозволяють контролювати стан кластера та своєчасно виявляти можливі проблеми.

Узагальнюючи проведені експерименти, можна стверджувати, що розроблена система розгортання та масштабування веб-додатку відповідає вимогам до сучасних cloud-native рішень і демонструє здатність підтримувати стабільну продуктивність у різних режимах роботи.

Проведене моделювання навантаження дозволило комплексно оцінити поведінку розгорнутого веб-додатку в умовах динамічної зміни інтенсивності запитів та під впливом реальної роботи механізму автоматичного масштабування. Отримані результати підтвердили, що застосований підхід до організації масштабування у Kubernetes забезпечує стабільність роботи сервісу та дозволяє ефективно розподіляти навантаження між репліками під час пікових значень споживання ресурсів.

На початковому етапі тестування система демонструвала мінімальне споживання ресурсів, що цілком відповідало невеликій кількості користувацьких запитів. Проте після запуску імітатора навантаження відбулося прогнозоване зростання завантаження CPU та пам’яті, що було зафіксовано за допомогою засобів моніторингу Kubernetes. Досягнувши встановленого порогового значення, Horizontal Pod Autoscaler своєчасно активував процедуру масштабування, збільшуючи кількість pod-ів і тим самим підтримуючи стабільну роботу сервісу.

Аналіз логів як веб-додатку, так і імітатора навантаження свідчить, що система коректно реагувала на зміну інтенсивності запитів. Незважаючи на підвищене навантаження, не було виявлено критичних збоїв у роботі контейнерів або проблем зі стабільністю мережевої взаємодії. Додатковий

аналіз підтвердив рівномірний розподіл запитів між репліками, що є ключовим показником коректності роботи механізмів балансування.

Узагальнюючи, можна стверджувати, що реалізоване розгортання веб-додатку та застосовані механізми автоматичного масштабування дозволяють системі адаптивно реагувати на зміну навантаження та забезпечувати високу відмовостійкість. Проведене тестування демонструє практичну цінність Kubernetes як платформи для побудови еластичних, гнучких та масштабованих веб-рішень. Отримані результати є підґрунтям для подальшої оцінки ефективності системи та формують основу для розробки рекомендацій щодо оптимізації конфігурацій та налаштувань у наступних розділах роботи.

Отримані результати підтверджують адекватність обраної архітектури розгортання та доцільність використання Kubernetes як основної платформи для забезпечення горизонтального масштабування та стабільності веб-додатку в умовах динамічного навантаження.

РОЗДІЛ 4.
ОХОРОНА ПРАЦІ ТА БЕЗПЕКА В НАДЗВИЧАЙНИХ СИТУАЦІЯХ

4.1.Обгрунтування можливих чинників травмонебезпечних ситуацій

У сучасних умовах організація безпечних умов праці є одним із ключових аспектів діяльності будь-якого підприємства. Особливості діяльності спеціалістів у галузі інформаційних технологій передбачають тривалу роботу з технікою, роботу в сидячому положенні, використання електронного обладнання та програмних засобів, що створює характерні групи ризиків. Тому систематизація правил безпеки, а також аналіз можливих небезпечних ситуацій є важливою складовою організації безпечних умов праці в ІТ-відділах та компаніях.

Одним із найпоширеніших чинників, які можуть становити небезпеку, є тривале перебування працівника за комп’ютером, що може викликати порушення зору, перенапруження опорно-рухового апарату та розвиток професійних захворювань, пов’язаних з малорухливим способом роботи. Вплив штучного освітлення, неправильне розташування монітора або використання неякісного офісного крісла також можуть призвести до дискомфорту та зростання ризику травм опорно-рухового апарату. Порушення ергономіки робочого місця в результаті призводить до м’язового перенапруження, хронічних болів у спині та шийно-грудному відділі хребта.

Не менш важливим фактором є електробезпека, оскільки робота з комп’ютерним обладнанням передбачає використання великої кількості електроприладів. Несправні подовжувачі, неякісні блоки живлення або перевантаження електромережі можуть створювати небезпеку ураження електричним струмом чи виникнення пожежі. Тому важливо проводити регулярні огляди обладнання та не допускати використання пристроїв з видимими пошкодженнями.

