ДЕРЖАВНИЙ УНІВЕРСИТЕТ
ІНФОРМАЦІЙНО-КОМУНІКАЦІЙНИХ ТЕХНОЛОГІЙ
НАВЧАЛЬНО-НАУКОВИЙ ІНСТИТУТ ІНФОРМАЦІЙНИХ ТЕХНОЛОГІЙ
КАФЕДРА КОМП'ЮТЕРНОЇ ІНЖЕНЕРІЇ

КВАЛІФІКАЦІЙНА РОБОТА

на тему: «СИСТЕМА АВТОМАТИЗОВАНОГО РОЗГОРТАННЯ КОНТЕЙНЕРИЗОВАНИХ ЗАСТОСУНКІВ»

на здобуття освітнього ступеня бакалавра
зі спеціальності 123 Комп'ютерна інженерія
(код, найменування спеціальності)
освітньо-професійної програми «Комп'ютерна інженерія»
(назва)

Кваліфікаційна робота містить результати власних досліджень. Використання ідей, результатів і текстів інших авторів мають посилання на відповідне джерело

_____ *Роман ІВАЩЕНКО*
(підпис) *Ім'я, ПРІЗВИЩЕ здобувача*

Виконав: здобувач вищої освіти гр. КІД-41
Роман ІВАЩЕНКО
Ім'я, ПРІЗВИЩЕ

Керівник: Наталія ЛАЩЕВСЬКА
науковий ступінь, вчене звання Ім'я, ПРІЗВИЩЕ

Рецензент:
науковий ступінь, вчене звання Ім'я, ПРІЗВИЩЕ

Київ 2026

ДЕРЖАВНИЙ УНІВЕРСИТЕТ
ІНФОРМАЦІЙНО-КОМУНІКАЦІЙНИХ ТЕХНОЛОГІЙ

Навчально-науковий інститут Інформаційних технологій

Кафедра Комп'ютерної інженерії

Ступінь вищої освіти бакалавр

Спеціальність 123 Комп'ютерна інженерія

Освітньо-професійна програма «Комп'ютерна інженерія»

ЗАТВЕРДЖУЮ
Завідувач кафедри
Комп'ютерної інженерії
_____ Наталія ЛАЩЕВСЬКА
«_____» _____ 2026 р.

ЗАВДАННЯ
НА КВАЛІФІКАЦІЙНУ РОБОТУ

Іващенко Роман Миколайович
(прізвище, ім'я, по батькові здобувача)

1. Тема кваліфікаційної роботи: «Система автоматизованого розгортання контейнеризованих застосунків»
керівник кваліфікаційної роботи: Наталія ЛАЩЕВСЬКА, кандидат технічних наук, доцент,
(Ім'я, ПРІЗВИЩЕ, науковий ступінь, вчене звання)
затверджені наказом Державного університету інформаційно-комунікаційних технологій від «20» лютого 2026р. №51
2. Строк подання кваліфікаційної роботи «14» травня 2026р.
3. Вихідні дані до кваліфікаційної роботи: науково-технічна література з автоматизації розгортання та DevOps-підходів, концепції та принципи CI/CD, архітектура та інструменти контейнеризації (Docker, Docker Compose), системи контролю версій та керування репозиторіями (Git), платформи автоматизації (GitHub Actions), методи побудови та тестування пайплайнів розгортання, підходи до забезпечення безпеки та оптимізації інфраструктури.
4. Зміст розрахунково-пояснювальної записки (перелік питань, які потрібно розробити)
    1. Теоретичні основи автоматизації розгортання контейнеризованих застосунків
    2. Проєктування та розробка системи автоматизованого розгортання на базі GitHub Actions
    3. Оптимізація конвеєра розгортання та перспективи розвитку
5. Перелік ілюстративного матеріалу: *презентація*
    1. Мета, об'єкт, предмет дослідження
    2. Актуальність роботи
    3. Аналіз існуючих рішень
    4. Загальна архітектура системи
    5. Конфігурація системи
    6. Налаштування середовища
    7. Алгоритм роботи CI/CD
    8. Демонстрація роботи
    9. Оптимізація системи

10. Аналіз ефективності
11. Перспективи розвитку
12. Висновки
13. Результати впровадження

6. Дата видачі завдання «20» лютого 2026р.

КАЛЕНДАРНИЙ ПЛАН

№ з/п	Назва етапів кваліфікаційної роботи	Строк виконання етапів роботи	Примітка
1	Збір даних	22.02-20.03.2026	Виконано
2	Обробка матеріалу	20.03-30.03.2026	Виконано
3	Розробка теоретичної частини	31.03-10.04.2026	Виконано
4	Проєктування та розробка системи автоматизованого розгортання на базі GitHub Actions	11.04-20.04.2026	Виконано
5	Висновки за результатами аналізу	21.04-26.04.2026	Виконано
6	Оформлення пояснювальної записки	26.04-28.04.2026	Виконано
7	Розробка презентації	29.04-01.05.2026	Виконано
8	Передзахист	08.05-11.05.2026	Виконано
9	Здача в деканат	08.05-14.05.2026	Виконано

Здобувач(ка) вищої освіти _____ Роман ІВАЩЕНКО
(підпис) *(Ім'я, ПРІЗВИЩЕ)*

Керівник кваліфікаційної роботи _____ Наталія ЛАЩЕВСЬКА
(підпис) *(Ім'я, ПРІЗВИЩЕ)*

РЕФЕРАТ

Текстова частина кваліфікаційної роботи на здобуття освітнього ступеня бакалавра: 72 стор., 34 рис., 5 табл., 32 джерел.

Мета роботи – мінімізація часу розгортання та підвищення надійності програмних застосунків за рахунок використання принципів CI/CD, контейнеризації та автоматизованих конвеєрів доставки.

Об'єкт дослідження – процес автоматизованого розгортання програмних застосунків у серверному середовищі.

Предмет дослідження – засоби та технології побудови CI/CD конвеєрів, контейнеризації та автоматизації розгортання із використанням Docker і GitHub Actions.

Короткий зміст роботи: У роботі представлено теоретичні основи автоматизації розгортання застосунків, включаючи концепції CI/CD, життєвий цикл програмного забезпечення, принципи контейнеризації та використання систем контролю версій. Розглянуто архітектуру Docker, підходи до оркестрації контейнерів через Docker Compose, а також моделі організації пайплайнів і механізми забезпечення безпеки, зокрема управління доступами та секретами у процесі автоматизації.

Особливу увагу приділено розробці системи автоматизованого розгортання із використанням GitHub Actions та Self-hosted runner, налаштуванню середовища виконання, реалізації алгоритмів збірки образів і безшовного оновлення контейнерів, а також тестуванню працездатності пайплайну. Додатково розглянуто методи оптимізації, питання безпеки, ефективності та можливості масштабування інфраструктури з переходом до кластерних рішень.

