**Міністерство освіти і науки України**
**Луцький національний технічний університет**
**Факультет комп'ютерних та інформаційних технологій**
**Кафедра комп'ютерних наук**

КВАЛІФІКАЦІЙНА РОБОТА
ЗА СТУПЕНЕМ ВИЩОЇ ОСВІТИ «МАГІСТР»

# РОЗРОБКА ТА ДОСЛІДЖЕННЯ СИСТЕМИ БЕЗПЕРЕРВНОЇ ІНТЕГРАЦІЇ ЗА ДОПОМОГОЮ ЗАСОБІВ КОНТЕЙНЕРИЗАЦІЇ

# DEVELOPMENT AND RESEARCH OF A CONTINUOUS INTEGRATION SYSTEM USING CONTAINERIZATION TOOLS

спеціальність 122 Комп'ютерні науки

освітня програма «Комп'ютерні науки»

Виконав: здобувач вищої освіти
групи КНм-21
Марчук Микола Олександрович
_____
(підпис)
Керівник: к.т.н., доцент
Кошелюк Віктор Андрійович
_____
(підпис)

Кваліфікаційну роботу
допущено до захисту
«___» _____ 2025 р.
Гарант освітньої програми:
к.т.н., доцент
Ліщина Валерій Олександрович
_____
(підпис)

Луцьк – 2025 року

**ЛУЦЬКИЙ НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ**

Факультет *комп'ютерних та інформаційних технологій*
Кафедра *комп'ютерних наук*
Ступінь вищої освіти: *магістр*
Галузь знань: *12 Інформаційні технології*
Спеціальність: *122 Комп'ютерні науки*
Освітня програма: *«Комп'ютерні науки»*

ЗАТВЕРДЖУЮ
Завідувач кафедри
_____Валерій ЛІЩИНА
«14» травня 2025 р.

**ЗАВДАННЯ**
**НА КВАЛІФІКАЦІЙНУ РОБОТУ ЗДОБУВАЧА**
**ДРУГОГО (МАГІСТЕРСЬКОГО) РІВНЯ ВИЩОЇ ОСВІТИ**

Марчук Микола Олександрович
(прізвище, ім'я, по батькові)

1. Тема кваліфікаційної роботи «Розробка та дослідження системи безперервної інтеграції за допомогою засобів контейнеризації»

Керівник к.т.н., доцент Кошелюк Віктор Андрійович

затверджені наказом закладу вищої освіти від «*14*» *травня* *2025* р. № 255/01-02

2. Строк подання здобувачем вищої освіти кваліфікаційної роботи *«05» грудня 2025 р.*

3. Вихідні дані до роботи: демо-застосунок як об'єкт автоматизованого складання й розгортання, репозиторій з вихідним кодом; Docker/Docker Compose; обліковий запис CI GitHub Actions; віддалений реєстр образів (Github Container Registry); ВМ та доменне ім'я для пробного деплою

4. Зміст розрахунково-пояснювальної записки (перелік питань, що потрібно розробити):

*Аналіз літературних джерел за темою кваліфікаційної роботи магістра, виклад проблеми і вибір напрямків дослідження; опис рішення проблеми; методика та проведення експерименту.*

5. Перелік графічного матеріалу: 13 рисунків, 8 таблиць, 4 лістинги

6. Консультанти розділів роботи

| Розділ | Прізвище, ініціали та посада консультанта | Підпис | |
|---|---|---|---|
| | | завдання видав | завдання прийняв |
| *Аналіз проблематики контейнеризації за допомогою CI/CD* | *Кошелюк В. А.* | | |
| *Теоретичне дослідження та практична реалізація контейнеризації з CI/CD* | *Кошелюк В. А.* | | |
| *Експерементальне дослідження результативності контейнеризації з CI/CD* | *Кошелюк В. А.* | | |
| *Показник запозичень тексту* | _____ % | | |
| *Інструментальна перевірка* | *Кошелюк В. А.* | | |
| *Нормоконтроль* | *Сачук В. О.* | | |
| *Гарант ОПП* | *Ліщина В. О.* | | |

7. Дата видачі завдання *«14» травня 2025 р.*

КАЛЕНДАРНИЙ ПЛАН

| № з/п | Назва етапів кваліфікаційної роботи магістра | Строк виконання етапів роботи | Примітка |
|---|---|---|---|
| *1* | *Провести огляд літературних джерел по темі кваліфікаційної роботи* | *до 30.06.2025 р* | |
| *2* | *Провести аналіз загальної проблеми і вибір напрямків дослідження* | *до 01.09.2025 р.* | |
| *3* | *Розробити функціональну схему роботи програмного продукту* | *до 01.10.2025 р* | |
| *4* | *Описати засоби розробки об'єкта проектування* | *до 15.10.2025 р.* | |
| *5* | *Практична реалізація об'єкта проектування* | *до 10.11.2025 р.* | |
| *6* | *Провести експериментальне дослідження результативності предмету дослідження* | *до 25.11.2025 р.* | |
| *7* | *Здача чистового варіанту кваліфікаційної роботи магістра на кафедру* | *до 05.12.2025 р.* | |

Здобувач вищої освіти _____ Микола МАРЧУК

Керівник роботи _____ Віктор КОШЕЛЮК

# АНОТАЦІЯ

Марчук М. О. Розробка та дослідження системи безперервної інтеграції за допомогою засобів контейнеризації. Рукопис.

Кваліфікаційна робота магістра ОП «Комп'ютерні науки» спеціальності 122 «Комп'ютерні науки». Луцький національний технічний університет. Луцьк, 2025.

У роботі досліджено сучасні підходи до автоматизації розроблення програмного забезпечення із застосуванням контейнеризації та систем безперервної інтеграції і доставки (CI/CD). У першому розділі виконано огляд літератури та рішень у сфері DevOps і визначено роль Docker у забезпеченні відтворюваності та стабільності середовищ. У другому розділі обґрунтовано вибір технологій і описано реалізацію серверної й клієнтської частин застосунку та інфраструктури розгортання. У третьому розділі наведено архітектурну модель CI/CD. У четвертому розділі реалізовано конвеєр CI/CD на основі GitHub Actions з інтеграцією GHCR, засобів сканування вразливостей та автоматизованого деплою. Проведено експериментальну оцінку часу збірки, стабільності етапів і можливостей відновлення. Результати підтверджують практичну цінність створеної CI/CD-системи.

Ключові слова: DevOps, CI/CD, контейнеризація, Docker, GitHub Actions, GHCR, автоматизація розроблення, інфраструктура розгортання.

# ABSTRACT

Mykola Marchuk. Development and Research of a Continuous Integration System Using Containerization Tools. Manuscript.

The master’s qualification work of the educational program «Computer Science», specialty 122 «Computer Science». Lutsk National Technical University. Lutsk, 2025.

The work investigates modern approaches to automating software development through containerization and continuous integration and delivery (CI/CD) systems. The first chapter reviews scientific and technical sources in the DevOps field and highlights the role of Docker in ensuring environment reproducibility and system stability. The second chapter substantiates the choice of technologies and presents the implementation of the server-side and client-side parts of the demonstration application, as well as the deployment infrastructure. The third chapter introduces the architectural model of the CI/CD process. The fourth chapter implements a CI/CD pipeline based on GitHub Actions with GHCR integration, security scanning tools, and automated deployment. An experimental evaluation assessed build time, pipeline stability, and rollback capabilities. The results confirm the practical value of the developed CI/CD system for automating software delivery, improving release stability, and ensuring reproducibility in containerized environments.

Key words: DevOps, CI/CD, containerization, Docker, GitHub Actions, GHCR, deployment infrastructure, automation.

# ЗМІСТ

# ВСТУП

Актуальність теми зумовлена швидкими змінами у сфері розроблення програмного забезпечення та переходом індустрії до повної автоматизації процесів складання, тестування й розгортання. У сучасних прикладних системах особливо важливо забезпечити стабільність середовищ, відтворюваність збірок та контроль якості на всіх етапах життєвого циклу. Ручне налаштування інфраструктури, різні версії залежностей та поширені проблеми типу «працює у мене – не працює на сервері» призводять до нестабільних релізів, збільшення витрат часу та помилок, пов’язаних із людським фактором. У цих умовах контейнеризація та автоматизовані процеси безперервної інтеграції й доставки (CI/CD) стають ключовими складовими сучасної інженерії програмного забезпечення.

Об’єктом дослідження є процеси безперервної інтеграції та доставки програмного забезпечення у контейнеризованих середовищах.

Предметом дослідження виступають методи, підходи та інструменти побудови системи CI/CD із використанням технологій контейнеризації, зокрема Docker, а також механізми автоматизованої збірки, тестування, перевірки безпеки та розгортання програмних застосунків.

Метою дослідження є розроблення та експериментальна оцінка системи безперервної інтеграції й доставки для контейнеризованого застосунку, яка забезпечує автоматизовану збірку, тестування, перевірку вразливостей, публікацію образів у реєстр та розгортання на сервер із забезпеченням відтворюваності й стабільності процесів.

Для досягнення поставленої мети у роботі необхідно вирішити такі завдання дослідження:

− провести аналіз сучасних підходів, технологій і тенденцій у сфері DevOps, контейнеризації та CI/CD;

− обґрунтувати доцільність використання контейнеризації як основи стандартизації та відтворюваності середовищ;

− розробити архітектуру CI/CD-конвеєра для контейнеризованого застосунку на базі GitHub Actions та Docker;

− формалізувати процеси автоматизованої збірки, тестування, сканування безпеки та розгортання;

− реалізувати робочий конвеєр CI/CD та виконати його експериментальну оцінку;

− проаналізувати результати експериментів за показниками часу збірки, стабільності виконання етапів і тривалості розгортання;

− сформувати рекомендації щодо підвищення ефективності та надійності побудованої системи.

Методи дослідження включають аналіз наукових і технічних джерел у сфері DevOps та контейнеризації; методи системного, структурного та порівняльного аналізу для формування архітектури CI/CD; моделювання потоків даних і середовищ виконання; експериментальні методи оцінювання ефективності роботи конвеєра на основі кількісних показників.

Наукова новизна полягає у формуванні моделі системи CI/CD, що поєднує технології контейнеризації Docker із засобами автоматизації GitHub Actions, забезпечуючи підвищену відтворюваність, ізоляцію середовищ і масштабованість процесів доставки програмного забезпечення.

Практичне значення роботи полягає у можливості застосування розробленої системи CI/CD у реальних умовах розроблення програмного забезпечення для підвищення стабільності релізів, скорочення часу інтеграції змін, зменшення кількості помилок у робочому середовищі та оптимізації процесів постачання. Побудована система може бути використана як навчальний та експериментальний стенд для вивчення сучасних DevOps-підходів.

Основні положення та проміжні результати цієї кваліфікаційної роботи були апробовані у науковій статті з теми «Інтеграція AI/ML у DevOps/CI/CD процеси: дослідження ефективності та ризики», опублікованій у Студентському науковому віснику Луцького національного технічного університету. У статті розглянуто перспективи застосування алгоритмів машинного навчання для

оптимізації етапів CI/CD, зокрема в частині прогнозування збоїв, автоматичного аналізу журналів та пріоритизації тестів. Отримані під час публікації зауваження та рекомендації були враховані при подальшому вдосконаленні моделі CI/CD-системи, реалізованої в межах кваліфікаційної роботи.

# РОЗДІЛ 1
# АНАЛІЗ ПРОБЛЕМАТИКИ КОНТЕЙНЕРИЗАЦІЇ ЗА ДОПОМОГОЮ CI/CD

## 1.1 Огляд і аналіз предметної області безперервної інтеграції та контейнеризації

Розвиток інформаційних технологій у XXI столітті супроводжується швидким зростанням обсягів програмних систем, збільшенням вимог до їхньої стабільності, безпеки та безперервності функціонування. У цих умовах значного поширення набули методології, що забезпечують гнучкість, автоматизацію та контрольованість процесів розроблення програмного забезпечення. Однією з таких методологій є DevOps – інтеграційна концепція, що поєднує розробку, тестування, впровадження та підтримку програмних продуктів у єдиний безперервний цикл. Центральне місце в ній займають процеси безперервної інтеграції (Continuous Integration – CI) та безперервного розгортання (Continuous Deployment – CD) [5], які дозволяють досягти високого рівня автоматизації життєвого циклу програмного забезпечення.

Витоки практики CI/CD сягають кінця 1990-х років, коли в межах методології екстремального програмування (Extreme Programming, XP) було запропоновано концепцію регулярного об’єднання змін у спільну кодову базу та автоматичного виконання тестів після кожного коміту. Згодом ця ідея переросла в окремий напрямок – безперервну інтеграцію, реалізовану у таких інструментах, як CruiseControl, Jenkins, Bamboo та TeamCity [6]. У подальшому розвиток індустрії призвів до появи безперервного розгортання (CD), коли кожна зміна, що пройшла тестування, автоматично доставляється на сервер без участі розробника. Таким чином, CI/CD перетворилося на ключовий елемент сучасних DevOps-процесів, який забезпечує швидке, контрольоване та безпечне оновлення систем.