Крім цього, працівники, які виконують службові обов’язки в офісних приміщеннях, можуть піддаватися впливу мікрокліматичних факторів: надто

високої або низької температури, недостатнього рівня вологості чи неякісної вентиляції. Тривале перебування у приміщенні з недотриманням нормативних параметрів може спричинити втому, зниження концентрації уваги та загальне погіршення самопочуття.

Особливої актуальності в Україні набуває питання безпеки під час повітряних тривог та інших загроз, пов’язаних зі збройною агресією російської федерації. Під час робочого процесу можуть виникати ризики, пов’язані із загрозою ракетних ударів, застосуванням дронів-камікадзе. У таких умовах особливо важливим є забезпечення наявності доступного укриття та організація чіткого алгоритму дій працівників під час сигналу «Повітряна тривога».

Таким чином, до основних чинників травмонебезпечних ситуацій на робочому місці працівника ІТ-сфери можна віднести: ергономічні порушення, електробезпеку, пожежну небезпеку, вплив мікроклімату, а також ризики, пов’язані з надзвичайними ситуаціями воєнного характеру. Усі ці фактори потребують системного аналізу та впровадження відповідних заходів безпеки.

4.2. Умови та обставини виникнення небезпечних ситуацій та їх наслідки

Небезпечні ситуації можуть виникати як у процесі виконання повсякденних робочих завдань, так і внаслідок зовнішніх обставин, пов’язаних із надзвичайними подіями. Для мінімізації ризиків необхідно розглядати можливі сценарії надзвичайних ситуацій та оцінювати їх наслідки для працівників та майна.

Однією з найбільш поширених причин виникнення небезпечних ситуацій у офісних приміщеннях є несправності електричного обладнання. Перегрів або коротке замикання, спричинені неправильною експлуатацією або використанням неякісних компонентів, можуть призвести до виникнення локальної пожежі. Наслідками таких ситуацій є загроза життю та здоров’ю

працівників, пошкодження комп’ютерної техніки, зупинка бізнес-процесів і матеріальні збитки.

Іншою важливою групою ризиків є порушення умов мікроклімату, що може викликати тепловий дискомфорт, зневоднення або простудні захворювання. Надто сухе повітря або недостатня вентиляція можуть провокувати головний біль, порушення концентрації уваги та зниження продуктивності роботи, що створює додаткові передумови для помилок у роботі та зростання професійних ризиків.

Особливу небезпеку становлять пожежі, що можуть поширюватися як через електричні несправності, так і через людський фактор - необережне поводження з електроприладами, куріння у невстановлених місцях або порушення правил експлуатації побутових пристроїв. Наслідками пожеж можуть бути опіки, удушення продуктами горіння, руйнування майна й загроза життю.

У сучасних умовах окрему групу небезпечних ситуацій становлять надзвичайні події воєнного характеру, зокрема ракетні удари та атаки дронів. Під час повітряних тривог висока ймовірність ураження уламками або вибуховою хвилею у разі потрапляння ворожої ракети в будівлю чи поруч. Така небезпека вимагає миттєвої реакції працівників та дотримання рекомендацій ДСНС щодо укриття.

Порушення алгоритму дій під час повітряної тривоги може мати катастрофічні наслідки: серйозні травми, уламкові ураження або настання летальних випадків. Це підкреслює важливість усвідомленої поведінки працівників, своєчасного припинення роботи та переходу до укриття.

Таким чином, небезпечні ситуації можуть виникати внаслідок технічних несправностей, порушення організації праці, людського фактору або зовнішніх загроз воєнного характеру. Їхні наслідки є потенційно критичними та вимагають належного планування заходів реагування і профілактики.

4.3. Безпека в надзвичайних ситуаціях

Організація безпеки праці в умовах надзвичайних ситуацій передбачає розробку чітких алгоритмів поведінки працівників, інструкцій та заходів щодо мінімізації ризиків. У межах цього розділу розглянуто ключові аспекти забезпечення захисту персоналу під час пожеж, техногенних аварій та загроз, пов’язаних із воєнними діями.