КЛЮЧОВІ СЛОВА: CI/CD, КОНТЕЙНЕРИЗАЦІЯ, DOCKER, DOCKER COMPOSE, GITHUB ACTIONS, SELF-HOSTED RUNNER, АВТОМАТИЗОВАНЕ РОЗГОРТАННЯ, DEVOPS, ІНФРАСТРУКТУРА ЯК КОД, ПАЙПЛАЙН.

ЗМІСТ

ВСТУП 8
1. ТЕОРЕТИЧНІ ОСНОВИ АВТОМАТИЗАЦІЇ РОЗГОРТАННЯ КОНТЕЙНЕРИЗОВАНИХ ЗАСТОСУНКІВ 9
1.1 Концепції неперервної інтеграції та розгортання (CI/CD) 9
1.2 Контейнеризація як стандарт інфраструктури 18
1.3 Системи контролю версій у процесі автоматизації 24
1.4 Безпека автоматизованих конвеєрів та управління секретами 29
2. 2.ПРОЄКТУВАННЯ ТА РОЗРОБКА СИСТЕМИ АВТОМАТИЗОВАНОГО РОЗГОРТАННЯ НА БАЗІ GITHUB ACTIONS 36
2.1 Архітектура пайплайну CI/CD 36
2.2 Налаштування цільового середовища та конфігурація Actions 43
2.3 Алгоритми функціонування конвеєра розгортання 51
2.4 Тестування процесу автоматичного розгортання 57
3. ОПТИМІЗАЦІЯ КОНВЕЄРА РОЗГОРТАННЯ ТА ПЕРСПЕКТИВИ РОЗВИТКУ 63
3.1 Методи оптимізації процесів CI/CD та вирішення проблем безпеки 63
3.2 Аналіз ефективності розробленої системи 69
3.3 Вектори масштабування інфраструктури 76
ВИСНОВКИ 79
ПЕРЕЛІК ВИКОРИСТАНИХ ДЖЕРЕЛ 80
Додаток А. Вихідний код файлу app.js 82
Додаток Б. Конфігурація залежностей package.json 83
Додаток В. Сценарій збірки Dockerfile 84
Додаток Г. Конфігурація середовища docker-compose.yml 85
Додаток Д. Пайплайн розгортання .github/workflows/deploy.yml 86
Додаток Е. ДЕМОНСТРАЦІЙНІ МАТЕРІАЛИ (Презентація) 87

ВСТУП

Розвиток програмного забезпечення супроводжується зростанням вимог до швидкості, стабільності та якості розгортання застосунків, що зумовлює впровадження автоматизації на всіх етапах життєвого циклу. У цьому контексті концепції неперервної інтеграції та розгортання формують основу безперервної доставки змін, забезпечуючи швидке виявлення помилок, узгодженість середовищ і скорочення часу між розробкою та впровадженням. Взаємодія систем контролю версій, тригерів і автоматизованих сценаріїв створює модель, у якій зміни коду безпосередньо впливають на розгортання.

Важливим технологічним фундаментом виступає контейнеризація, яка дозволяє ізолювати застосунки разом із залежностями та гарантувати відтворюваність виконання у різних середовищах. Використання контейнерних платформ і механізмів оркестрації спрощує управління багатокомпонентними системами та забезпечує гнучкість масштабування й оновлення сервісів. У поєднанні з підходом інфраструктури як коду формується стандартизований спосіб опису та відтворення інфраструктури.

Важливим аспектом є забезпечення безпеки автоматизованих конвеєрів, що включає контроль доступу, ізоляцію середовищ виконання та захист облікових даних. Використання централізованих механізмів зберігання секретів, обмеження прав доступу та безпечних мережевих моделей дозволяє зменшити ризики несанкціонованого втручання. У результаті формується основа для побудови систем, здатних працювати в умовах постійних змін і високих навантажень.

У практичній частині передбачається реалізація системи автоматизованого розгортання контейнеризованого застосунку з використанням CI/CD конвеєра, що базується на інтеграції репозиторію, механізму виконання та контейнерної платформи. Планується налаштування середовища, автоматизація збірки образів, безперервне оновлення контейнерів і впровадження механізмів тестування та контролю, що дозволить продемонструвати повний цикл доставки.

1. ТЕОРЕТИЧНІ ОСНОВИ АВТОМАТИЗАЦІЇ РОЗГОРТАННЯ КОНТЕЙНЕРИЗОВАНИХ ЗАСТОСУНКІВ

1.1 Концепції неперервної інтеграції та розгортання (CI/CD)

Розробка сучасного програмного забезпечення характеризується високою динамікою змін, що обумовлює необхідність частого випуску нових версій, швидкого внесення виправлень та постійного підтримання стабільності роботи системи, у таких умовах традиційні підходи до збірки та розгортання застосунків виявляються неефективними, оскільки потребують значних часових витрат, залежать від людського фактору та ускладнюють масштабування інфраструктури, що особливо критично для проєктів із високими вимогами до доступності та швидкості доставки функціоналу. Автоматизація процесів розробки та розгортання поступово стала невід'ємною частиною сучасних IT-інфраструктур, дозволяючи стандартизувати операції, підвищити надійність систем та скоротити проміжок часу між внесенням змін у код і їх появою у робочому середовищі, що особливо актуально в умовах використання контейнеризації, де застосунки упаковуються у стандартизовані середовища виконання.

У межах автоматизації ключовим підходом виступає концепція CI/CD (Continuous Integration та Continuous Deployment), яка передбачає безперервну інтеграцію змін у спільний кодовий репозиторій із подальшим автоматизованим тестуванням, збіркою та розгортанням, формуючи єдиний безперервний конвеєр доставки програмного забезпечення. Основна ідея полягає у створенні такого процесу, за якого кожна зміна коду, починаючи з моменту її додавання до репозиторію, автоматично проходить усі необхідні етапи перевірки та підготовки до розгортання, що дозволяє мінімізувати ручне втручання, забезпечити швидке виявлення помилок, підвищити стабільність релізів та значно скоротити час доставки функціоналу кінцевому користувачу. Такий підхід особливо ефективно

поєднується з контейнеризацією, оскільки дозволяє автоматично створювати, тестувати та розгортати контейнерні образи у стандартизованому середовищі, забезпечуючи передбачуваність виконання незалежно від цільової інфраструктури [1-2].

Життєвий цикл розробки та розгортання ПЗ. Життєвий цикл програмного забезпечення (Software Development Life Cycle, SDLC) визначає послідовність етапів, через які проходить програмний продукт від початкової ідеї до етапу експлуатації та підтримки, формуючи логічну структуру організації процесу розробки. Класичний життєвий цикл включає аналіз вимог, під час якого визначаються функціональні та нефункціональні характеристики системи, архітектура, технологічний стек і структура застосунку, після чого виконується етап розробки, що полягає у написанні програмного коду відповідно до визначених специфікацій. Наступним етапом виступає тестування, спрямоване на перевірку коректності роботи застосунку, виявлення помилок та забезпечення відповідності вимогам, далі відбувається збірка, у процесі якої формується виконуваний продукт або контейнерний образ, придатний до розгортання [3].