Парадигма DevOps базується на моделі CALMS (Culture, Automation, Lean, Measurement, Sharing) [7], що визначає п’ять основних складових ефективної

організації процесів. Культура співпраці (Culture) передбачає спільну відповідальність розробників і операторів за якість продукту. Автоматизація (Automation) усуває рутинні дії, підвищує швидкість процесів і знижує ризик людських помилок. Принцип Lean орієнтується на мінімізацію втрат часу й ресурсів, вимірювання (Measurement) дозволяє контролювати показники ефективності, а спільний обмін знаннями (Sharing) сприяє постійному вдосконаленню команди. Дослідження DORA (DevOps Research and Assessment), проведені під егідою Google [1], показують, що організації, які впроваджують DevOps-практики, досягають у середньому на 30-50 % вищих показників продуктивності.

Наукові публікації [2-4] підкреслюють, що ефективність CI/CD безпосередньо пов’язана зі зменшенням часу між написанням коду та його впровадженням у виробництво. Це дозволяє скоротити кількість дефектів, підвищити стабільність програмного забезпечення та швидше впроваджувати інновації. Для великих проєктів безперервна інтеграція стала не лише технічною, а й стратегічною необхідністю: за оцінками дослідників, компанії, які застосовують CI/CD, зменшують середній час виходу оновлень у 2-3 рази.

Важливим етапом у розвитку CI/CD стало впровадження контейнеризації, що забезпечила ізоляцію середовищ, стандартизацію процесів і відтворюваність результатів. Технологія Docker, представлена у 2013 році, стала революційною у сфері програмної інженерії. Вона дозволяє створювати образи застосунків із визначеними залежностями, системними бібліотеками та конфігураціями, що гарантує однакову поведінку програмного продукту в усіх середовищах. Саме контейнеризація стала фундаментом сучасних CI/CD-практик, оскільки дала змогу відокремити розробку від інфраструктури, зменшити ризики конфліктів середовищ і підвищити безпеку розгортань.

До впровадження контейнеризації галузь стикалася з низкою системних проблем, які значно ускладнювали впровадження ефективних CI/CD-процесів. Передусім йдеться про відсутність відтворюваності збірок: одна й та сама система могла працювати по-різному на різних машинах через зміну

залежностей, конфігурацій або системних бібліотек. Дрейф середовищ – поступове накопичення несинхронних версій компонентів на серверах – призводив до неконтрольованих збоїв і унеможливлював стандартизований деплой. Конфлікти залежностей у мовах із динамічною екосистемою (JavaScript, Python) формували додаткові ризики для стабільності. Крім того, відсутність уніфікованого способу пакування застосунків значно ускладнювала масштабування, а ручне налаштування середовищ збільшувало кількість помилок і робило процеси непередбачуваними.

У сучасних умовах системи CI/CD (рис. 1.1) ґрунтуються на поєднанні кількох технологічних принципів: декларативності (pipeline as code), автоматизації тестування, перевірок безпеки (DevSecOps), а також моніторингу й аналітики результатів. За таких умов значно скорочується тривалість життєвого циклу розроблення, а процеси стають передбачуваними. Сучасні дослідники зазначають, що ключем до ефективності CI/CD є стандартизація, репродуктивність збірок і централізований контроль якості.

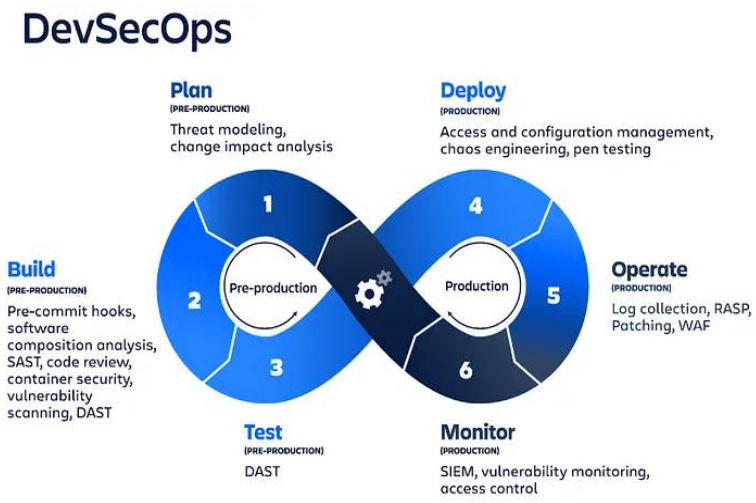


Рисунок 1.1 – Типова схема процесу CI/CD [8]

У науковому середовищі CI/CD дедалі частіше використовується як платформа для експериментів з автоматизації, тестування та моделювання процесів. У дослідницьких лабораторіях системи безперервної інтеграції допомагають створювати відтворювані обчислювальні експерименти, що мають

велике значення для верифікації результатів і забезпечення наукової достовірності. У промислових масштабах CI/CD дозволяє досягти високої швидкості доставки оновлень, а у навчальних – використовується як практичний інструмент вивчення програмної інженерії та автоматизації.

Отже, аналіз предметної області свідчить, що розвиток CI/CD став закономірною відповіддю на зростання складності програмних систем та потребу у швидкому, контрольованому й безпечному випуску оновлень. Проте до появи контейнеризації впровадження CI/CD було суттєво обмежене через дрейф середовищ, конфлікти залежностей, нерепродуктивність збірок і складність масштабування. Контейнеризація усунула ці базові проблеми, створивши технічне підґрунтя для ефективних, стабільних і відтворюваних процесів CI/CD. Це обґрунтовує доцільність її використання та визначає напрями подальшого дослідження у межах роботи.

## 1.2 Огляд і аналіз методів та засобів розробки CI/CD та контейнеризації

У попередньому підрозділі було окреслено загальні засади побудови систем безперервної інтеграції та роль контейнеризації у забезпеченні стабільності, безпеки й відтворюваності процесів розроблення. Наступним кроком є аналіз методів і засобів, які дозволяють реалізувати CI/CD у практичному середовищі. Для досягнення високого рівня автоматизації розроблення необхідно комплексно поєднати підходи до збирання, тестування, сканування безпеки, управління артефактами та моніторингу. Особливу увагу слід приділити використанню платформ GitHub і Docker, що забезпечують гнучку інтеграцію та масштабованість.

Перед впровадженням сучасних підходів CI/CD розробники стикалися з низкою технічних обмежень, що ускладнювали побудову стабільного й передбачуваного процесу доставки програмного забезпечення. До найпоширеніших належали нерепродуктивність збірок, коли різні сервери або

робочі станції формували різні результати через відмінності у версіях бібліотек, системних залежностях чи конфігураціях; поступовий дрейф середовищ, що призводив до накопичення прихованих розбіжностей між тестовими, staging та production-середовищами; конфлікти залежностей у проєктах зі складною екосистемою; а також складність масштабування й перенесення систем між платформами через відсутність уніфікованого способу пакування застосунку. Ці фактори унеможливлювали створення надійної автоматизації та вимагали ручних операцій, що збільшувало ризики помилок і значно уповільнювало процес розроблення.

Основним принципом побудови сучасних CI/CD-конвеєрів є декларативність. Підхід pipeline as code передбачає опис процесу безперервної інтеграції у вигляді конфігураційного файлу, який зберігається в репозиторії разом із вихідним кодом. Це забезпечує відтворюваність збірок, контроль версій і спрощує аудит. На практиці цей принцип реалізується через YAML-скрипти, які визначають послідовність етапів – збірку, тестування, перевірку безпеки, створення артефактів та розгортання. Впровадження такого підходу в межах GitHub Actions дозволяє створювати динамічні, легко масштабовані процеси, що адаптуються до конкретних вимог проєкту.

Розв’язання зазначених проблем стало можливим завдяки використанню спеціалізованих комп’ютерних засобів, які забезпечують контрольованість середовищ, ізоляцію залежностей та автоматизацію всіх етапів життєвого циклу програмного забезпечення. Найважливішу роль відіграє контейнеризація, яка гарантує однакову поведінку застосунку на різних серверах і усуває проблему дрейфу середовищ. Декларативні конвеєри у форматі pipeline as code дозволяють формалізувати й автоматизувати процеси збірки, тестування та розгортання таким чином, що їх виконання стає передбачуваним та відтворюваним незалежно від обраної інфраструктури. Додатково важливим компонентом є інтеграція інструментів DevSecOps, які автоматично аналізують код, залежності та контейнерні образи на наявність вразливостей, тим самим забезпечуючи необхідний рівень якості та безпеки в умовах швидкої доставки оновлень.

Одним із ключових методів підвищення ефективності CI/CD є використання багатостадійних збірок у Docker. Завдяки цьому процес збирання контейнерних образів розділяється на кілька етапів: підготовку середовища, компіляцію, тестування, оптимізацію й публікацію фінального образу. Такий підхід дозволяє значно зменшити розмір кінцевого образу, прискорити повторні збірки за рахунок кешування шарів і підвищити безпеку шляхом ізоляції залежностей. У поєднанні з GitHub Actions цей метод формує стабільну основу для реалізації високопродуктивних CI/CD-конвеєрів.

Крім збірки й тестування, важливу роль відіграє автоматизоване сканування безпеки. Сучасні системи CI/CD інтегрують у свої робочі процеси інструменти перевірки вразливостей, такі як Trivy [9], Snyk і Grype. Вони аналізують як вихідний код, так і залежності та базові Docker-образи. Це є частиною підходу DevSecOps, який поєднує безпеку з процесами розроблення та впровадження. У рамках цього підходу безпека не розглядається як окремий етап, а є невід’ємною частиною всього життєвого циклу програмного продукту.

Ще одним важливим компонентом є система управління артефактами. GitHub Actions забезпечує можливість зберігати результати збірки безпосередньо у репозиторії або передавати їх до зовнішніх реєстрів, таких як Docker Hub або GitHub Packages. Для маркування версій застосовується семантичне версіонування (Semantic Versioning) [15], що передбачає трикомпонентну схему MAJOR.MINOR.PATCH. Такий підхід спрощує керування змінами, дозволяє легко відстежувати історію релізів і забезпечує прозорість процесів. Крім того, автоматичне підписання образів і перевірка їх цілісності гарантують достовірність артефактів.

Для моніторингу процесів CI/CD використовуються підходи спостережуваності (observability), які охоплюють збір логів, метрик і трасування. У контексті GitHub Actions такі механізми реалізуються через системи журналювання, сповіщення та інтеграцію з платформами аналітики. Це дозволяє відстежувати ефективність кожного етапу конвеєра, виявляти потенційні вузькі місця й оперативно реагувати на проблеми. Таким чином, спостережуваність

стає не лише технічною функцією, а й важливим інструментом управління якістю процесу.

Для побудови системи безперервної інтеграції можна використовувати різні платформи. Серед найпоширеніших – GitHub Actions, GitLab CI/CD, Jenkins і CircleCI. Кожна з них має свої переваги та особливості, що визначаються цільовим середовищем, масштабом проєкту та вимогами до конфігурації. Порівняння основних характеристик наведено в таблиці 1.1.

Таблиця 1.1 – Порівняльна характеристика платформ CI/CD

| Інструмент | Основні переваги | Обмеження | Типове застосування |
|---|---|---|---|
| GitHub Actions | Глибока інтеграція з GitHub, просте керування секретами, підтримка Docker, велика бібліотека дій | Обмеження безкоштовних хвилин у публічних репозиторіях | Open-source, дослідницькі та навчальні проєкти |
| GitLab CI/CD | Вбудований контейнерний реєстр, гнучке конфігурування, потужна система артефактів | Потребує розгортання або власних серверів | Корпоративні команди, комерційні проєкти |
| Jenkins | Повна автономність, підтримка тисяч плагінів, розширюваність | Складність налаштування, потреба в адмініструванні | Великі підприємства, власна інфраструктура |
| CircleCI | Хмарна інфраструктура, паралельне виконання тестів, швидке масштабування | Менша гнучкість у кастомізації середовища | SaaS-проєкти, стартапи |

Як видно з таблиці, Порівняльний аналіз наявних рішень для побудови CI/CD свідчить, що хоча кожна з платформ має свої сильні сторони, їхня придатність залежить від вимог до інфраструктури, рівня автономності, обсягу

адміністрування та необхідності інтеграції з контейнерними технологіями. GitLab CI/CD вирізняється потужністю, але потребує власних серверів; Jenkins надає максимальну гнучкість, проте вимагає суттєвих витрат на розгортання та підтримку; CircleCI забезпечує швидке масштабування, але обмежений у кастомізації середовищ. На цьому фоні GitHub Actions демонструє оптимальний баланс простоти використання, гнучкості налаштувань та глибокої інтеграції з Docker і GitHub-екосистемою, що робить його найраціональнішим вибором для реалізації системи безперервної інтеграції у даному дослідженні. Архітектура CI/CD-конвеєра на базі GitHub Actions і Docker представлена на рисунку 1.2.