Одним із основних елементів системи безпеки є наявність доступного укриття, здатного захистити працівників від уламків, ударної хвилі та інших небезпечних факторів. У кожній будівлі має бути визначене приміщення, яке відповідає вимогам ДСНС: відсутність вікон, наявність міцних стін, вентиляції, аварійного освітлення та запасу питної води. Працівники мають бути ознайомлені з маршрутами евакуації та шляхами швидкого доступу до укриття.

Під час повітряної тривоги працівники повинні негайно припинити роботу, вимкнути електрообладнання та, не створюючи паніки, пройти до укриття. Особливо важливо не нехтувати сигналами повітряної тривоги, оскільки час реагування є ключовим фактором збереження життя. Дотримання офіційних рекомендацій, а також наявність в укритті аптечки, засобів першої допомоги, ковдр та запасу води є обов’язковою умовою.

Іншим важливим аспектом є пожежна безпека. Працівники повинні вміти користуватися вогнегасником і знати основні правила поведінки під час пожежі: не відкривати вікна та двері, щоб не сприяти доступу кисню, залишати приміщення якнайшвидше, використовуючи найближчий евакуаційний вихід, та повідомляти відповідні служби. У приміщенні мають бути встановлені датчики диму та вогнегасники у легкодоступних місцях.

У разі техногенних аварій - витоку газу, короткого замикання або затоплення - працівникам необхідно діяти відповідно до інструкцій: знеструмити обладнання, залишити приміщення, повідомити відповідні аварійні служби й не намагатися самостійно усунути небезпеку без належної підготовки.

Окрему увагу слід приділити психологічній готовності працівників до надзвичайних ситуацій. Регулярні тренування та інструктажі допомагають мінімізувати паніку та забезпечують чіткість дій у разі небезпеки. Працівники мають розуміти важливість дотримання інструкцій, адже саме від них залежить збереження життя і здоров’я всього колективу.

Узагальнюючи наведене, можна зробити висновок, що забезпечення безпеки в надзвичайних ситуаціях вимагає комплексного підходу, який об’єднує технічні засоби захисту, організаційні заходи та особисту відповідальність працівників. Знання правил поведінки, наявність укриття, справність електрообладнання та організація евакуаційних шляхів є основою безпечного робочого середовища.

РОЗДІЛ 5.
ВИЗНАЧЕННЯ ЕФЕКТИВНОСТІ

Ефективність розробленої системи масштабування веб-додатку у контейнеризованому середовищі Kubernetes визначається її здатністю забезпечувати стабільну роботу за умов змінного навантаження, зберігати працездатність у пікові періоди та раціонально використовувати доступні ресурси. У ході дослідження було проведено комплексну оцінку архітектури, продуктивності та поведінки системи в залежності від зовнішніх факторів, що дозволило сформувати цілісне уявлення про її функціональні можливості та відповідність поставленим вимогам.

Розроблений веб-додаток, який виконує обробку графічних файлів, є типовим прикладом ресурсоємного сервісу, здатного створювати суттєве навантаження на обчислювальні ресурси. Саме тому він є зручним об’єктом для тестування механізмів горизонтального масштабування. Проведені експерименти підтвердили, що контейнеризація застосунку забезпечує стабільність середовища виконання та дозволяє розгортати ідентичні копії сервісу без додаткових налаштувань. Це значно спрощує процес масштабування та робить систему придатною для динамічного регулювання кількості реплік.

Під час тестування продуктивності було виявлено, що робота лише одного екземпляра веб-додатку є недостатньою для стабільної обробки великої кількості паралельних запитів. У моменти зростання навантаження час обробки запитів поступово збільшувався, що свідчило про необхідність запуску додаткових подів. Завдяки впровадженому механізму автоматичного масштабування Kubernetes успішно визначав перевищення граничного рівня завантаження процесора та створював нові репліки застосунку. Це дозволило розподілити навантаження рівномірно між екземплярами, зменшити затримки у роботі сервісу та запобігти зниженню якості обслуговування користувачів.

Після усунення пікового навантаження система демонструвала зворотну реакцію - кількість реплік скорочувалася до мінімально необхідного значення. Така поведінка є важливою характеристикою ефективної інфраструктури, адже вона дозволяє уникати необґрунтованого витрачання ресурсів. У реальних хмарних середовищах це безпосередньо впливає на фінансові витрати, оскільки користувачі сплачують лише за фактичне використання обчислювальної потужності.