Після завершення збірки здійснюється розгортання застосунку у цільове середовище, яким може виступати фізичний сервер, віртуальна машина або хмарна інфраструктура, де програмне забезпечення стає доступним для кінцевого використання, завершальним етапом є експлуатація та підтримка, що включає моніторинг роботи системи, оновлення, виправлення помилок та вдосконалення функціоналу. У традиційних підходах, зокрема у каскадній моделі (Waterfall), ці етапи виконуються послідовно, а тестування відбувається переважно наприкінці процесу, що призводить до накопичення помилок, ускладнення їх виправлення та затримок у випуску нових версій (див. рис. 1.1).

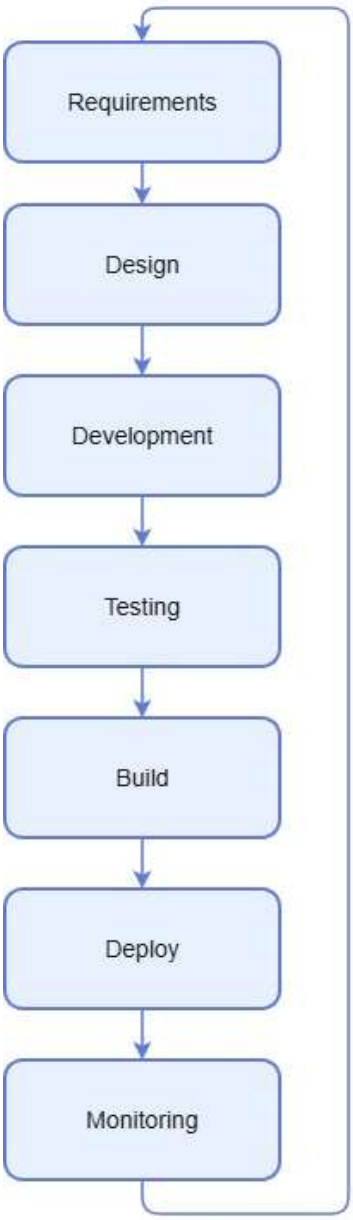


Рисунок 1.1 – Життєвий цикл розробки та розгортання програмного забезпечення

Застосування CI/CD трансформує життєвий цикл у безперервний процес, де кожна зміна коду автоматично проходить перевірку, збірку та тестування, а розгортання може виконуватися безпосередньо після успішного проходження всіх етапів, забезпечуючи циклічність, гнучкість та швидкість оновлень. Такий підхід дозволяє зменшити ризики, пов’язані з інтеграцією змін, підвищити якість програмного продукту та адаптувати процес розробки до вимог сучасних

динамічних систем, у яких швидкість реагування на зміни є критичним фактором успіху.

Основні принципи Continuous Integration та Continuous Deployment. Основні принципи CI/CD формуються на основі двох взаємопов'язаних підходів — Continuous Integration та Continuous Deployment, які разом утворюють єдину безперервну систему автоматизованої розробки та розгортання програмного забезпечення, де кожна зміна коду проходить чітко визначений і автоматизований шлях від моменту внесення до запуску у середовищі виконання. Continuous Integration передбачає регулярне та часте об'єднання змін у основну гілку репозиторію, що дозволяє уникати накопичення великих обсягів несумісних змін і суттєво зменшує ризик виникнення конфліктів під час інтеграції, при цьому розробники додають невеликі частини функціоналу, які одразу перевіряються у спільному середовищі [4].

Ключовою характеристикою цього підходу виступають часті коміти, за яких зміни вносяться у репозиторій регулярно, що забезпечує постійний контроль стану системи, одночасно з цим кожен коміт автоматично ініціює процес збірки застосунку, у результаті чого формується актуальна версія програмного продукту без необхідності ручного втручання, після чого запускається етап автоматичного тестування, який дозволяє перевірити коректність роботи нових змін і виявити помилки на ранніх стадіях розробки, що значно знижує витрати на їх виправлення. У межах такого процесу забезпечується швидкий зворотний зв'язок, оскільки результати збірки та тестування стають доступними майже одразу після внесення змін, що дозволяє оперативно реагувати на проблеми та підтримувати стабільність кодової бази.

Continuous Deployment логічно продовжує процеси, розпочаті на етапі інтеграції, і передбачає автоматичне розгортання застосунку у середовищі виконання після успішного проходження всіх перевірок, у такому підході відсутнє ручне втручання під час деплою, що дозволяє усунути вплив людського фактору та забезпечити передбачуваність процесу розгортання. Автоматизація деплою забезпечує доставку змін безпосередньо на сервер або інше середовище виконання,

при цьому кожен етап пайплайну має бути надійним і відтворюваним, що формує вимогу стабільності всієї системи, у межах якої будь-який запуск має давати однаковий результат за однакових умов (див. рис. 1.2) [4].

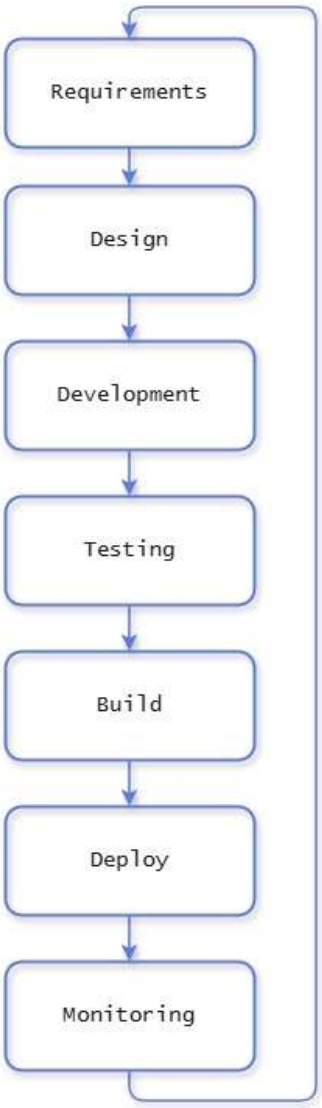


Рисунок 1.2 – Послідовність виконання етапів Continuous Integration

Важливою властивістю таких процесів виступає ідемпотентність, яка передбачає, що повторний запуск пайплайну не змінює кінцевий результат, забезпечуючи однаковий стан системи незалежно від кількості виконань, додатково застосовується контроль версій, що дозволяє швидко повернутися до попереднього стабільного стану у разі виникнення помилок або збоїв. У межах CD

розрізняються два підходи — Continuous Delivery та Continuous Deployment, де перший передбачає можливість автоматичної підготовки до розгортання із необхідністю підтвердження, а другий реалізує повністю автоматизований процес без участі людини, що дозволяє досягти максимальної швидкості доставки змін (див. рис. 1.3) [4].