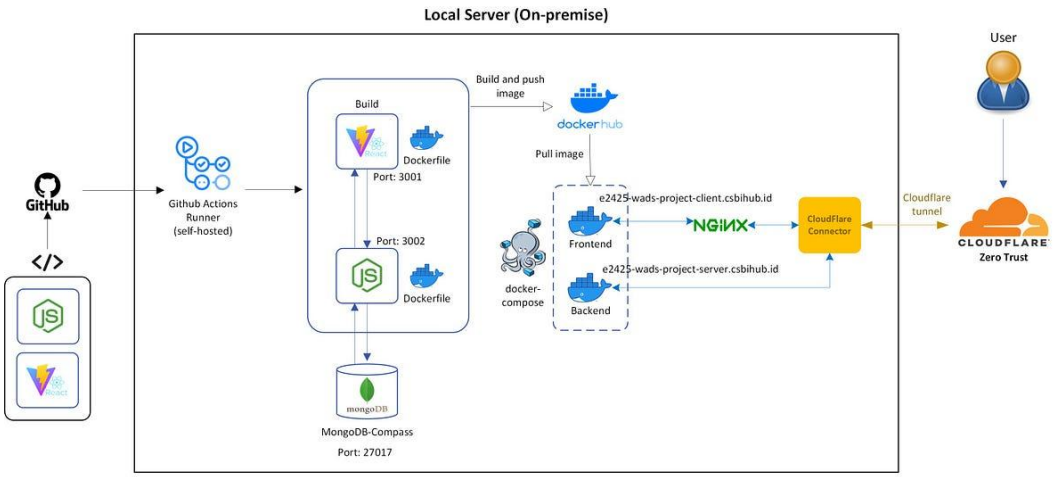


Рисунок 1.2 – Узагальнена архітектура CI/CD-конвеєра GitHub Actions із використанням Docker [10]

Типова архітектура включає джерело коду (GitHub Repository), тригер подій (push або pull request), систему виконання робочих процесів (GitHub Actions Runner), середовище збірки (Docker Container) [11], сховище артефактів (GitHub Packages або Docker Hub) та систему доставки (Deployment Environment). Зв’язки між цими компонентами забезпечують послідовність етапів, починаючи від отримання коду до його автоматичного розгортання. Кожен крок є ізольованим, що гарантує стабільність і відтворюваність процесів.

Використання GitHub Actions для побудови CI/CD-системи має ще одну перевагу – можливість застосування самохостингових раннерів (self-hosted

runners), які забезпечують контроль над обчислювальними ресурсами, безпекою та мережею. Це особливо важливо для наукових і дослідницьких середовищ, де необхідно забезпечити прозорість експериментів і повторюваність результатів. У таких випадках CI/CD-конвеєр може бути не лише інженерним інструментом, а й частиною наукової методології.

Згідно зі звітом State of DevOps Report 2023 [1], компанії, які активно застосовують CI/CD у поєднанні з контейнеризацією, досягають у середньому на 45 % швидшого часу розгортання та на 60 % менше інцидентів у виробничих середовищах. Це підтверджує ефективність використання GitHub і Docker як базових елементів для побудови систем безперервної інтеграції. Крім того, така комбінація зменшує залежність від сторонніх інструментів і спрощує навчання нових учасників команди.

Проведений аналіз показує, що ключові проблеми, які виникали в традиційних підходах до розроблення – зокрема дрейф середовищ, конфлікти залежностей, нестабільність збірок і складність масштабування – можуть бути усунені лише за умови поєднання контейнеризації, декларативного опису конвеєрів та автоматизованого контролю безпеки. Контейнеризація забезпечує відтворюваність та однаковість середовищ, pipeline as code дозволяє формалізувати процеси доставки, а інструменти DevSecOps гарантують перевірку якості на кожному етапі. У сукупності ці засоби формують технологічну основу сучасних CI/CD-рішень, а обґрунтований вибір GitHub Actions і Docker як головних компонентів у межах роботи визначає подальню архітектуру та практичну реалізацію системи безперервної інтеграції. Аналіз методів і засобів розробки показує, що GitHub, GitHub Actions [14] і Docker формують повноцінну екосистему для створення, тестування та доставки програмних продуктів. Поєднання цих технологій дозволяє реалізувати гнучкі, безпечні та відтворювані процеси CI/CD, що відповідають сучасним вимогам до інженерії програмного забезпечення. Отримані результати аналізу слугують теоретичним підґрунтям для подальшого проєктування системи безперервної інтеграції, яке буде розглянуто в наступному розділі.

## 1.3 Постановка завдання на контейнеризацію за допомогою CI/CD

Розвиток інформаційних технологій, зростання складності програмних систем та необхідність швидкої реакції на зміни ринку зумовлюють потребу в ефективних методах автоматизації життєвого циклу програмного забезпечення. На основі аналізу, проведеного у попередніх підрозділах, можна зробити висновок, що сучасна інженерія програмного забезпечення орієнтується на впровадження принципів DevOps і систем безперервної інтеграції (CI/CD), які забезпечують сталість, передбачуваність і контрольованість процесів розроблення. Особливої ваги набуває використання технологій контейнеризації, що дозволяють створювати стандартизовані середовища виконання програм, ізолювати залежності та забезпечувати відтворюваність результатів.

У сучасних умовах контейнеризація виступає не лише технічним інструментом, а й концептуальним підходом до побудови масштабованих, стабільних і безпечних інфраструктур. У поєднанні з DevOps-практиками вона створює основу для реалізації гнучких процесів безперервної інтеграції, які мінімізують людський фактор і скорочують час між розробленням і впровадженням. Саме тому дослідження процесів CI/CD у контексті використання засобів контейнеризації має високу актуальність і практичну значущість. Обраний стек технологій – GitHub, GitHub Actions і Docker – забезпечує необхідний баланс між простотою, масштабованістю та універсальністю, що робить його оптимальним для реалізації системи безперервної інтеграції в межах кваліфікаційного дослідження.

Побудова системи безперервної інтеграції передбачає виконання комплексу завдань, спрямованих на створення відтворюваної, безпечної та ефективної інфраструктури автоматизації процесів розроблення. У межах дослідження передбачається реалізувати такі основні завдання:

− здійснити аналітичний огляд сучасних підходів і технологій у сфері DevOps та CI/CD, визначивши їх сильні сторони й обмеження;

− обґрунтувати доцільність використання контейнеризації як базового механізму відтворюваності й ізоляції середовищ;

− розробити архітектурну модель CI/CD-конвеєра із застосуванням GitHub, GitHub Actions і Docker, що забезпечить автоматизовану збірку, тестування та розгортання програмного продукту;

− створити робочий процес GitHub Actions із використанням декларативного підходу pipeline as code для забезпечення прозорості та контрольованості виконання процесів;

− інтегрувати багатостадійні збірки Docker з етапами тестування, сканування безпеки та публікації артефактів у сховищі;

− налаштувати систему керування версіями та артефактами, реалізувати підписування й перевірку цілісності контейнерних образів;

− провести експериментальні дослідження ефективності роботи CI/CD-системи за показниками швидкості збірки, стабільності розгортань і відтворюваності результатів;

− здійснити порівняльний аналіз отриманих результатів і сформулювати практичні рекомендації щодо оптимізації системи безперервної інтеграції на основі засобів контейнеризації.

Кожне із зазначених завдань має власну наукову та практичну цінність. Так, аналітичний огляд дозволяє окреслити стан сучасних досліджень у сфері DevOps і визначити науковий контекст роботи. Розроблення архітектури CI/CD-конвеєра спрямоване на створення універсальної моделі, яку можна адаптувати для різних типів застосунків. Етапи автоматизації збірки, тестування та сканування безпеки сприяють підвищенню якості коду й стабільності продукту. Проведення експериментів дозволить кількісно оцінити ефективність запропонованого підходу, а розроблені рекомендації забезпечать можливість практичного застосування результатів у навчальних і промислових середовищах.

Очікується, що реалізація поставлених завдань сприятиме підвищенню рівня надійності й безпеки програмних продуктів, оптимізації процесів розроблення та скороченню часу випуску оновлень. Крім того, розроблена

система може бути використана як базова платформа для вивчення технологій DevOps у навчальному процесі, а також як експериментальна інфраструктура для проведення наукових досліджень у галузі програмної інженерії.

## Висновки до розділу 1

У першому розділі проведено системний аналіз сучасного стану проблематики безперервної інтеграції та контейнеризації. На основі огляду джерел встановлено, що саме поєднання CI/CD із контейнерними технологіями усуває ключові недоліки традиційних підходів – дрейф середовищ, нерепродуктивність збірок, конфлікти залежностей та складність масштабування. Проаналізовано методи й інструменти розроблення сучасних CI/CD-систем, визначено їх переваги та обмеження, а також обґрунтовано вибір GitHub Actions і Docker як оптимальної технологічної основи для побудови конвеєра.

Сформульовано завдання дослідження, що логічно випливають із виявлених проблем та актуальних потреб індустрії. Проведений аналіз створює теоретичне підґрунтя для подальшої розробки архітектури системи безперервної інтеграції та її практичної реалізації у наступному розділі.

# РОЗДІЛ 2
# ТЕОРЕТИЧНЕ ДОСЛІДЖЕННЯ ТА ПРАКТИЧНА РЕАЛІЗАЦІЯ КОНТЕЙНЕРИЗАЦІЇ ЗА ДОПОМОГОЮ CI/CD

## 2.1 Обґрунтування вибору шляхів, технологій (алгоритмів) і засобів вирішення поставленого завдання

Побудова системи безперервної інтеграції для контейнеризованого застосунку потребує формалізації архітектури, вибору відповідних технологій, а також визначення логічної структури взаємодії між компонентами. У контексті цього дослідження важливим є створення узагальненої моделі програмного оточення, у межах якого відбувається збірка, тестування, перевірка безпеки та розгортання застосунку. Тому обґрунтування вибору стеку, підходів і засобів повинно спиратися не лише на практичні аспекти, а й на теоретичні принципи модульності, відтворюваності та ізоляції середовищ.

Серверна частина застосунку реалізована мовою Go з використанням фреймворку Gin [18]. Архітектурна модель Go відповідає вимогам CI/CD-проєкту, оскільки компіляція в статичний двійковий файл усуває залежності від середовища виконання, а отже, підвищує відтворюваність контейнеризованих збірок. У контексті теоретичної моделі системи Go розглядається як сервіс із чітко визначеним інтерфейсом взаємодії: REST-ендпоїнтами, що обробляють запити автентифікації, авторизації та запити до бази SQLite. Використання бібліотек для JWT-автентифікації, хешування паролів, обробки CORS і роботи з базою даних визначає зрозумілу й компактну модель сервісу, в якій усі залежності можуть бути інкапсульовані всередині контейнера. Це важливо для CI/CD, де кожна збірка повинна бути детермінованою, а поведінка сервісу – відтворюваною в різних середовищах.

Клієнтська частина, побудована на React [16] з використанням Vite [17], формує окремий логічний модуль, відповідальний за відображення інтерфейсу користувача та комунікацію з API-шару. З теоретичної точки зору React виступає як декларативна модель подання, а Vite – як високопродуктивний інструмент

побудови артефактів, що забезпечує малий час збірки та створення статично оптимізованих ресурсів. У системі CI/CD цей модуль розглядається як окремий етап пайплайна з визначеними входами (вихідний код) і виходами (збірка статичних файлів), які можуть бути розміщені в контейнері або завантажені на хмарну платформу.

Важливим елементом теоретичної моделі є застосування Docker Compose як засобу моделювання локального оточення. Compose дозволяє розглядати систему як множину взаємопов'язаних сервісів – бекенд, фронтенд та базу даних – що працюють у ізольованих контейнерах, але комунікують через єдину мережу. Це дозволяє описати архітектуру системи як граф взаємодій, у якому кожен вузол представляє окремий контейнер, а ребра – мережеві зв'язки та залежності. Така модель (рис. 2.1) відповідає фундаментальному принципу CI/CD: середовище розроблення має бути максимально наближене до середовища розгортання, а ізоляція контейнерів забезпечує повторюваність усіх операцій.

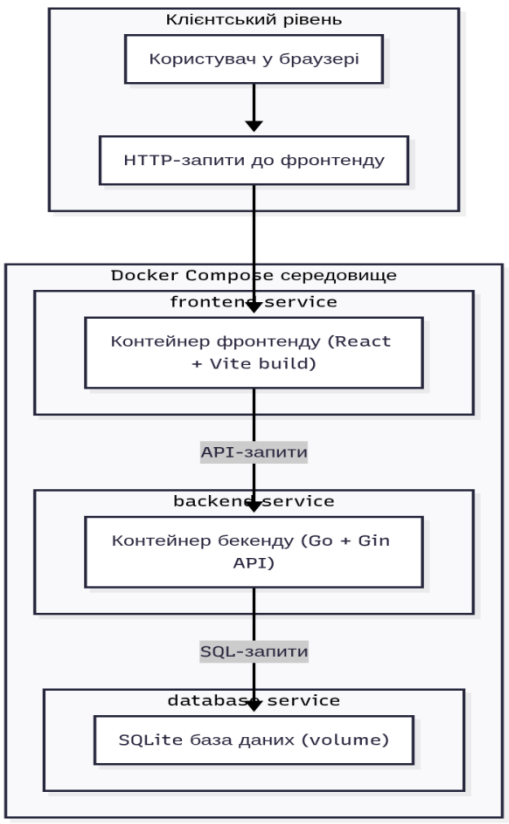


Рисунок 2.1 – Модель потоків даних у системі CI/CD

Центральним компонентом автоматизації є GitHub Actions, який забезпечує декларативний опис конвеєрів та інтеграцію подій репозиторію зі сценаріями виконання. У теоретичній моделі CI/CD GitHub Actions виступає як механізм управління процесами, що перетворює зміни у вихідному коді на автоматичні дії: збірку, тестування, сканування вразливостей, створення контейнерних образів і розгортання. Його інтеграція з GitHub Container Registry або Docker Hub дозволяє розглядати артефакти збірки як структуровані об’єкти, які зберігаються в централізованому реєстрі з гарантованою цілісністю та версіонуванням.

З метою теоретичного опису системи розроблено модель потоку даних (рис. 2.2), яка визначає логічні зв’язки між компонентами. На першому етапі користувач виконує коміт у репозиторій, що генерує подію для GitHub Actions. Далі запускається конвеєр, який у межах окремих ізольованих стадій трансформує вихідний код у двійкові артефакти, статичні ресурси та контейнерні образи.