До позитивних результатів дослідження належить також стабільна робота системи під час моделювання різких коливань навантаження. Веб-додаток зберігав працездатність навіть у тих випадках, коли кількість запитів істотно зростала за короткий проміжок часу. Це свідчить про правильність обраної архітектури, зокрема про доцільність використання об’єктів Deployment і Service, а також про ефективність мережевого балансування, яке здійснює рівномірний розподіл запитів між репліками.

Контейнеризація та використання Kubernetes продемонстрували також високі показники відмовостійкості. У разі зупинки одного з подів система автоматично відновлювала його роботу, створюючи новий екземпляр на основі наявного Docker-образу. Такий підхід мінімізує ризики зупинки сервісу та забезпечує безперервність роботи застосунку, що є особливо важливою вимогою для сучасних веб-систем.

Крім технічних аспектів, важливою складовою оцінювання ефективності системи є можливість її подальшої масштабованості та інтеграції у хмарні платформи. Архітектурні рішення, які були застосовані в рамках дипломної роботи, повністю відповідають кращим практикам побудови cloud-native додатків. Завдяки використанню стандартних механізмів Kubernetes система може бути перенесена в будь-яке хмарне середовище - AWS, Azure або Google Cloud - без необхідності суттєвої модифікації структури проєкту. Це підтверджує її універсальність та довготривалу придатність для реальних виробничих умов.

Окремої уваги заслуговує організаційна ефективність розробленої інфраструктури. Автоматизація процесів розгортання, масштабування та відновлення знижує навантаження на IT-персонал і забезпечує більш ефективне використання робочого часу. Завдяки наявності механізмів логування та моніторингу адміністратори можуть відстежувати роботу системи та своєчасно реагувати на відхилення у її функціонуванні, що сприяє підвищенню загальної надійності рішення.

Узагальнюючи результати дослідження, можна зробити висновок, що розроблена система демонструє високий рівень ефективності та повністю відповідає поставленій меті. Вона забезпечує стабільну роботу веб-додатку в умовах змінного навантаження, автоматично адаптується до зовнішніх факторів, ефективно використовує доступні ресурси та має потенціал для подальшого розширення. Така система може бути успішно застосована для реальних комерційних веб-сервісів, що потребують високої продуктивності, надійності та гнучкості.

ВИСНОВКИ

Під час виконання кваліфікаційної роботи було здійснено проектування та розробка системи масштабування веб-додатку на основі технологій контейнеризації та оркестрації, що дозволило комплексно дослідити сучасні підходи до забезпечення високої продуктивності, гнучкості та стійкості програмних систем. У процесі роботи було проаналізовано теоретичні основи веб-масштабування, розглянуто особливості роботи сучасних хмарних платформ і проведено порівняння їхніх інструментів та можливостей. Це надало змогу сформувати уявлення про принципи побудови cloud-native рішень та визначити оптимальні технології для реалізації поставленої задачі.

У ході практичної частини було створено веб-додаток для обробки зображень, який слугував об’єктом для дослідження механізмів масштабування. Застосунок було контейнеризовано за допомогою Docker, що забезпечило незалежність його середовища виконання та можливість швидкого ідентичного розгортання. Подальше керування життєвим циклом додатку здійснювалося засобами Kubernetes, який проявив себе як ефективний інструмент для автоматизації розгортання, балансування навантаження та відновлення після збоїв.

Розроблений Kubernetes-кластер дозволив протестувати роботу системи у різних режимах. У ході експериментів було підтверджено здатність додатку стабільно функціонувати при збільшенні кількості паралельних запитів, а механізм Horizontal Pod Autoscaler продемонстрував ефективність у реагуванні на підвищене навантаження шляхом автоматичного збільшення кількості реплік додатку. Після зменшення навантаження система автоматично поверталася до оптимальної кількості екземплярів, що свідчить про раціональне використання ресурсів. Подібна поведінка є характерною для сучасних динамічних хмарних систем і підтверджує доцільність обраного підходу.

Дослідження також показало високу відмовостійкість системи: при завершенні роботи одного з подів Kubernetes автоматично створював новий, що забезпечувало безперервність роботи веб-додатку. Механізм балансування навантаження рівномірно розподіляв запити між усіма репліками, що сприяло зниженню часу обробки запитів і підтримці стабільної продуктивності навіть у пікові періоди. Усі ці результати підтверджують, що створена інфраструктура відповідає вимогам до сучасних масштабованих веб-систем.