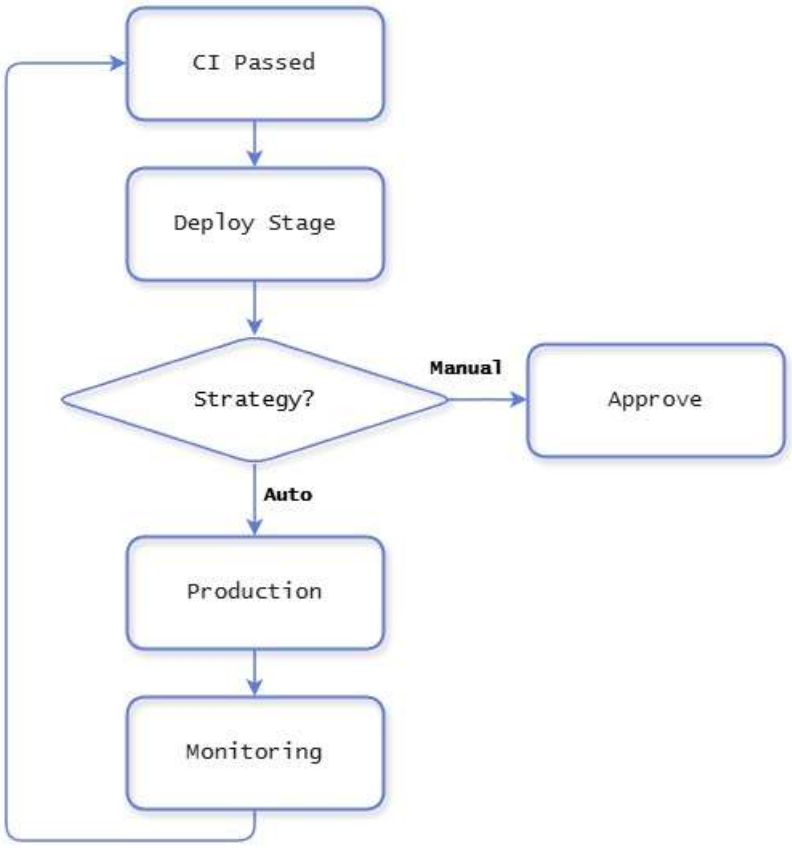


Рисунок 1.3 – Принцип роботи Continuous Deployment у середовищі автоматизації

Узгоджена робота CI та CD формує єдиний пайплайн, у якому внесення змін у репозиторій автоматично запускає процеси збірки та тестування, після успішного завершення яких ініціюється розгортання застосунку, у разі виникнення помилок система одразу повідомляє про це, що дозволяє швидко локалізувати проблему та внести необхідні виправлення, у той час як успішне проходження всіх етапів призводить до автоматичного оновлення застосунку у цільовому середовищі. У контексті контейнеризації така модель набуває додаткової ефективності, оскільки на етапі інтеграції формується контейнерний образ, який проходить перевірку та

тестування, а на етапі розгортання використовується для запуску у середовищі оркестрації, наприклад через Docker Compose, що забезпечує ідентичність середовищ та передбачуваність виконання.

Інструментарій автоматизації (GitHub Actions, GitLab CI). Реалізація процесів CI/CD потребує використання спеціалізованих інструментів, які дозволяють:

— описувати автоматизацію у вигляді конфігураційних файлів;

— керувати виконанням пайплайнів;

— інтегрувати їх із системами контролю версій.

При цьому до таких інструментів висуваються вимоги:

— підтримки автоматичних тригерів;

— можливості виконання сценаріїв;

— інтеграції з репозиторіями;

— роботи з контейнерами;

— управління змінними середовища та секретами.

Одним із найбільш поширених рішень є GitHub Actions, який інтегрований безпосередньо у платформу GitHub та дозволяє організовувати процеси автоматизації без використання зовнішніх сервісів [5].

Конфігурація у GitHub Actions здійснюється за допомогою YAML-файлів, що розміщуються у спеціальній директорії репозиторію, де описується структура пайплайну, події запуску та послідовність виконання команд, запуск процесів відбувається на основі подій, таких як додавання змін у репозиторій або створення "pull request", що дозволяє автоматично реагувати на будь-які зміни у коді. Пайплайн складається з окремих завдань, які об’єднуються у логічні блоки, а кожне завдання включає набір кроків, що виконуються послідовно, забезпечуючи гнучкість та контроль над процесом виконання [5].

Важливою особливістю виступає використання так званих runner’ів, які забезпечують середовище виконання задач, причому вони можуть бути як хмарними, так і розгорнутими на власній інфраструктурі, що дозволяє адаптувати систему під конкретні вимоги, включаючи інтеграцію з різними середовищами

виконання. Додатково підтримується використання готових модулів, що спрощують реалізацію типових задач, таких як:

— збірка застосунку;

— запуск тестів;

— розгортання контейнерів.

У межах практичної реалізації GitHub Actions використовується для автоматичного підключення до серверної інфраструктури, виконання команд розгортання та оновлення застосунку після внесення змін у репозиторій, що дозволяє забезпечити повністю автоматизований цикл доставки програмного забезпечення без ручного втручання. Альтернативним рішенням виступає GitLab CI/CD, який інтегрований у платформу GitLab та надає подібний функціонал із підтримкою конфігурації через YAML-файли, визначенням етапів пайплайну та можливістю гнучкого управління виконанням задач (див. рис. 1.4) [5-6].

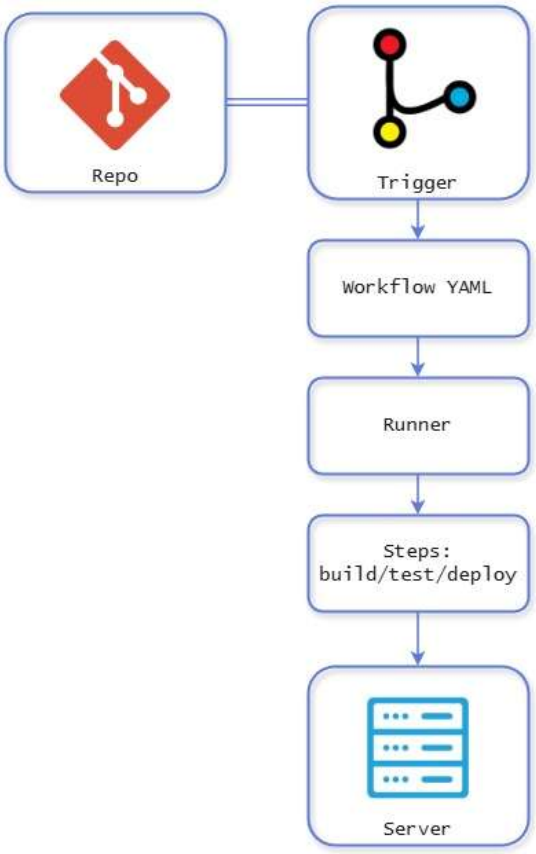


Рисунок 1.4 – Взаємодія репозиторію та пайплайну розгортання

GitLab CI/CD забезпечує глибоку інтеграцію всіх етапів життєвого циклу програмного забезпечення в межах єдиного середовища, підтримує роботу з контейнерними реєстрами та дозволяє розгортати власну інфраструктуру для виконання задач, що розширює можливості налаштування та контролю (див. табл. 1.1).