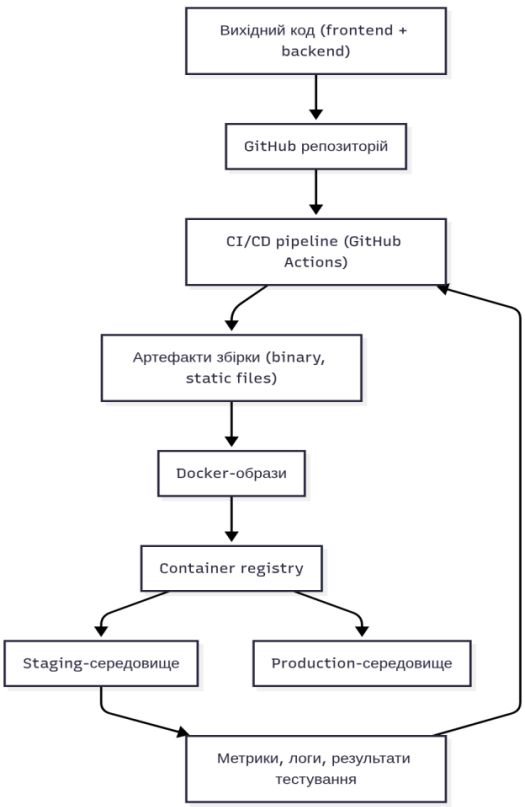


Рисунок 2.1 – Модель потоків даних у системі CI/CD

Важливу роль у теоретичній моделі відіграють гіпотези дослідження, які визначають передбачувану поведінку системи. Зокрема передбачається, що використання контейнеризації забезпечить стабільну відтворюваність збірок незалежно від середовища; що багатостадійна компіляція Go-застосунку зменшить розмір образу та час розгортання; що GitHub Actions дозволить гарантувати сталу роботу конвеєра при зміні вихідного коду; і що інтеграція інструментів аналізу безпеки знизить кількість вразливостей у опублікованих контейнерних образах. Перевірка цих гіпотез у подальших розділах роботи дозволить оцінити ефективність запропонованої архітектури.

Окремої уваги потребує обґрунтування вибору стеку технологій. Використання мови Go є доцільним завдяки її продуктивності та детермінованості процесів компіляції, що значно спрощує створення високоефективних CI/CD-конвеєрів. Фреймворк Gin забезпечує мінімалістичну структуру API та низькі накладні витрати, що відповідає підходам до побудови мікросервісів. React із Vite мінімізує час збірки та формує структуровані статичні ресурси, які можна легко контейнеризувати. Docker Compose дозволяє формалізувати структуру системи як множину сервісів, а GitHub Actions забезпечує відтворюваність і гнучкість конвеєрів у хмарному середовищі.

Таким чином, теоретичне дослідження демонструє, що поєднання Go, Gin, React, Vite, Docker Compose і GitHub Actions утворює цілісну модель екосистеми, яка відповідає сучасним принципам безперервної інтеграції та доставки. Формалізація архітектури, гіпотези дослідження, опис потоків даних і обґрунтування вибору технологій створюють методологічний фундамент для практичної реалізації системи CI/CD.

## 2.2 Практична реалізація контейнеризації за допомогою CI/CD

Серверна частина розробленої системи реалізована мовою Go з використанням веб-фреймворку Gin, що забезпечує поєднання високої продуктивності, низьких накладних витрат і зручної маршрутизації HTTP-

запитів. Логіка бекенд-модуля побудована таким чином, щоб забезпечити мінімалістичний, проте завершений приклад серверного застосунку, який може слугувати демонстраційною основою для побудови повного CI/CD-конвеєра. Архітектура сервера передбачає оброблення запитів автентифікації, авторизації та отримання тестових даних, що дозволяє відтворити типовий робочий процес веб-сервісу в контейнеризованому середовищі.

При запуску сервер виконує ініціалізацію бази даних SQLite [19], яка обрана як легковагове та просте рішення для зберігання інформації про користувачів. Формується єдина таблиця users, що містить унікальний ідентифікатор, ім’я користувача та хешований пароль. Вибір саме SQLite є доцільним у контексті демонстраційного проєкту, оскільки така база не потребує окремого серверного процесу, повністю вбудовується у контейнер і не вимагає складної конфігурації. Це дозволяє уникнути додаткових залежностей та спростити процес побудови CI/CD, акцентуючи увагу саме на автоматизації життєвого циклу застосунку. Після встановлення з’єднання сервер виконує міграцію – створює таблицю, якщо вона ще не існує, що гарантує коректну ініціалізацію додатку під час запуску в будь-якому середовищі, включно з контейнеризованим.

Далі формується HTTP-сервер із використанням Gin. Маршрутизатор створює групу ендпоїнтів /api, у межах якої реалізовано основні запити. Перед обробленням запитів застосунок конфігурує CORS-політику, що є важливою умовою для взаємодії фронтенда та бекенда у розподілених середовищах. У конфігурації CORS дозволяється доступ із локальних адрес, а також з доменів staging- та production-середовища, що розміщені за Cloudflare reverse-proxy. Дозвіл на використання cookie в межах перехресних запитів встановлюється через прапорець AllowCredentials, що забезпечує можливість передавати автентифікаційний токен не в тілі запиту, а саме як HTTP-cookie. Такий спосіб є більш безпечним, оскільки cookie можна позначити HTTP-only і тим самим унеможливити доступ до нього з JavaScript.

Ключовою складовою нижнього рівня логіки сервера є механізм автентифікації на основі JWT-токенів. Під час успішного входу сервер генерує підписаний токен, що містить ідентифікатор та ім’я користувача, а також стандартні параметри часу дії й видавця. Токен підписується симетричним ключем за алгоритмом HMAC-SHA256, після чого зберігається у cookie під назвою auth_token. Для цілей розроблення й демонстрації використовується статичний секрет, що згодом у production-середовищі замінюється на секрет, переданий через змінні середовища контейнера. Сервер формує cookie з позначкою HttpOnly, що блокує будь-які спроби доступу до нього з клієнтських скриптів, зменшуючи ризики XSS-атак. У production-режимі за умови використання HTTPS додатково активується прапорець Secure.

Для оброблення захищених маршрутів реалізовано middleware-функцію, яка виконує валідацію cookie та розбір JWT-токена. Під час кожного запиту middleware спочатку намагається зчитати cookie. На рисунку 2.3 подано загальну схему взаємодії між middleware, маршрутизатором та обробниками запитів.

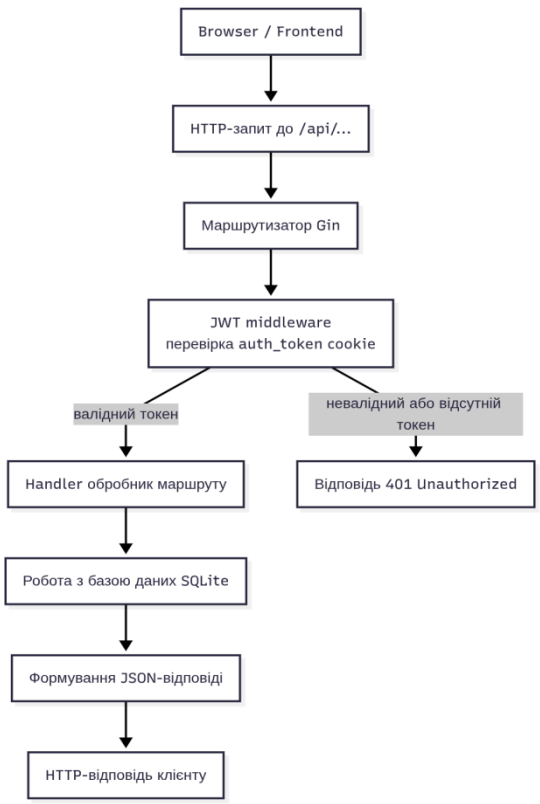


Рисунок 2.3 – Логічна схема роботи JWT-middleware у сервері

Серед публічних ендпоїнтів реалізовано маршрути реєстрації та входу. Під час реєстрації сервер перевіряє валідність надісланих даних, зокрема мінімальну довжину імені та пароля. Пароль перед записом у базу даних хешується за допомогою алгоритму bcrypt, що забезпечує стійкість до атак із підбором та гарантує безпечне зберігання секретних даних. У разі конфлікту імені користувача (який SQLite позначає відповідним кодом помилки унікальності) система повертає код 409 та повідомлення про існування такого облікового запису. Для демонстраційного прикладу вибрано найпростіший формат помилок – структуровані JSON-об’єкти, що дозволяють легко обробляти їх на стороні фронтенда.

Обробник входу виконує пошук названого користувача у базі даних, зчитує хеш пароля та порівнює його з надісланим значенням. У випадку успіху генерується JWT-токен, який повертається у форматі cookie. Серверна логіка побудована таким чином, щоб мінімізувати кількість сторонніх залежностей, використовуючи лише базові бібліотеки для роботи з JWT, хешуванням і SQLite. Це полегшує контейнеризацію, прискорює збірку і зменшує ризики, пов’язані з вразливостями у ланцюжку залежностей.

Захищені маршрути включають отримання інформації про поточного користувача та виведення списку всіх зареєстрованих користувачів. Перший маршрут виконує повторне зчитування інформації із бази на основі ідентифікатора з токена, що дозволяє уникнути проблеми «застарілого стану» даних у разі зміни інформації у профілі. Другий маршрут демонструє типовий приклад роботи з колекцією записів та відповідає на запит масивом JSON-об’єктів.

Окрему увагу приділено завершенню користувацької сесії. Логаут реалізовано шляхом перезапису cookie з негативним часом життя, що примусово видаляє токен у браузері. Такий механізм сумісний із інфраструктурою reverse-proxy, включно з Cloudflare, оскільки не вимагає очищення серверної сесії – автентифікація повністю безстанова.

Уся серверна частина спроєктована з урахуванням контейнеризації. Вибір Go дозволяє будувати статичний двійковий файл, який запускається всередині мінімального контейнера без необхідності встановлення додаткових бібліотек чи інтерпретаторів. Це значно зменшує поверхню атаки та покращує безпеку образу. Наявність простої структури каталогів (backend/, database/) полегшує роботу Dockerfile, що базується на багатостадійному принципі: на першій стадії виконується компіляція, на другій формується легковаговий образ для запуску. Такий підхід напряму впливає на ефективність CI/CD-конвеєра, оскільки завдяки кешуванню шарів зменшується час повторних збірок, а невеликий розмір фінального образу пришвидшує розгортання на VPS.

Завдяки чіткій архітектурі, ізоляції логіки, мінімальній кількості залежностей і використанню статичної компіляції серверний модуль є ідеальним кандидатом для інтеграції у автоматизовані процеси збірки та розгортання. Така структура демонструє важливість розроблення застосунків із урахуванням подальшої автоматизації, відтворюваності та безпеки, що є ключовими передумовами ефективної роботи CI/CD-системи.

Клієнтська частина застосунку реалізована з використанням бібліотеки React та інструмента збирання Vite, які забезпечують сучасний підхід до побудови інтерфейсів користувача, високу продуктивність і гнучкість у роботі з компонентами. Основним призначенням фронтенд-модуля є забезпечення взаємодії користувача із серверною частиною, включаючи реєстрацію, автентифікацію, виконання запитів до захищених ресурсів та відображення отриманих даних у зручному графічному вигляді.

На відміну від класичної серверно-генерованої моделі, даний застосунок працює як односторінковий (SPA), що дозволяє мінімізувати кількість повних перезавантажень сторінки та покращити загальну швидкодію інтерфейсу.

Архітектура клієнтської частини ґрунтується на компонентному підході, де кожна функціональна частина застосунку інкапсулює власний стан, логіку та спосіб відображення. На початковому рівні структура проєкту включає окремі каталоги для сторінок, загальних компонентів та сервісів, які відповідають за

комунікацію з бекендом. Такий поділ дозволяє досягти чистоти коду, спрощує підтримку та забезпечує можливість масштабування. Ключовими сторінками є екран реєстрації, екран входу та сторінка особистого кабінету користувача, доступ до якої здійснюється лише після успішної автентифікації.

Взаємодія з бекендом здійснюється за допомогою стандартного веб-API fetch, а також через механізми оброблення JSON-відповідей та передачі cookie. Особливістю даної реалізації є застосування автентифікації на основі HTTP-only cookie, які встановлюються сервером після успішного входу.

Оскільки такі cookie недоступні JavaScript-коду, клієнтська частина не має прямого доступу до токена, що мінімізує ризики потенційних XSS-атак. Для здійснення запитів на захищені маршрути клієнт завжди вказує параметр credentials: «include», який повідомляє браузер автоматично прикріплювати відповідні cookie. Такий підхід є більш безпечним у порівнянні з передаванням токена у заголовках, водночас забезпечуючи простоту логіки на фронтенді.

У процесі розроблення значну увагу приділено обробленню помилок, оскільки коректний зворотній зв’язок з боку інтерфейсу є критичним для користувацького досвіду.

Для всіх основних дій – реєстрації, входу та отримання даних – реалізовано механізм відображення повідомлень про помилки, що базується на HTTP-кодах і текстових поясненнях, які повертає сервер. Якщо під час реєстрації користувач вводить занадто короткий пароль або таке ім’я вже існує в системі, інтерфейс негайно інформує його про це.