У межах роботи було також приділено увагу питанням охорони праці та безпеки в надзвичайних ситуаціях, що є особливо актуальним в умовах триваючої війни в Україні. Проведений аналіз дозволив окреслити основні фактори ризику, пов’язані з роботою в офісному середовищі, а також визначити правила поведінки працівників під час повітряних тривог і необхідність використання укриттів. Інтеграція цих рекомендацій у робочий процес є важливою складовою створення безпечних умов праці.

Узагальнюючи результати дослідження, можна зробити висновок, що поставлена у дипломній роботі мета була досягнута. Розроблена система продемонструвала високу ефективність у задачах автоматичного масштабування, забезпечила стабільність та продуктивність веб-додатку й підтвердила перспективність застосування технологій Docker та Kubernetes для побудови сучасних високонавантажених програмних систем. Отримані результати можуть бути використані як основа для подальших досліджень у галузі хмарних технологій, а також для практичного впровадження в реальні проєкти, що потребують високої адаптивності та надійності.

СПИСОК ВИКОРИСТАНОЇ ЛІТЕРАТУРИ

1. Neoris. What are IaaS, PaaS, and Saas? URL: https://neoris.com/es/blog/que-es-iaas-paas-y-saas (дата звернення 22.02.2025)
2. Bernard Golden. Kubernetes for Cloud Developers. O'Reilly, 2022.
3. CloudToggle. Horizontal vs Vertical Scaling. URL: https://www.cloudtoggle.com/blog-en/horizontal-vs-vertical-scaling-2/ (дата звернення 23.03.2025)
4. Thomas Erl, Ricardo Puttini, Zaigham Mahmood. Cloud Computing: Concepts, Technology & Architecture. Prentice Hall, 2013.
5. Amazon Web Services. Overview of Amazon Web Services. URL: https://docs.aws.amazon.com/whitepapers/latest/aws-overview/ (дата звернення 25.03.2025)
6. Amazon Web Services. Amazon EC2 Auto Scaling User Guide.URL: https://docs.aws.amazon.com/autoscaling/ (дата звернення 03.04.2025)
7. Microsoft. What is Azure? URL: https://learn.microsoft.com/azure/ (дата звернення 03.04.2025)
8. Microsoft. Azure Kubernetes Service (AKS) Documentation.URL: https://learn.microsoft.com/azure/aks/ (дата звернення 18.04.2025)
9. Google Cloud. Google Cloud Overview. URL: https://cloud.google.com/docs/overview (дата звернення 24.04.2025)
10. Google Cloud. Google Kubernetes Engine (GKE) Documentation. URL: https://cloud.google.com/kubernetes-engine/docs (дата звернення 27.04.2025)
11. Kubernetes Documentation. Kubernetes Components. URL: https://kubernetes.io/docs/concepts/architecture/(дата звернення 05.05.2025)
12. Google Research. The Evolution of Scalable Web Architectures, 2021.

13. Docker Official Documentation. URL: https://www.docker.com/resources/what-container/ (дата звернення 12.05.2025)
14. Kubernetes Documentation. URL: https://kubernetes.io/docs/concepts/architecture/ (дата звернення 04.09.2025)
15. Kelsey Hightower, Brendan Burns, Joe Beda. Kubernetes: Up and Running. 3rd ed. O'Reilly Media, 2021.
16. Brendan Burns. Designing Distributed Systems: Patterns and Paradigms for Scalable, Reliable Services. O'Reilly Media, 2018.
17. AWS Auto Scaling Documentation. URL: https://docs.aws.amazon.com/autoscaling/ (дата звернення 14.09.2025)
18. Amazon Web Services. Target Tracking Scaling Policies for Amazon EC2 Auto Scaling. URL: https://docs.aws.amazon.com/autoscaling/ec2/userguide/ec2-auto-scaling-target-tracking.html (дата звернення 23.09.2025)
19. NGINX Documentation. URL: https://www.nginx.com/resources/glossary/load-balancing/ (дата звернення 27.09.2025)
20. NGINX. HTTP Load Balancing. URL: https://docs.nginx.com/nginx/admin-guide/load-balancer/http-load-balancer/ (дата звернення 12.10.2025)
21. Gene Kim, Jez Humble, Patrick Debois, John Willis. The DevOps Handbook: How to Create World-Class Agility, Reliability, and Security in Technology Organizations. IT Revolution Press, 2016.
22. Jez Humble, David Farley. Continuous Delivery: Reliable Software Releases through Build, Test, and Deployment Automation. Addison-Wesley, 2010.
23. Niall Richard Murphy, Betsy Beyer, Chris Jones, Jennifer Petoff (eds.). Site Reliability Engineering: How Google Runs Production Systems. O'Reilly Media, 2016.