Таблиця 1.1 – Порівняння інструментів автоматизації CI/CD (GitHub Actions, GitLab CI)

Критерій	GitHub Actions	GitLab CI/CD
Тип середовища	Хмарне та підтримка self-hosted	Хмарне та можливість повністю локального розгортання
Інтеграція	Глибока інтеграція з GitHub	Повна інтеграція в екосистему GitLab
Конфігурація	YAML (workflow)	YAML (.gitlab-ci.yml)
Тригери	Push, pull request, schedule, events	Push, merge request, schedule
Runner-и	GitHub-hosted та self-hosted	Shared та self-hosted
Робота з контейнерами	Вбудована підтримка Docker	Вбудована підтримка Docker та registry
Управління секретами	GitHub Secrets	GitLab CI/CD Variables
Масштабування	Автоматичне (хмарне)	Автоматичне та власна інфраструктура
Обмеження	Ліміти часу та ресурсів	Залежить від тарифу або інфраструктури

Використання подібних інструментів дозволяє досягти високого рівня автоматизації, забезпечити стабільність процесів розробки та розгортання, а також мінімізувати ризики, пов’язані з людським фактором, формуючи надійну основу для побудови ефективних систем доставки програмного забезпечення, особливо у поєднанні з контейнеризацією та сучасними підходами до управління інфраструктурою [6].

1.2 Контейнеризація як стандарт інфраструктури

Контейнеризація стала одним із базових підходів до побудови, розгортання та масштабування систем, особливо в контексті автоматизованих CI/CD пайплайнів, де важливо забезпечити швидкість, стабільність і передбачуваність виконання застосунків, при цьому сам підхід дозволяє ізолювати програмне забезпечення разом із усіма його залежностями у окреме середовище виконання, що забезпечує однакову поведінку незалежно від типу інфраструктури, на якій відбувається запуск, включаючи локальні машини, сервери або хмарні платформи. Контейнеризація виступає логічним продовженням концепції інфраструктури як коду, оскільки дозволяє описувати не лише конфігурацію серверів, але й саме середовище виконання застосунку у вигляді декларативних інструкцій, що значно спрощує процеси розгортання, тестування та підтримки, забезпечуючи відтворюваність усіх етапів життєвого циклу програмного забезпечення.

Широке поширення контейнеризації стало можливим завдяки платформі Docker, яка фактично сформувала стандарт у цій галузі та надала інструменти для створення, зберігання і запуску ізольованих середовищ виконання у вигляді контейнерів, що характеризуються низьким споживанням ресурсів і високою швидкістю запуску порівняно з традиційними віртуальними машинами. Контейнери забезпечують портативність, оскільки можуть запускатися у будь-якому середовищі, де встановлено Docker, ізоляцію, яка дозволяє кожному застосунку функціонувати незалежно від інших, швидкість розгортання, що досягається за рахунок використання спільного ядра операційної системи, масштабованість через можливість швидкого створення нових екземплярів застосунку, а також відтворюваність, яка гарантує ідентичність умов виконання на етапах розробки, тестування та експлуатації, у результаті чого контейнеризація стає основою для побудови стабільних і передбачуваних систем автоматизованого розгортання [7].

Архітектура та базові компоненти системи Docker. Архітектура Docker побудована за принципом клієнт-серверної моделі, у межах якої взаємодія між користувачем і системою відбувається через клієнтський інтерфейс, який передає команди до серверної частини, що відповідає за їх виконання, при цьому додатково використовується система реєстрів для зберігання образів, що забезпечує можливість централізованого управління та розповсюдження контейнерів як у локальному, так і у віддаленому середовищі. Така архітектура дозволяє гнучко керувати процесами створення та запуску контейнерів, забезпечуючи доступ до них як з боку розробників, так і з боку інженерів, відповідальних за інфраструктуру.

Ключовим компонентом системи виступає Docker Engine, який є основною платформою для роботи з контейнерами та відповідає за їх створення, запуск, зупинку і управління, а також за обробку команд, отриманих від користувача, при цьому внутрішньо він складається з декількох складових, включаючи демон-процес, інтерфейс прикладного програмування та механізми взаємодії з операційною системою, що разом забезпечує стабільне функціонування всієї системи. Важливим елементом виступає Docker Image, який являє собою шаблон для створення контейнера і містить у собі застосунок разом із усіма необхідними залежностями, при цьому образ є незмінним і формується на основі інструкцій, описаних у спеціальному конфігураційному файлі, що дозволяє гарантувати однаковість середовища при кожному запуску [8-9].

Контейнер, у свою чергу, є запущеним екземпляром образу, який функціонує як ізольований процес із власною файловою системою, мережею та набором залежностей, при цьому використання спільного ядра операційної системи дозволяє значно зменшити витрати ресурсів і забезпечити високу швидкість роботи, що вигідно відрізняє контейнери від віртуальних машин. Для створення образів використовується Dockerfile — текстовий файл, який описує послідовність дій для формування середовища виконання, включаючи [8]:

— вибір базового образу;

— встановлення залежностей;

— копіювання файлів;

— визначення команди запуску.

У результаті чого забезпечується повністю автоматизований і відтворюваний процес побудови образів.

Додатковим компонентом виступає Docker Registry, який використовується для зберігання та розповсюдження образів, причому він може бути як публічним, наприклад Docker Hub, так і приватним, що дозволяє організовувати контроль доступу та управління версіями образів у межах конкретного проєкту. Важливу роль також відіграють механізми мережевої взаємодії, які дозволяють контейнерам обмінюватися даними через різні типи мереж, а також система томів, що забезпечує збереження даних поза межами контейнера, дозволяючи використовувати їх повторно або передавати між різними екземплярами застосунків, у результаті чого забезпечується гнучкість та стабільність роботи системи навіть при перезапуску контейнерів [8].

Процес роботи Docker передбачає створення Dockerfile, на основі якого формується образ, після чого він може бути збережений у реєстрі або використаний локально, далі з цього образу запускається контейнер, який взаємодіє з іншими контейнерами через мережу та використовує необхідні ресурси для виконання задач, що дозволяє стандартизувати процес розгортання та уникнути проблем сумісності між різними середовищами виконання (див. рис. 1.5).