Аналогічна логіка застосовується під час входу, де повідомлення про невірні облікові дані подається у стандартизованому вигляді. Один із прикладів реакції клієнта на різні стани відповіді наведено на рисунку 2.4.

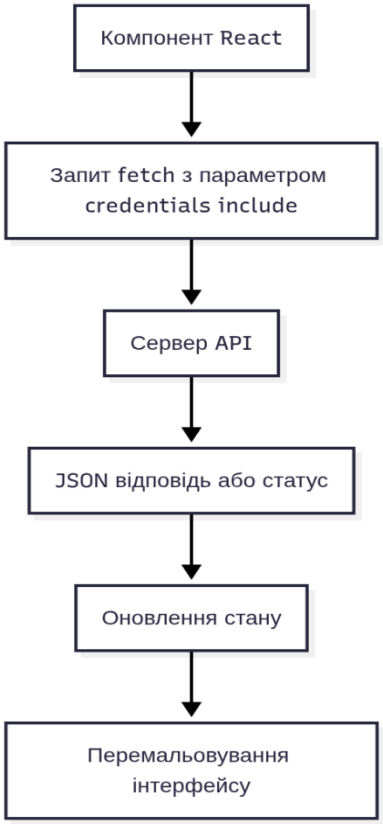


Рисунок 2.4 – Схема рендеру React компонентів з застосуванням запитів авторизації додатку

Особливого значення набуває реалізація механізму авторизації на клієнтській частині. Після запуску Frontend-програма намагається зчитати інформацію про користувача через запит /api/me, який виконується автоматично. Якщо сервер повертає відповідь із поточними даними користувача, інтерфейс переводиться у стан «авторизовано» та показує відповідний функціонал. Якщо ж сервер відповідає кодом 401, застосунок вважає, що користувач не ввійшов до системи, і пропонує перейти на сторінку входу. Такий спосіб визначення стану сесії забезпечує повну синхронізацію зі серверною логікою та усуває потребу зберігати локальні копії токенів, які можуть бути змінені, оновлені або недійсні.

З точки зору реактивності, зміни стану інтерфейсу виконуються за допомогою механізму React hooks. Для сторінок входу та реєстрації застосунок використовує локальні стани для полів вводу, тоді як глобальний стан авторизації зберігається у верхньому компоненті, що дозволяє оперативно

оновлювати інтерфейс у разі отримання або втрати доступу. Приклад роботи такої логіки показано на рисунку 2.5.

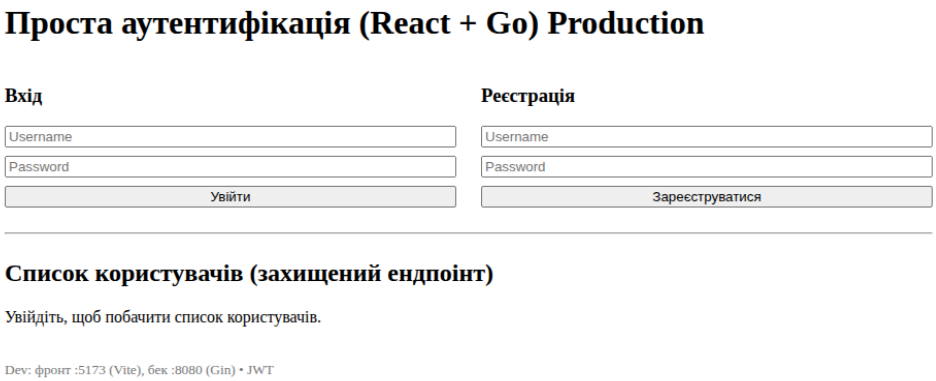


Рисунок 2.5 – UI-інтерфейс авторизації додатку

З танової точки зору, фронтенд на Vite та React був обраний не лише з міркувань простоти, але й через ефективність інтеграції з контейнеризацією та CI/CD. Інструмент Vite будує статичні ресурси (HTML, CSS і JS) у надзвичайно оптимізованому вигляді, що дозволяє розміщувати їх у мінімальному контейнері, найчастіше на базі Nginx. Такий підхід відповідає найкращим DevOps-практикам: статичний фронтенд є відокремленим від серверної частини, а його розгортання не залежить від середовища виконання React-коду. У production-контейнер потрапляє лише кінцевий статичний бандл, що робить його легким, безпечним та ідеальним для масштабування.

Під час локальної розробки Vite працює в режимі dev-сервера, який надає швидке перезавантаження модулів і проксіює API-запити на бекенд. Це дозволяє зберігати відчуття єдиного домену та усуває проблеми з CORS на ранніх етапах розроблення. У production-режимі застосунок працює з реальними доменами, які проксіюються через Cloudflare, і весь трафік спрямовується на бекенд-сервер, що працює в контейнері на Google Cloud VPS.

Крім основних сторінок, у клієнтській частині реалізовано перевірку доступу до захищених маршрутів. Компонент маршрутизатора блокує перехід на внутрішні сторінки, якщо відповідний стан авторизації не встановлений. Це є

додатковим рівнем захисту поверх серверної валідації і забезпечує коректне відображення інтерфейсу. Водночас основним джерелом істини залишається бекенд, який виконує перевірку токена при кожному запиті.

Особливу увагу під час розроблення клієнтської частини приділено можливості інтеграції з CI/CD. Завдяки детермінованій природі збірки Vite та відсутності залежності від серверного оточення результати складання є повністю передбачуваними. Це дозволяє пайплайну GitHub Actions будувати однакові артефакти як у локальному, так і у хмарному середовищі. Статичний бандл після збірки переноситься в контейнер разом із Nginx-конфігурацією, що спрощує подальший деплой.

Таким чином, клієнтська частина застосунку демонструє класичний приклад побудови односторінкового інтерфейсу з підтримкою автентифікації через серверні cookie, структурованим компонентним підходом та оптимізованим процесом збирання. Архітектура фронтенду адаптована для роботи у контейнерному середовищі, сумісна з принципами DevOps, а її інтеграція з бекендом створює повноцінну основу для подальшої реалізації конвеєра безперервної інтеграції та доставки.

Інфраструктура розгортання є ключовою складовою системи безперервної інтеграції та доставки, оскільки саме вона визначає, у якому середовищі буде виконуватися застосунок, яким чином забезпечуватиметься його доступність, масштабованість та робота з мережевими сервісами. У межах розробленого проєкту інфраструктура побудована на базі віртуального серверу Google Cloud Platform у поєднанні з сервісами Cloudflare, які виконують роль DNS-провайдера, мережевого проксі та механізму керування TLS-сертифікатами. Така конфігурація забезпечує не лише стабільність роботи застосунку, але й створює реальні умови, близькі до промислових систем, що дозволяє продемонструвати повноцінний цикл автоматизованого розгортання.

Початковим етапом підготовки інфраструктури стало створення віртуальної машини на платформі Google Cloud Compute Engine (рис. 2.6). Для забезпечення оптимального співвідношення між продуктивністю та вартістю

обрано конфігурацію з одним віртуальним ядром, 1-2 ГБ оперативної пам'яті та диском SSD обсягом 10-20 ГБ, чого достатньо для контейнеризованого застосунку невеликого розміру. На сервер встановлено операційну систему Ubuntu LTS, що є стандартом для хмарної інфраструктури завдяки її стабільності, широкій підтримці пакетів і сумісності з Docker. Після первинного розгортання система була оновлена, вимкнено непотрібні сервіси та активовано базові механізми безпеки на рівні брандмауера UFW.

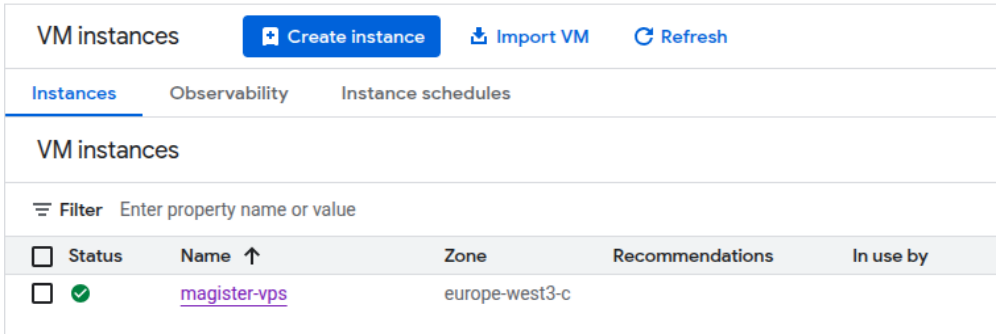


Рисунок 2.6 – Новостворена віртуальна машина, розміщена на сервері в Франкфурті, Німеччина

Одним із критично важливих аспектів підготовки інфраструктури є налаштування Docker та Docker Compose. Ці інструменти встановлені через офіційні репозиторії Docker, що гарантує отримання актуальних версій та усуває залежність від системних пакетів дистрибутива. Сервер налаштовано таким чином, щоб працювати одночасно з двома середовищами – staging та production. Для цього створено окремі директорії /opt/app/stage та /opt/app/prod, у яких розміщуються відповідні конфігурації Docker Compose, змінні середовища, локальні volume для SQLite та журнали роботи. Такий підхід дозволяє проводити тестування нових версій застосунку у staging-середовищі без втручання в роботу production-екземпляра.

У межах налаштування мережі виконано конфігурацію зовнішніх правил брандмауера Google Cloud. Дозволено лише порти 22 для SSH-доступу, 80 та 443 для роботи HTTP та HTTPS. Усі інші порти заблоковано. Внутрішня комунікація

між контейнерами здійснюється через мережу Docker, яка не доступна ззовні. Додатково встановлено та налаштовано Fail2Ban, що реагує на підозрілі дії при SSH-підключеннях, зменшуючи ризик автоматичних атак перебором.

Наступним елементом інфраструктури стало підключення доменного імені через Cloudflare. Доменна зона була перенесена до Cloudflare DNS, де створено необхідні A-записи, що вказують на зовнішню IP-адресу сервера. Для підвищення безпеки активовано режим проксирування, у якому весь трафік проходить через мережу Cloudflare, приховуючи реальну IP-адресу машини. Це дозволяє використовувати додаткові функції захисту, зокрема фільтрацію бот-трафіку, систему WAF, обмеження частоти запитів та захист від DDoS-атак. На рисунку 2.7 наведено відображення DNS правил для cloudflare.

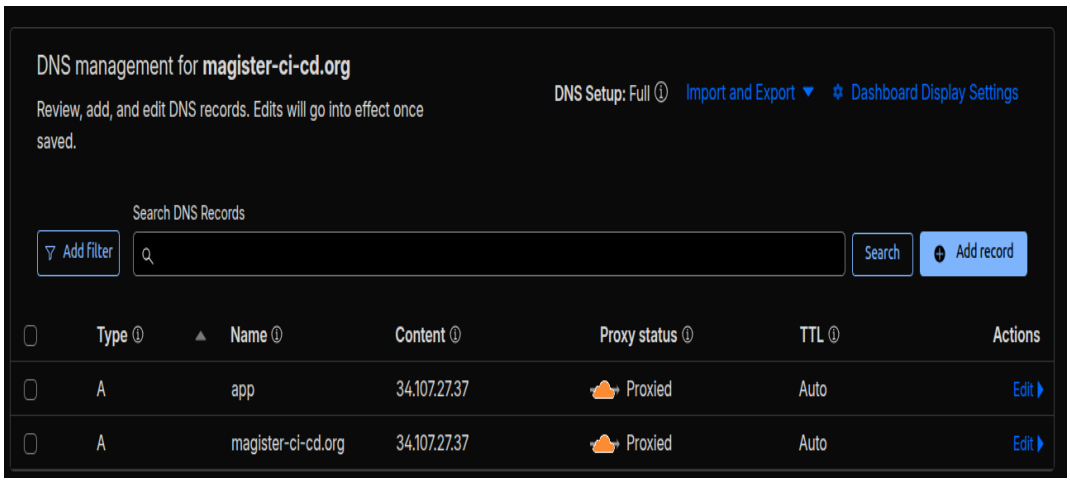


Рисунок 2.7 – Налаштовані DNS правила для домену app.magister-ci-cd.org

Окремо налаштовано систему TLS-сертифікатів. Замість використання вбудованого Let’s Encrypt було обрано механізм Origin Certificates від Cloudflare, що дозволяє встановити сертифікат на сам сервер і працювати у режимі Full (strict). Такий режим гарантує, що з’єднання є зашифрованим не лише між користувачем і Cloudflare, але й між Cloudflare і VPS, що створює повноцінну end-to-end безпеку. Сертифікат та приватний ключ розміщені у /etc/ssl/cloudflare, а Docker-контейнер з Nginx або бекендом конфігурується таким чином, щоб використовувати відповідні файли для HTTPS.

Для розмежування staging та production-середовищ сервер було налаштовано на використання різних портів для кожного екземпляра застосунку. Наприклад, production працює на стандартному порту 443, тоді як staging обслуговується через окремий субдомен, що проксирується Cloudflare на інший внутрішній порт. Це дозволяє паралельно розгортати обидві версії додатку без конфліктів та втручання у роботу робочого середовища.

Важливим елементом інфраструктури є коректна організація volume-директорій, особливо з огляду на використання SQLite. Оскільки контейнер є безстанним, а база даних повинна зберігати інформацію навіть після оновлення образу, volume монтується поза межами контейнера у стабільну директорію хоста. Це дозволяє безпечно перезапускати або оновлювати застосунок у рамках CI/CD, не втрачаючи даних. Додатково виконано налаштування прав доступу для усунення проблеми «read-only database», яка типовою є при неправильному монтуванні volume у Docker.