24. Betsy Beyer, Niall Richard Murphy, David K. Rensin, Kent Kawahara, Stephen Thorne. The Site Reliability Workbook: Practical Ways to Implement SRE. O'Reilly Media, 2018.
25. Nicole Forsgren, Jez Humble, Gene Kim. Accelerate: The Science of Lean Software and DevOps. IT Revolution Press, 2018.
26. Cloud Native Computing Foundation (CNCF). Cloud Native Definition v1.0. URL: https://github.com/cncf/toc/blob/main/DEFINITION.md (дата звернення 28.10.2025)
27. Semaphore CI/CD Documentation. URL: https://semaphoreci.com/blog/cicd-pipeline (дата звернення 02.11.2025)
28. Kubernetes. Minikube Documentation. URL: https://minikube.sigs.k8s.io/docs/ (дата звернення 03.11.2025)
29. Docker. Docker Overview. URL: https://docs.docker.com/get-started/overview/ (дата звернення 15.11.2025)
30. Nigel Poulton. Docker Deep Dive. 4th ed. Packt Publishing, 2020.
31. James Turnbull. The Docker Book: Containerization is the New Virtualization. James Turnbull, 2014.

ДОДАТКИ

Додаток А

Фрагмент програмного коду серверної частини веб-додатку «Image Processing Service»

```
// app.js – серверна частина веб-додатку "Image Processing Service"

const express = require('express');
const multer = require('multer');
const sharp = require('sharp');
const path = require('path');
const fs = require('fs');

const app = express();
const PORT = process.env.PORT || 3000;

// Каталог для тимчасового збереження завантажених зображень
const uploadDir = path.join(__dirname, 'uploads');
if (!fs.existsSync(uploadDir)) {
  fs.mkdirSync(uploadDir);
}

// Налаштування зберігання файлів за допомогою multer
const storage = multer.diskStorage({
  destination: (req, file, cb) => {
    cb(null, uploadDir);
  },
  filename: (req, file, cb) => {
    const uniqueName = Date.now() + '-' + file.originalname;
    cb(null, uniqueName);
```

```
    }
});

const upload = multer({ storage });

// Службовий маршрут для перевірки стану сервісу
app.get('/health', (req, res) => {
    res.json({ status: 'ok', timestamp: new Date().toISOString() });
});

// Головний маршрут для обробки зображення (ресайз)
app.post('/resize', upload.single('image'), async (req, res) => {
    try {
        if (!req.file) {
            return res.status(400).json({ error: 'Файл зображення не
завантажено' });
        }

        const width = parseInt(req.body.width) || 800;
        const height = parseInt(req.body.height) || 600;

        const inputPath = req.file.path;
        const outputPath = path.join(uploadDir, 'resized-' + req.file.filename);

        await sharp(inputPath)
            .resize(width, height, { fit: 'inside' })
            .png()
            .toFile(outputPath);

        // Повертаємо користувачу оброблене зображення
```

```
        res.sendFile(outputPath, err => {
            // Після відправки файлів можна видалити тимчасові
            fs.unlink(inputPath, () => {});
            fs.unlink(outputPath, () => {});
        });

    } catch (error) {
        console.error('Помилка обробки зображення:', error);
        res.status(500).json({ error: 'Помилка обробки зображення' });
    }
});

// Обробка некоректних маршрутів
app.use((req, res) => {
    res.status(404).json({ error: 'Маршрут не знайдено' });
});

app.listen(PORT, () => {
    console.log(`Image Processing Service запущено на порту ${PORT}`);
});
```