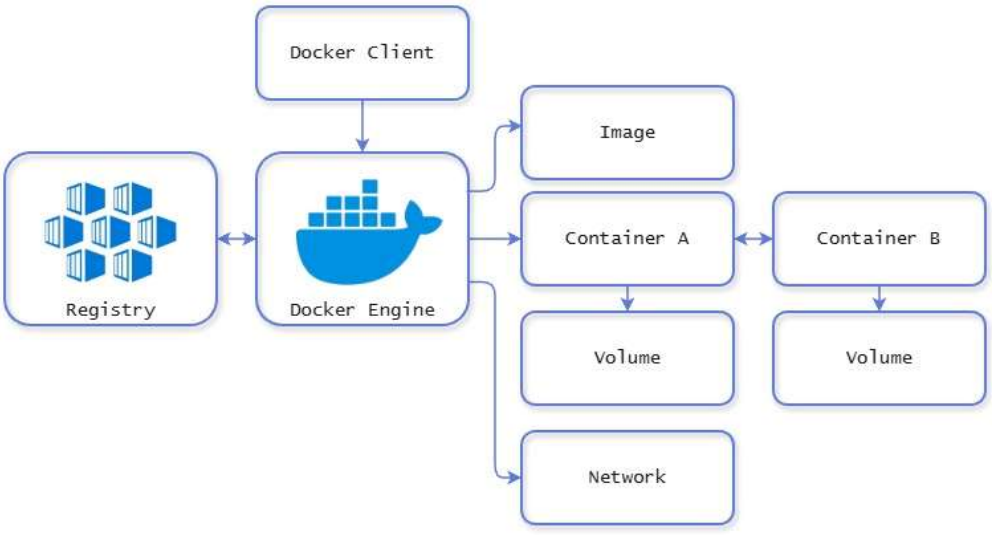


Рисунок 1.5 – Взаємодія контейнерів у середовищі виконання

Оркестрація багатоконтейнерних застосунків через Docker Compose. Сучасні програмні застосунки у більшості випадків складаються не з одного компонента, а з набору взаємопов'язаних сервісів, серед яких можуть бути:

— веб-сервери;

— бази даних;

— кешуючі системи;

— Інші допоміжні елементи інфраструктури.

При цьому ручне керування такими середовищами стає складним і неефективним навіть за умов наявності контейнеризації, оскільки виникає необхідність координувати запуск, зупинку, взаємодію та залежності між окремими контейнерами, що ускладнює процес розгортання і підвищує ризик помилок, у зв'язку з чим виникає потреба в інструменті, здатному автоматизувати ці процеси та забезпечити узгоджену роботу всіх компонентів системи, яким виступає docker compose як засіб базової оркестрації для багатоконтейнерних застосунків.

Поняття оркестрації у цьому контексті означає автоматизоване управління життєвим циклом кількох контейнерів одночасно, включаючи:

— запуск;

— зупинку;

— перезапуск;

— масштабування;

— взаємодію.

Все це дозволяє розглядати систему не як набір окремих контейнерів, а як єдину узгоджену структуру, у межах якої всі компоненти працюють синхронно відповідно до заданої конфігурації, при цьому Docker Compose забезпечує необхідний рівень такої оркестрації переважно для локальних або невеликих серверних середовищ, де відсутня потреба у складніших системах управління [10].

Основою роботи Docker Compose виступає конфігураційний файл формату YAML, зазвичай з назвою "docker-compose.yml", у якому декларативно описується структура застосунку, включаючи [10]:

— перелік сервісів;

— джерела образів для кожного контейнера;

— параметри мережевої взаємодії;

— підключення сховищ даних;

— залежності між окремими компонентами.

При цьому така декларативність дозволяє зберігати конфігурацію у вигляді коду, що забезпечує:

— повторне використання;

— версіонування;

— інтеграцію.

Все це виконується у процесі автоматизації [10].

У межах цього файлу визначаються сервіси як логічні одиниці застосунку, для кожного з яких задається спосіб створення контейнера через використання готового образу або його побудову, мережеві порти, що зв’язують контейнер із зовнішнім середовищем, а також томи для збереження даних поза межами контейнера, що забезпечує їхню персистентність навіть у разі перезапуску, додатково визначаються залежності між сервісами, які дозволяють встановити порядок їх запуску, що є важливим у випадках, коли один компонент не може функціонувати без попереднього запуску іншого (див. рис. 1.6).

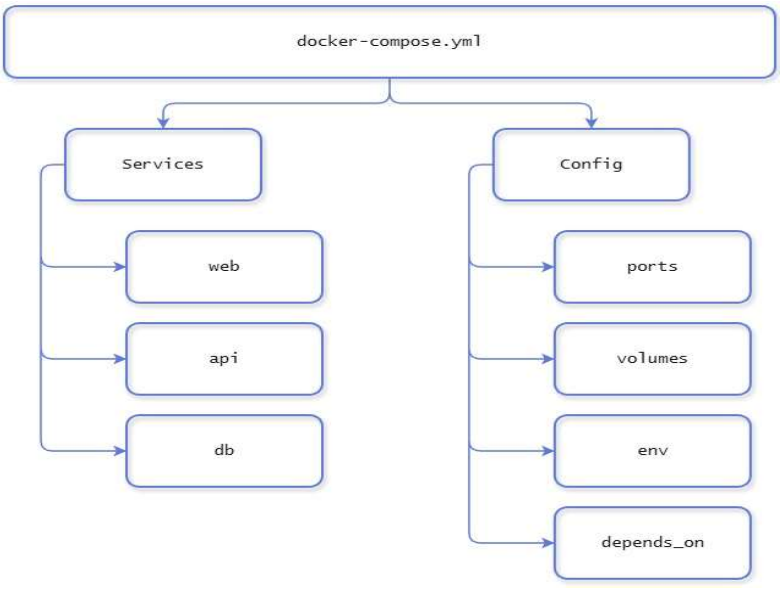


Рисунок 1.6 – Структура docker-compose.yml для багатоконтейнерного застосунку

Функціональні можливості Docker Compose дозволяють здійснювати одночасний запуск усіх сервісів, визначених у конфігурації, за допомогою єдиної команди, що значно спрощує процес розгортання, при цьому підтримується режим роботи у фоновому режимі, який дозволяє системі функціонувати без прив'язки до активної сесії користувача, управління життєвим циклом контейнерів охоплює також їх зупинку, перезапуск та повне видалення, що забезпечує контроль над станом системи у будь-який момент часу [10].

Окрему увагу привертає автоматичне створення мережі, у межах якої всі сервіси можуть взаємодіяти між собою за іменами, визначеними у конфігурації, що усуває необхідність ручного налаштування мережевих параметрів і значно спрощує комунікацію між компонентами, при цьому Docker Compose за замовчуванням створює ізольоване середовище для таких взаємодій, що підвищує безпеку та передбачуваність роботи системи [10].

Масштабування сервісів також підтримується шляхом запуску кількох екземплярів одного й того ж контейнера, що дозволяє розподіляти навантаження та підвищувати доступність застосунку, при цьому кожному екземпляру автоматично присвоюється унікальний ідентифікатор, що забезпечує можливість їх подальшого управління, а інтеграція з процесами безперервної інтеграції та доставки дозволяє використовувати Docker Compose як частину автоматизованих пайплайнів, у межах яких відбувається збірка нових образів та їх розгортання без ручного втручання.