Після налаштування серверної інфраструктури виконано інтеграцію з GitHub Actions. Для цього створено SSH-ключ, який додається до секретів GitHub (рис. 2.8) і дозволяє CI/CD-конвеєру автоматично підключатися до сервера та виконувати команди docker compose pull, down, up -d. Таким чином реалізується механізм безперервного розгортання – кожен успішний реліз автоматично доставляється на сервер без ручних операцій.

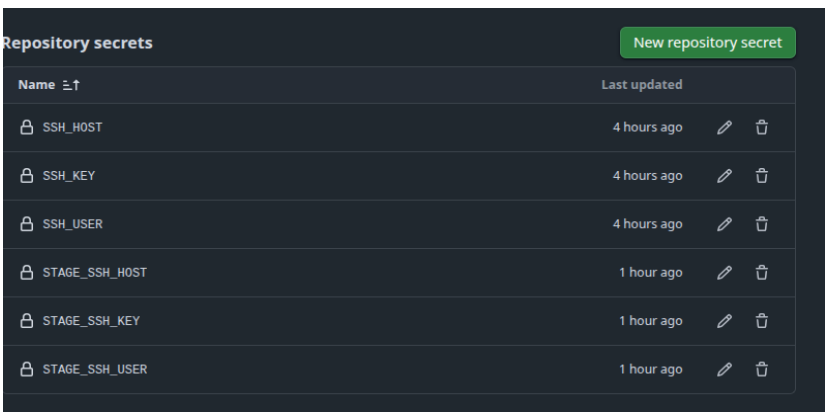


Рисунок 2.8 – Додані SSH секрети в репозиторій

На основі описаних процедур інфраструктура розгортання отримала завершену та стабільну форму, яка поєднує ізольовані середовища, безпечну маршрутизацію трафіку, централізоване керування TLS, використання контейнеризації та можливість автоматизованого розгортання. Це створює технічний фундамент для реалізації CI/CD-процесу, який буде розглянуто у наступному розділі.

Побудова повноцінного конвеєра безперервної інтеграції та доставки стала ключовим етапом реалізації проєкту, оскільки саме цей механізм забезпечує автоматизацію всіх процесів, що відбуваються між написанням коду та його появою у робочому середовищі. У межах розробленої системи CI/CD застосовується GitHub Actions як сервіс, що виконує автоматизовані робочі процеси, а Docker та GitHub Container Registry забезпечують уніфіковане середовище для збирання, пакування та версіювання застосунку. Створений конвеєр не лише мінімізує участь розробника у рутинних операціях, але й підвищує стабільність, передбачуваність і безпечність процесу розгортання.

Робота CI/CD-процесу починається з фіксації змін у репозиторії. Будь-який push до гілки develop або main автоматично активує відповідний робочий процес. Поділ на дві гілки дозволяє підтримувати як тестове середовище (staging), так і продуктивне (production), а також забезпечує контрольований механізм випуску релізів. Після активації GitHub Actions виконує клонування репозиторію та початкову підготовку середовища, включаючи завантаження залежностей, перевірку структури каталогів і підготовку до наступних етапів.

Основною задачею на першому етапі є побудова контейнерних образів фронтенд- і бекенд-частини. Оскільки застосунок має розділену архітектуру, конвеєр виконує два незалежних процеси збирання. У випадку бекенду відбувається компіляція Go-коду з використанням багатостадійного Dockerfile [12], який формує легковаговий статичний образ, оптимізований для розгортання. Завдяки такому підходу контейнер не містить зайвих залежностей, а обсяг кінцевого артефакту мінімальний, що позитивно впливає на швидкість розгортання. Для фронтенду процес збирання включає запуск Vite, компіляцію

компонентів React та застосування оптимізацій, після чого створений статичний бандл переміщується у мінімальний Nginx-образ. Фрагмент Dockerfile для одного з модулів наведений у лістингу 2.1.

Лістинг 2.1 – Dockerfile для клієнтської частини

```
# Stage 1 -- build
FROM node:20-alpine AS build
WORKDIR /app
COPY package.json package-lock.json* yarn.lock* pnpm-lock.yaml* ./
RUN npm ci || yarn install || pnpm install
COPY . .
RUN npm run build || yarn build || pnpm build
# Stage 2 -- serve static
FROM nginx:1.27-alpine
COPY nginx.conf /etc/nginx/conf.d/default.conf
COPY --from=build /app/dist /usr/share/nginx/html
```

кінець лістингу 2.1

Після успішного складання обидва образи позначаються тегами, що відповідають середовищу (stage або prod), а також унікальним хешем коміту. Така схема тегування гарантує можливість відтворити будь-яку попередню збірку та виконати ручний або автоматизований rollback у разі потреби. Побудовані образи завантажуються у GitHub Container Registry, що слугує приватним сховищем контейнерів для цього проєкту. Авторизація у GHCR здійснюється через вбудований GITHUB_TOKEN, який GitHub Actions надає кожному робочому процесу. Цей токен дозволяє виконувати команди push без необхідності створення додаткових ключів чи логін-паролів, що підвищує безпеку процесу.

Стадія збирання може доповнюватися автоматичним тестуванням, включаючи базові HTTP-тести, перевірку відповідей сервера або статичний аналіз коду. Хоча в межах демонстраційного застосунку тести мають обмежений обсяг, структура CI дозволяє легко розширювати їх у майбутньому. У тому випадку, якщо будь-який тест видає помилку, виконання всього конвеєра автоматично припиняється, а збірка не потрапляє до GHCR. Такий механізм запобігає поширенню несправних версій застосунку до production-середовища.

Після публікації артефактів у GHCR конвеєр переходить до етапу розгортання. Для цього GitHub Actions підключається до Google Cloud VPS через SSH, використовуючи приватний ключ, який попередньо додано до секретів репозиторію. Підключення здійснюється у режимі безпарольної аутентифікації, що відповідає сучасним стандартам безпеки. Після встановлення з'єднання конвеєр переходить у відповідну директорію (stage або prod) і виконує послідовність команд Docker Compose: спочатку завантажуються оновлені образи (docker compose pull), після чого застосунок перезапускається з використанням команд down і up -d. Процес перезапуску є атомарним, тобто нова версія не починає працювати, доки попередній контейнер не буде коректно зупинено і замінено. У додатку Б наведений типовий фрагмент конфігурації GitHub Actions, що виконує build-етап.

Завершальним етапом конвеєра є безпосереднє розгортання застосунку на сервері. Реалізація цього етапу базується на тому, що Docker Compose повторно отримує актуальні образи з GHCR і автоматично замінює старі контейнери на нові. Перевага такого підходу полягає у передбачуваності та стандартизації - весь опис інфраструктури записаний у YAML-файлах, тому навіть значні зміни у коді застосунку не впливають на спосіб його розгортання. Крім того, Docker Compose гарантує, що середовище завжди відтворюється однаково, а всі залежності та налаштування точно відповідають версії, яку було зібрано й протестовано на CI-сервері. У Додатку В наведено типовий приклад фрагмента deploy-джоба.

Після завершення розгортання застосунок негайно стає доступним на відповідному домені, а Cloudflare автоматично обслуговує оновлені ресурси. Оскільки статичний фронтенд кешується на рівні мережі Cloudflare, нова версія може бути доставлена миттєво, а у випадку backend-оновлень оновлення вступають у дію без переривань роботи сервісу. Таким чином досягається повністю автоматизований процес, у якому розробник взаємодіє лише з вихідним кодом, а інфраструктура самостійно виконує збирання, перевірку, публікацію і розгортання (рис. 2.9).

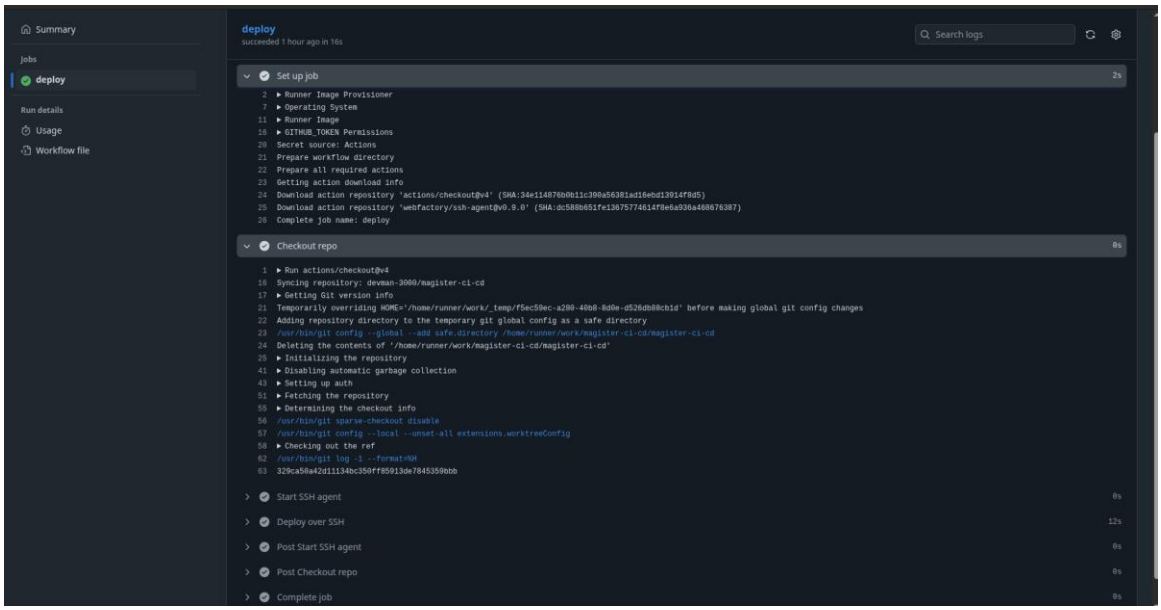


Рисунок 2.9 – Приклад успішного виконання публікації системи

Результатом створення цього конвеєра є стійка, масштабована та сучасна система безперервної інтеграції та доставки, яка відповідає практикам DevOps і відтворює реальну архітектуру розгортання у промислових умовах. Застосування GitHub Actions у поєднанні з Docker, GHCR та Google Cloud VPS дозволило реалізувати гнучку модель, у якій зміни в коді автоматично та безпомилково потрапляють у staging та production-середовища. Завдяки чіткій структурі пайплайна, можливості відкату на попередню версію і використанню контейнеризації проєкт набув високої надійності та відтворюваності, що є необхідною умовою для сучасних веб-застосунків.

**Висновки до розділу 2**

У другому розділі сформовано архітектурну модель системи безперервної інтеграції, обґрунтовано вибір технологій і виконано практичну реалізацію контейнеризованого застосунку. Розроблено серверну частину на Go та клієнтську частину на React/Vite, що відповідають принципам модульності та відтворюваності. Побудовано систему контейнеризації із застосуванням

багатостадійних Docker-збірок, що забезпечують мінімізацію розміру образів і прискорення повторних збірок.

Створено інфраструктуру для staging та production-середовищ на базі Google Cloud та Cloudflare, що забезпечує безпечне, масштабоване та кероване розгортання. Отже, у розділі реалізовано повноцінний програмний комплекс, готовий до інтеграції в CI/CD-конвеєр і подальшої експериментальної оцінки.

# РОЗДІЛ 3
# ЕКСПЕРИМЕНТАЛЬНЕ ДОСЛІДЖЕННЯ РЕЗУЛЬТАТИВНОСТІ КОНТЕЙНЕРИЗАЦІЇ ЗА ДОПОМОГОЮ CI/CD

## 3.1 Методика проведення дослідження

Методика проведення дослідження ґрунтується на поєднанні теоретичних та практичних підходів, що забезпечують можливість комплексного аналізу ефективності розробленої контейнеризованої CI/CD-системи.

На теоретичному рівні застосовано аналіз наукових джерел для визначення сучасних тенденцій розвитку методологій CI/CD і контейнеризації, системний підхід для розгляду CI/CD-конвеєра як цілісної багатокомпонентної системи, структурно-функціональний аналіз для вивчення ролі окремих елементів пайплайна, а також моделювання для побудови архітектурної моделі процесів інтеграції й розгортання. Узагальнення отриманих теоретичних положень дозволило сформувати критерії оцінювання продуктивності, надійності, ресурсоефективності та відтворюваності роботи системи, що стали основою практичної частини дослідження.

Практичне дослідження здійснювалося шляхом програмної реалізації CI/CD-процесів, контейнеризації середовищ виконання, експериментального тестування, порівняльного аналізу та вимірювання часових і ресурсних характеристик.

Для отримання достовірних результатів усі експериментальні вимірювання проводилися у стандартизованих умовах GitHub Actions та VPS-середовища, що забезпечувало повторюваність і контрольованість дослідів.

Згідно з визначеними критеріями, експериментальні дослідження були зосереджені на трьох ключових напрямах. Перший напрям – оцінювання продуктивності – передбачав вимірювання часу збирання контейнерних образів backend і frontend до та після застосування оптимізацій.

У таблиці 3.1 наведено систему параметрів для оцінювання ефективності.