Практичне застосування Docker Compose у таких процесах передбачає зміну вихідного коду застосунку, після чого автоматизована система ініціює побудову нового образу та перезапуск контейнерів із оновленою версією, що дозволяє реалізувати безперервне оновлення сервісів без простоїв або складних процедур міграції, у результаті чого досягається високий рівень автоматизації та стабільності.

Узагальнюючи, контейнеризація у поєднанні з інструментами оркестрації забезпечує формування стандартизованого підходу до побудови інфраструктури, де всі компоненти системи описуються у вигляді коду, автоматично розгортаються та взаємодіють між собою відповідно до заданих правил, а використання Docker

Compose розширює можливості роботи з контейнерами, дозволяючи ефективно керувати багатокомпонентними застосунками та інтегрувати ці процеси у сучасні практики автоматизації розробки і розгортання програмного забезпечення.

1.3 Системи контролю версій у процесі автоматизації

В процесі розробки програмного забезпечення системи контролю версій стали невід'ємною складовою не лише збереження вихідного коду, а й організації автоматизованих процесів розгортання, тестування та підтримки застосунків, при цьому вони забезпечують централізоване управління змінами, конфігураціями середовища та сценаріями автоматизації, що дозволяє інженерам координувати взаємодію між розробкою та експлуатацією систем без необхідності ручного втручання на кожному етапі. У таких умовах система контролю версій перетворюється на основу, навколо якої будується вся логіка автоматизації, включаючи інтеграцію з інструментами безперервної інтеграції та доставки, що дозволяє реалізувати повністю автоматизований життєвий цикл змін, внесених у програмний продукт.

Найбільш поширеним інструментом у цій сфері виступає Git, який став фактичним стандартом у розробці програмного забезпечення завдяки своїй гнучкості, швидкості та можливостям розподіленої роботи, при цьому його застосування виходить далеко за межі простого зберігання коду, оскільки він використовується як тригер для запуску автоматизованих процесів, де кожна зміна у репозиторії може ініціювати виконання CI/CD пайплайнів, що забезпечує безперервну інтеграцію та доставку змін у середовище виконання. У такій моделі будь-яка дія розробника, пов'язана з фіксацією змін, автоматично стає частиною більш широкого процесу, який включає перевірку, збірку, тестування та розгортання застосунку, що значно підвищує ефективність та знижує ризик помилок.

Системи контролю версій забезпечують збереження повної історії змін, що дозволяє відстежувати еволюцію проєкту, аналізувати внесені модифікації та при необхідності повертатися до попередніх станів системи, при цьому кожна зміна супроводжується інформацією про автора, час внесення та зміст, що створює прозору та контрольовану модель роботи з кодом. Паралельно забезпечується можливість спільної роботи кількох розробників, які можуть працювати над різними частинами проєкту незалежно один від одного, використовуючи механізми розгалуження та злиття змін, що дозволяє уникати конфліктів і підтримувати стабільність основної гілки розробки.

Роль Git у сучасній концепції інфраструктури як коду (IaC). Концепція інфраструктури як коду передбачає опис усіх елементів інфраструктури у вигляді програмного коду, що дозволяє відмовитися від ручного налаштування серверів і перейти до декларативного підходу, у межах якого конфігурація системи визначається через сценарії та файли конфігурації, що зберігаються у системі контролю версій, при цьому GitHub часто використовується як централізоване середовище для зберігання таких конфігурацій та управління ними. Такий підхід дозволяє розглядати інфраструктуру як частину програмного продукту, що підлягає тим самим принципам розробки, тестування та контролю, що й основний код застосунку [11].

У традиційних підходах налаштування серверів виконувалося вручну системними адміністраторами, що призводило до непослідовності конфігурацій, складності масштабування та високої ймовірності помилок, при цьому відсутність єдиного джерела істини ускладнювала відтворення середовища у разі необхідності, у той час як використання Git дозволяє зберігати всі конфігураційні файли, сценарії автоматизації, Dockerfile та інші елементи інфраструктури в одному репозиторії, що створює єдину точку контролю та забезпечує узгодженість усіх компонентів системи [10-11].

У такій моделі Git виступає як єдине джерело істини, де кожна зміна інфраструктури або застосунку фіксується у вигляді коміту, що дозволяє відстежувати всі модифікації та забезпечує можливість повернення до попередніх

станів, при цьому прозорість історії змін дозволяє аналізувати причини виникнення помилок та ефективно їх усувати. Додатково забезпечується можливість проведення рев'ю змін перед їх інтеграцією у основну гілку, що підвищує якість коду та зменшує ризик внесення некоректних змін у робочу систему (див. рис. 1.7) [11-12].

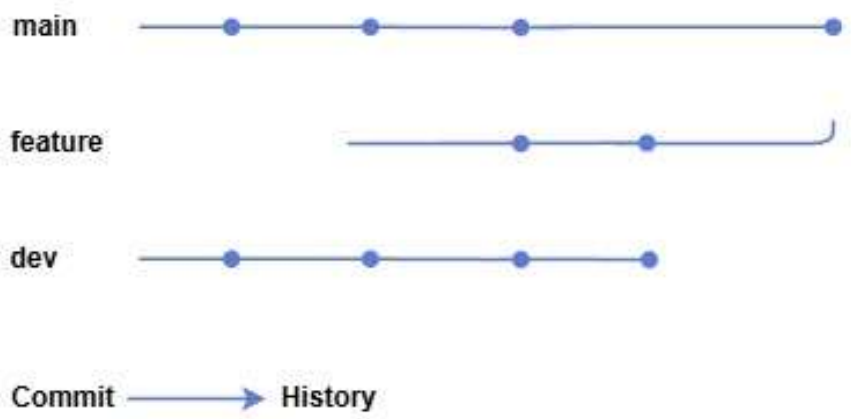


Рисунок 1.7 – Модель роботи системи контролю версій Git

Інтеграція Git із системами безперервної інтеграції та доставки дозволяє автоматизувати процеси розгортання, де кожен коміт або оновлення у репозиторії може запускати відповідний пайплайн, що виконує необхідні дії без участі людини, включаючи збірку образів, тестування та оновлення застосунків у середовищі виконання, при цьому такий підхід забезпечує високу швидкість доставки змін та їхню узгодженість із поточним станом системи. У результаті формується модель, у якій система контролю версій стає центральним елементом управління як кодом застосунку, так і всією інфраструктурою, забезпечуючи автоматизацію, відтворюваність та контрольованість усіх процесів, пов'язаних із розробкою та експлуатацією програмного забезпечення.