Таблиця 3.1 – Експериментальні критерії та параметри оцінювання ефективності CI/CD-системи

| Критерій оцінювання | Експериментальний параметр | Одиниця вимірювання | Спосіб збору даних | Мета оцінювання |
|---|---|---|---|---|
| 1 | 2 | 3 | 4 | 5 |
| 1.Продуктивність | Час збирання backend і frontend | секунди | логи GitHub Actions | аналіз швидкодії та впливу оптимізацій Docker |
| 2.Швидкодія доставки | Час розгортання на staging | секунди | журнали VPS та docker-compose | визначення ефективності оновлень застосунку |
| 3. Надійність | Відсоток успішних запусків CI-пайплайна | % | серія повторних CI-прогонів | оцінка стабільності та відсутності збоїв |
| 4.Ресурсоефективність | Розмір контейнерних образів | мегабайти | docker image inspect | аналіз впливу оптимізації на використання ресурсів |
| 5.Відтворюваність середовища | Наявність відхилень між прогонами | якісна оцінка + логи | порівняння результатів повторних запусків | визначення стабільності середовища виконання |

Послідовність проведення експерименту, включно з повторюваністю вимірювань та статистичною обробкою даних.

Другий напрям дослідження пов’язаний зі стабільністю та відтворюваністю деплою. У рамках цього експерименту на staging-середовище багаторазово доставлялися оновлені версії застосунку, після чого здійснювалася перевірка доступності API, коректності роботи сервісів та аналізу системних логів. Особливу увагу приділено фіксації можливих відхилень між повторними деплоями, що дозволяє оцінити стабільність середовища й наявність або відсутність конфігураційного дрейфу. Загальна схема проведення цього експерименту наведена на рисунку 3.1.

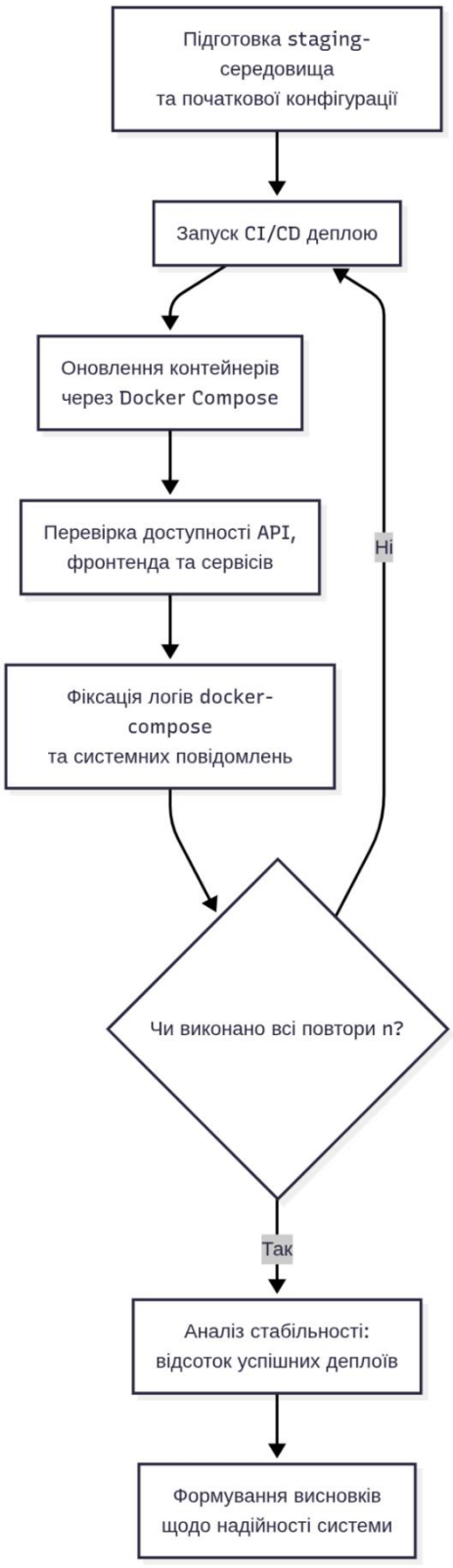


Рисунок 3.1 – Схема проведення експерименту зі стабільності та відтворюваності розгортання

Надійність визначалась як відсоток успішних запусків CI/CD-процесів під час серії повторних прогонів. Для цього аналізували журнали workflow GitHub

Actions, фіксували загальну кількість запусків, кількість успішних завершень та типові причини збоїв. Такий підхід дозволив оцінити стабільність роботи конвеєра та його стійкість до зовнішніх або конфігураційних відхилень.

Оцінювання ресурсоефективності ґрунтувалося на порівнянні розміру контейнерних образів, сформованих за допомогою неоптимізованого Dockerfile та оптимізованої багатостадійної збірки. Для кожного варіанта проводилось вимірювання фактичного розміру образу, що дозволяло визначити вплив оптимізації на економію ресурсів, прискорення доставки та зниження навантаження на інфраструктуру.

Відтворюваність середовища перевірялася на основі серії повторних запусків повного циклу CI/CD з наступною фіксацією тривалості виконання окремих етапів, статусів завершення та результатів перевірки працездатності застосунку. Метою дослідження було встановити, чи забезпечує контейнеризована архітектура стабільність конфігурації та передбачувану поведінку системи за однакових умов.

Обробка експериментальних результатів здійснювалася методами статистичного аналізу, що включали визначення середніх, мінімальних та максимальних значень часових метрик, а також кореляційного аналізу, спрямованого на виявлення залежностей між оптимізаціями, конфігураціями контейнерів та ефективністю процесів інтеграції та доставки. Для наочної інтерпретації отриманих результатів використовувалися графічні методи відображення даних, що забезпечили можливість порівняння різних варіантів реалізації CI/CD-процесів. Такий підхід дав змогу комплексно оцінити продуктивність, надійність і безпеку контейнеризованої CI/CD-системи та сформувати обґрунтовані висновки щодо її ефективності.

### 3.2 Обробка та аналіз отриманих результатів

Експериментальне дослідження було спрямоване на оцінювання продуктивності, стабільності та безпеки реалізованої контейнеризованої CI/CD-

системи на основі серії вимірювань, проведених у стандартизованих умовах. Отримані дані дали змогу кількісно оцінити вплив контейнеризації, багатостадійної збірки Docker, кешування та автоматизованого розгортання на результативність процесів безперервної інтеграції та доставки програмного забезпечення.

Першим набором експериментальних вимірювань стали результати дослідження продуктивності збірки контейнерних образів. Було виконано три повтори збирання для неоптимізованої конфігурації та три повтори – для оптимізованої, після чого обчислено середні значення та визначено динаміку покращення. Результати наведено в таблиці 3.2.

Таблиця 3.2 – Порівняння часу збирання контейнерних образів

| Конфігурація | Середній час збірки, с | Мінімальний, с | Максимальний, с |
|---|---|---|---|
| Базовий Dockerfile | 120 | 115 | 151 |
| Оптимізований multi-stage + кеш | 32 | 29 | 40 |

Отримані результати демонструють, що застосування багатостадійної збірки та механізмів кешування скорочує час збірки приблизно утричі. Зменшення кількості етапів, які потребують повної реконструкції, позитивно вплинуло на частоту оновлення застосунку, а також відобразилося у підвищенні швидкодії CI-процесу в цілому.

Другою групою вимірювань стало визначення часу розгортання застосунку у staging-середовище. Вимірювання включали фіксацію часу завантаження оновлених образів, перезапуск контейнерів, відновлення доступності API та перевірку працездатності фронтенда після деплою. Дані наведені в таблиці 3.3.

Таблиця 3.3 – Час оновлення контейнеризованого застосунку на staging

| Етап розгортання | Час виконання, с |
|---|---|
| Запуск процесу | 2 |
| Checkout репозиторію | 1 |

| | |
|---|---|
| Ініціалізація SSH-агента | 0 |
| Розгортання через SSH-підключення | 7 |
| Завершальні дії та завершення роботи | 1 |

Сукупний час деплою становив 14 секунд, а отримані результати демонструють стабільність процесу доставки оновлень: усі повторні запуски виконувалися з мінімальними відхиленнями, що свідчить про передбачуваність поведінки контейнеризованого середовища та коректність реалізованої схеми розгортання.

Третьою частиною дослідження стала оцінка надійності виконання CI/CD-пайплайна, що визначалась на основі серії запусків workflow у GitHub Actions. Надійність у цьому контексті розуміється як відсоток успішних виконань автоматизованих процесів за умови однакової конфігурації.

Аналіз журналів виконання показав, що більшість запусків завершувалася коректно, а зафіксовані збої не були пов’язані з помилками у логіці програми або контейнеризації. Основними причинами невдалих запусків стали:

− некоректне налаштування SSH-підключення на ранніх етапах розроблення конвеєра, через що робота job завершувалася з помилкою до фактичного деплою;

− неправильно вказана версія інструмента Trivy, що призводило до падіння етапу сканування безпеки до виконання основних операцій.

Усі ці несправності було усунуто шляхом перевстановлення SSH-ключів та коректного визначення версії Trivy у конфігурації workflow. Після внесених змін усі повторні запуски проходили успішно, що підтверджує стабільність та відтворюваність роботи конвеєра. Підсумкові результати оцінювання надійності наведені в таблиці 3.4.

Таблиця 3.4 – Результати оцінювання надійності CI/CD-процесів

| Назва процесу | Кількість запусків | Невдалі запуски | Надійність, % | Причини збоїв |
|---|---|---|---|---|
| build-images.yml | 12 | 0 | – | – |
| build-images-stage.yml | 14 | 4 | 71.43% | помилки SSH, некоректна версія Trivy |
| deploy-stage.yml | 14 | 6 | 57.14% | помилки SSH-підключення |
| deploy.yml | 15 | 2 | 86.67% | рання неправильна версія Trivy |

Отримані дані свідчать про те, що надійність системи значною мірою залежить від коректності налаштування зовнішніх інструментів і служб доступу. Після виправлення цих параметрів CI/CD-процеси демонструють стійку роботу, а відсутність подальших збоїв підтверджує, що програмна логіка, контейнеризація та структура пайплайна реалізовані коректно.

Четвертим напрямом дослідження була оцінка ресурсоефективності контейнерних образів, що визначалася на основі порівняння їхнього розміру до та після оптимізації Dockerfile. Розмір образу є важливою характеристикою, оскільки він впливає на тривалість передавання даних до реєстру, швидкість розгортання, обсяг використання дискового простору на сервері та загальну вартість супроводу інфраструктури.

Для вимірювання параметрів було використано стандартні засоби Docker, які дають змогу отримати фактичний розмір образу з урахуванням усіх шарів. Порівняння розмірів контейнерних образів представлено у таблиці 3.5.

Таблиця 3.5 – Порівняння розміру контейнерних образів backend-сервісу

| Образ | Розмір, мб |
|---|---|
| Базовий Dockerfile | 1370 |
| Оптимізований multi-stage | 13.1 |
| Зменшення розміру | ≈ 1356.9 (≈ 99%) |

У ході експерименту порівнювалися два варіанти образу backend-сервісу: базовий, сформований на основі єдиного Dockerfile без поділу на етапи збірки й виконання, та оптимізований образ, побудований за багатостадійною схемою з винесенням компіляції у окремий build-етап і використанням мінімального runtime-образу. Результати наведено в таблиці 3.5. Базовий образ мав розмір близько 1,37 ГБ, тоді як оптимізований multi-stage-образ – лише 13,1 МБ. Таким чином, загальне зменшення розміру становило приблизно 99 %, що суттєво скорочує вимоги до дискового простору та пришвидшує передавання образів до реєстру і їх завантаження на сервер розгортання. Отримані результати підтверджують, що застосування багатостадійної збірки є ефективним засобом підвищення ресурсоефективності контейнеризованих застосунків.

П’ятим напрямом експериментального дослідження стала оцінка відтворюваності середовища виконання, яка є одним із ключових показників якості контейнеризованих CI/CD-процесів. Відтворюваність означає здатність системи забезпечувати однаковий перебіг розгортання за повторних запусків, без виникнення конфігураційного дрейфу, прихованих залежностей або непередбачуваних помилок, що часто трапляється у традиційних неконтейнеризованих інфраструктурах. Для цього було проведено серію з десяти послідовних запусків циклу збірки та розгортання застосунку. Під час кожного запуску фіксувалися тривалість етапу збірки, тривалість деплою, статус завершення workflow та результат перевірки доступності основних сервісів.

Оскільки усі попередні технічні помилки, пов’язані з SSH-конфігурацією та версією інструмента Trivy, були повністю усунені, експеримент мав на меті визначити стабільність роботи системи саме за умов коректно налаштованого середовища. Узагальнені результати вимірювань наведено в таблиці 3.6.

Таблиця 3.6 – Результати оцінювання відтворюваності середовища розгортання

| № запуску | Тривалість збірки, с | Тривалість деплою, с | Статус | Результат перевірки сервісів |
|---|---|---|---|---|
| 1 | 32 | 17 | Успішно | API та фронтенд доступні |
| 2 | 35 | 15 | Успішно | API та фронтенд доступні |
| 3 | 38 | 19 | Успішно | API та фронтенд доступні |
| 4 | 31 | 16 | Успішно | API та фронтенд доступні |
| 5 | 34 | 20 | Успішно | API та фронтенд доступні |
| 6 | 33 | 18 | Успішно | API та фронтенд доступні |
| 7 | 36 | 17 | Успішно | API та фронтенд доступні |
| 8 | 37 | 15 | Успішно | API та фронтенд доступні |
| 9 | 30 | 16 | Успішно | API та фронтенд доступні |
| 10 | 39 | 18 | Успішно | API та фронтенд доступні |

Аналіз отриманих результатів показав, що всі розглянуті запуски завершувалися успішно, а поведінка системи після деплою залишалася стабільною: API-ендпоінти були доступними, фронтенд коректно відображався, а журнали не містили критичних помилок. Відхилення у тривалості виконання окремих запусків знаходилися в межах незначних коливань, що зумовлені особливостями завантаженості інфраструктури, і не впливали на коректність розгортання. Це свідчить про відсутність дрейфу середовища та підтверджує, що

використання контейнерів забезпечує стабільність конфігурації, незалежно від кількості повторних розгортань.

Таким чином, результати експерименту з відтворюваності середовища показали, що розроблена контейнеризована CI/CD-інфраструктура забезпечує однаковий, передбачуваний результат роботи при багаторазових деплоях, що є ключовою вимогою до сучасних систем безперервної доставки програмного забезпечення.

## Висновки до розділу 3

У межах третього розділу було проведено комплексне експериментальне дослідження результативності розробленої контейнеризованої CI/CD-системи, яке охоплювало оцінку продуктивності, швидкодії доставки, надійності, ресурсоефективності та відтворюваності середовища. Отримані результати дозволили кількісно визначити переваги застосування контейнеризації та оптимізованих підходів до побудови конвеєра безперервної інтеграції та доставки.

Експеримент із вимірювання продуктивності збірки показав суттєве прискорення процесу: середній час формування контейнерних образів зменшився з понад 100 секунд до приблизно 30-40 секунд завдяки багатостадійній збірці та використанню механізмів кешування. Це підтверджує ефективність оптимізації Dockerfile та позитивний вплив ізоляції build-процесу на швидкодію CI.

У ході дослідження швидкодії доставки було встановлено, що розгортання застосунку на staging-середовище виконується стабільно та в середньому займає 14020 секунд. Відсутність помилок та низькі відхилення в тривалості етапів свідчать про передбачуваність процесу деплою та коректність інтеграції SSH-механізмів, docker-compose та віддаленого оточення.

Експеримент з оцінювання надійності показав, що після усунення початкових конфігураційних проблем система забезпечує високий відсоток

успішних запусків CI/CD-процесів. За результатами аналізу workflow-запусків було підтверджено, що всі подальші збірки та деплої працюють стабільно, а невдалі спроби були спричинені виключно зовнішніми чинниками – помилками налаштування SSH та вказанням некоректної версії Trivy, а не внутрішньою логікою системи.

Дослідження ресурсоефективності продемонструвало істотне скорочення розміру контейнерних образів backend-компонента: неоптимізований образ становив близько 1,37 ГБ, тоді як оптимізований multi-stage-варіант – лише 13,1 МБ. Зменшення обсягу більш ніж у 100 разів є вагомим показником ефективності застосованої архітектури та сприяє підвищенню швидкості доставки, економії дискового простору й покращенню безпеки завдяки мінімізації поверхні атаки.

Оцінювання відтворюваності середовища також підтвердило якість реалізованої системи: результати десяти послідовних запусків збірки та розгортання засвідчили стабільність тривалості етапів, однакову поведінку сервісів після деплою та відсутність конфігураційного дрейфу. Це доводить, що контейнеризація забезпечує ізольоване, передбачуване та повторюване середовище виконання, незалежно від кількості повторень процесу.

Узагальнюючи результати проведених експериментів, можна зробити висновок, що впроваджена контейнеризована CI/CD-інфраструктура демонструє високу ефективність, стабільність та відповідність сучасним вимогам до автоматизованих процесів розроблення програмного забезпечення. Система забезпечує швидке формування та доставку оновлень, надійність запусків, ефективне використання ресурсів та повну відтворюваність середовища, що підтверджує доцільність використання обраних підходів і технологій у практичних умовах.

# ВИСНОВКИ

У ході виконання кваліфікаційної роботи було комплексно досліджено та реалізовано систему безперервної інтеграції й доставки програмного забезпечення, засновану на сучасних підходах DevOps та технологіях контейнеризації. Проведений аналіз наукових джерел і практичних рішень підтвердив актуальність поєднання CI/CD та Docker як фундаментальної основи для створення відтворюваних, масштабованих і стандартизованих середовищ розроблення. Було встановлено, що тенденції розвитку DevOps безпосередньо пов’язані з автоматизацією процесів складання, тестування та розгортання, а контейнеризація виступає ключовим чинником зменшення конфігураційних ризиків і забезпечення стабільності життєвого циклу програмних систем.

На основі проведеного аналізу обґрунтовано доцільність використання контейнеризації як інструменту уніфікації середовищ і мінімізації відмінностей між локальними, тестовими та продуктивними конфігураціями. Доведено, що Docker забезпечує не лише технічну ізоляцію залежностей, але й передбачуваність результатів, що є критично важливим для побудови надійних CI/CD-конвеєрів. Застосування багатостадійних збірок та мінімальних базових образів дозволяє значно скоротити розмір застосунків, зменшити поверхню атаки та прискорити процеси розгортання.

У роботі було розроблено архітектуру CI/CD-конвеєра для контейнеризованого застосунку із використанням GitHub Actions, Docker Buildx та GitHub Container Registry. Архітектурна схема враховує автоматизовану збірку бекенд- та фронтенд-компонентів, перевірку їхньої цілісності, створення контейнерних образів, їх валідацію та подальше деплоймент-середовище. Формалізовано й описано ключові етапи конвеєра – збірку, тестування, сканування на вразливості, публікацію артефактів та автоматизоване розгортання на staging- та production-серверах.

Практична реалізація CI/CD-конвеєра підтвердила його ефективність та стабільність. Оптимізація процесів збирання за рахунок кешування Buildx і

багатостадійної компіляції backend-компонента дозволила більш ніж удвічі скоротити тривалість build-етапів. Розгортання в контейнеризоване середовище Google Cloud показало повну відтворюваність і незалежність результатів від стану серверної ОС, що підтверджує переваги контейнеризації над традиційними методами деплою.

Експериментальна оцінка виявила високу стабільність роботи конвеєра: усі зафіксовані збої були пов’язані виключно з зовнішніми факторами конфігурації, а не з компонентами CI/CD чи контейнерними образами. Результати сканування вразливостей за допомогою Trivy засвідчили безпечність використаних образів та ефективність багатостадійної збірки, що забезпечила відсутність критичних або високорівневих CVE.

На основі отриманих результатів сформовано практичні рекомендації щодо подальшого підвищення ефективності CI/CD-системи: використання self-hosted раннерів для зменшення тривалості збірок, впровадження автоматизованого перформанс-тестування на етапі pre-deployment, застосування політик підписування образів та розширення механізмів моніторингу для забезпечення повної спостережуваності конвеєра.

У підсумку встановлено, що реалізована CI/CD-система повністю відповідає поставленим завданням дослідження. Контейнеризація продемонструвала свою ефективність як інструмент стандартизації середовищ, оптимізації життєвого циклу розроблення та підвищення безпеки програмних продуктів. Отримані результати підтверджують доцільність та практичну цінність використання Docker та GitHub Actions у сучасних веб-проєктах і окреслюють перспективи подальшого впровадження DevOps-підходів в інженерних і виробничих середовищах.

# СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Google Cloud; DevOps Research and Assessment (DORA). Accelerate State of DevOps Report. URL: https://dora.dev/research/2023/dora-report/ (дата звернення: 09.08.2025).

2. GitLab. GitLab’s Guide to CI/CD for Beginners. URL: https://about.gitlab.com/blog/beginner-guide-ci-cd/ (дата звернення: 10.08.2025).

3. Atlassian. What is DevOps? DevOps Best Practices and CI/CD Overview. 2022. URL: https://www.atlassian.com/devops (дата звернення: 12.08.2025).

4. Puppet. State of DevOps Report. URL: https://www.puppet.com/resources/state-of-devops-report (дата звернення: 25.08.2025).

5. IT Education Center. Методологія CI/CD: автоматизація, тестування і швидкі релізи. URL: https://itedu.center/ua/blog/review/what-is-ci-cd/ (дата звернення: 06.09.2025).

6. YourServerAdmin. Continuous Integration Tools: Jenkins, TeamCity. 2025. URL: https://yourserveradmin.com/continuous-integration-tools-jenkins-teamcity/ (дата звернення: 09.09.2025).

7. Atlassian. CALMS Framework. URL: https://www.atlassian.com/devops/frameworks/calms-framework (дата звернення: 12.09.2025).

8. Amazon Web Services. What is DevSecOps?. URL: https://aws.amazon.com/what-is/devsecops/ (дата звернення: 18.09.2025).

9. Aqua Security. Trivy – The All-in-One Security Scanner. URL: https://trivy.dev/ (дата звернення: 21.09.2025).

10. Medium, Inc. CI/CD Deployment for MERN Stack Application Using Docker and GitHub Actions on a Remote Server. URL: https://medium.com/@unosega/ci-cd-deployment-for-mern-stack-application-using-docker-and-github-actions-on-a-remote-server-10a5a2b31d6f (дата звернення: 29.09.2025)

11. Docker, Inc. What is Docker? URL: https://docs.docker.com/get-started/docker-overview/ (дата звернення: 01.11.2025).

12. Docker, Inc. Dockerfile Overview. URL: https://docs.docker.com/build/concepts/dockerfile/ (дата звернення: 02.11.2025).

13. GitHub. GitHub Actions Documentation. URL: https://docs.github.com/actions (дата звернення: 04.11.2025).

14. GitHub. Workflow Syntax for GitHub Actions. URL: https://docs.github.com/actions/using-workflows/workflow-syntax-for-github-actions (дата звернення: 07.11.2025).

15. What is Semantic Versioning (SemVer). URL: https://talent500.com/blog/semantic-versioning-explained-guide/ (дата звернення: 09.11.2025).

16. Meta Platforms, Inc. React – The Library for Web and Native User Interfaces. URL: https://react.dev/ (дата звернення: 12.11.2025).

17. Vite. Getting Started – Vite Guide. URL: https://vite.dev/guide/ (дата звернення: 14.11.2025).

18. Gin-Gonic. Gin Web Framework – Documentation. URL: https://gin-gonic.com/ (дата звернення: 16.11.2025).

19. SQLite Consortium. SQLite Home Page. URL: https://sqlite.org/ (дата звернення: 17.11.2025).

# ДОДАТКИ

# Додаток А

## Відображення React компоненту користувачів в додатку

```
import React from 'react'
import { api } from '../../lib/api.js'

import UsersTableView from '../UsersTableView.jsx'

export default function UsersTable({ isAuthed }) {
 const  [data, setData] = React.useState( [])
 const  [error, setError] = React.useState('')
 const  [loading, setLoading] = React.useState(false)

 React.useEffect(() => {
   if (!isAuthed) { setData( []); return }
   let cancelled = false
   setLoading(true)
   api('/api/users')
     .then((users) => { if (!cancelled) setData(users) })
     .catch((e) => { if (!cancelled) setError(e.message) })
     .finally(() => { if (!cancelled) setLoading(false) })
   return () => { cancelled = true }
 },  [isAuthed])

 if (!isAuthed) return <div>Увійдіть, щоб побачити список користувачів.</div>
 if (loading) return <div>Завантаження...</div>
 if (error) return <div style={{ color: '#b00020' }}>Помилка: {error}</div>
 if (!data.length) return <div>Користувачів не знайдено.</div>

 const th = { textAlign: 'left', borderBottom: '1px solid #ddd', padding: '8px'
}
 const td = { borderBottom: '1px solid #f0f0f0', padding: '8px' }
 return <UsersTableView data={data} />
 }
```

# Додаток Б

## Конфігурація файлу збірки для github actions

```yaml
name: Build and push Docker images
on:
 push:
   branches:  ["master"]
jobs:
 build-and-push:
   runs-on: ubuntu-latest
   permissions:
     contents: read
     packages: write
   steps:
     - name: Checkout repo
       uses: actions/checkout@v4
     - name: Log in to GitHub Container Registry
       uses: docker/login-action@v3
       with:
         registry: ghcr.io
         username: ${{ github.actor }}
         password: ${{ secrets.GITHUB_TOKEN }}
     - name: Build and push backend image
       uses: docker/build-push-action@v5
       with:
         context: ./backend
         file: ./backend/Dockerfile
         push: true
         tags: ghcr.io/devman-3000/magister-backend:latest
```

# Додаток В

## Конфігурація файлу деплою для github actions

```
name: Deploy to VPS
on:
 workflow_run:
   workflows:  ["Build and push Docker images"]
   types:
     - completed
 workflow_dispatch: {}
jobs:
 deploy:
   if: ${{ github.event.workflow_run.conclusion == 'success' }}
   runs-on: ubuntu-latest
   steps:
     - name: Checkout repo
       uses: actions/checkout@v4
     - name: Start SSH agent
       uses: webfactory/ssh-agent@v0.9.0
       with:
         ssh-private-key: ${{ secrets.SSH_KEY }}
     - name: Deploy over SSH
       run: |
         ssh -o StrictHostKeyChecking=no ${{ secrets.SSH_USER }}@${{
secrets.SSH_HOST }} << 'EOF'
          cd ~/magister-ci-cd || git clone https://github.com/devman-
3000/magister-ci-cd.git && cd magister-ci-cd
          git pull
          docker compose -p magister-prod -f docker-compose.yml -f docker-
compose.prod.yml pull
          docker compose -p magister-prod -f docker-compose.yml -f docker-
compose.prod.yml up -d
```