Стратегії розгалуження (Branching strategies) та тригери пайплайнів. Ефективне використання Git передбачає не лише збереження коду та історії змін, а й чітко визначену структуру розгалуження, яка формує основу процесів розробки, тестування та розгортання програмного забезпечення, при цьому стратегія

розгалуження являє собою набір правил і підходів, що визначають порядок створення гілок, їх використання, об’єднання змін та момент інтеграції результатів у основну гілку, яка виступає стабільною версією системи, у межах якої зберігається перевірений та готовий до використання код. Така організація дозволяє підтримувати порядок у процесі розробки, мінімізувати конфлікти між змінами, а також забезпечити контроль якості перед потраплянням нових функцій у стабільне середовище.

У практиці розробки використовується декілька основних стратегій розгалуження, кожна з яких має свої особливості та сферу застосування, при цьому класичний підхід, відомий як GitFlow, передбачає наявність кількох типів гілок, серед яких основна гілка містить стабільну версію продукту, окрема гілка використовується для активної розробки, додаткові гілки створюються для реалізації нових функцій, підготовки релізів та швидкого виправлення помилок, що дозволяє чітко структурувати процес розробки та контролювати етапи підготовки змін, хоча така модель може ускладнювати роботу через велику кількість гілок і додаткові операції злиття [13].

Більш спрощений підхід представлений моделлю GitHub Flow, яка широко застосовується у сучасних проєктах, де основна гілка залишається центральною точкою, а всі нові зміни розробляються у короткоживучих гілках, після чого проходять перевірку та об’єднуються з основною гілкою, що значно спрощує процес і добре інтегрується з автоматизованими пайплайнами, дозволяючи швидко доставляти зміни у робоче середовище без складних процедур підготовки релізів. Ще одним підходом виступає "trunk-based development", у межах якого більшість змін вносяться безпосередньо в основну гілку або через дуже короткочасні гілки, що мінімізує ризик конфліктів і забезпечує швидку інтеграцію, однак потребує високого рівня автоматизованого тестування для гарантування стабільності [13-14].

Організація гілок тісно пов’язана з механізмами автоматизації, оскільки кожна дія у репозиторії може виступати тригером для запуску CI/CD пайплайнів, де тригери визначають умови, за яких автоматизовані процеси починають

виконання, при цьому найпоширенішим варіантом є запуск пайплайну після внесення змін у репозиторій, що дозволяє автоматично виконувати збірку, тестування та розгортання застосунку одразу після оновлення коду. Іншим поширеним сценарієм виступає запуск процесів під час створення запиту на об’єднання змін, що дає змогу перевірити коректність коду перед його інтеграцією у стабільну гілку, а також існує можливість ручного запуску, яка використовується для виконання специфічних задач або контрольованих розгортань [14].

Взаємозв’язок між стратегіями розгалуження та тригерами автоматизації дозволяє будувати гнучкі сценарії розгортання, де різні гілки можуть відповідати різним середовищам виконання [13-14]:

1. основна гілка може використовуватися для розгортання у продакшн-середовище;
2. гілка розробки — для тестового середовища;
3. гілки з новими функціями — лише для локального тестування або перевірки змін.

У результаті чого забезпечується розмежування середовищ та контроль якості на кожному етапі життєвого циклу застосунку.

Інтеграція таких підходів із інструментами автоматизації, зокрема GitHub Actions, дозволяє задавати правила запуску пайплайнів безпосередньо у конфігураційних файлах, де визначаються події, що ініціюють виконання сценаріїв, а також перелік дій, які мають бути виконані, включаючи тестування, збірку та розгортання, при цьому така автоматизація значно зменшує вплив людського фактора та підвищує швидкість доставки змін (див. рис. 1.8).

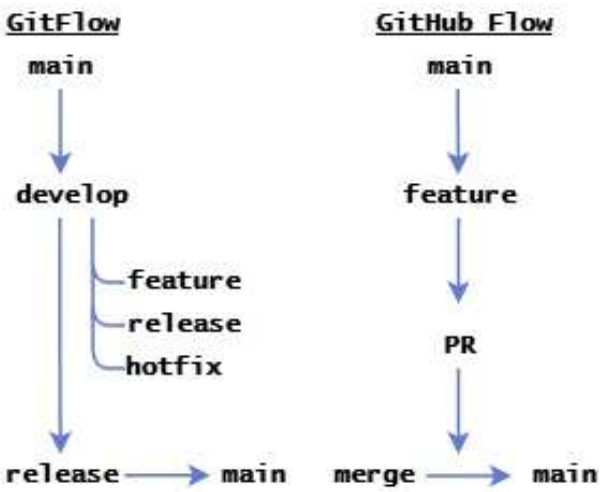


Рисунок 1.8 – Основні стратегії розгалуження (Git Flow, GitHub Flow)

У підсумку використання стратегій розгалуження у поєднанні з тригерами автоматизованих процесів формує структурований підхід до організації розробки, який забезпечує можливість:

— паралельної роботи;

— контроль якості;

— зменшення кількості конфліктів;

— стабільність системи.

При цьому одночасно створюючи основу для повністю автоматизованого розгортання, що відповідає сучасним вимогам до побудови ефективних DevOps-процесів.

1.4 Безпека автоматизованих конвеєрів та управління секретами

Автоматизоване розгортання програмного забезпечення тісно пов’язане з необхідністю забезпечення належного рівня безпеки, оскільки використання CI/CD пайплайнів значно прискорює процеси доставки змін і спрощує взаємодію між розробкою та експлуатацією, водночас створюючи нові потенційні вектори атак,

пов’язані з автоматизованим доступом до інфраструктури, облікових даних та конфіденційної інформації, при цьому сам пайплайн виступає центральною ланкою між репозиторієм вихідного коду та цільовим середовищем виконання, а у разі його компрометації зловмисник може отримати контроль над серверною інфраструктурою, змінити поведінку застосунку або здійснити витік даних, що формує необхідність впровадження комплексних механізмів захисту на всіх етапах автоматизації.

Безпека у цьому контексті охоплює не лише захист коду, але й контроль доступу до ресурсів, управління обліковими даними та мінімізацію ризиків, пов’язаних із використанням автоматизованих процесів, де кожна операція, що виконується пайплайном, потенційно має доступ до критичних компонентів системи, включаючи сервери, реєстри контейнерів та зовнішні сервіси, що потребує чіткого розмежування прав доступу та контролю за використанням конфіденційної інформації.

Одним із ключових аспектів забезпечення безпеки виступає управління секретами, до яких належать:

— паролі;
— токени доступу;
— ключі автентифікації;
— адреси серверів;
— інші конфіденційні дані.

Все те, що використовуються у процесі автоматизації, при цьому неправильне зберігання таких даних або їх включення безпосередньо у код може призвести до серйозних наслідків, включаючи несанкціонований доступ до інфраструктури, тому необхідно використовувати спеціалізовані механізми їх захищеного зберігання та передачі [32].

Принципи безпечного зберігання облікових даних (GitHub Secrets). У процесі побудови CI/CD пайплайнів виникає необхідність використання конфіденційних даних для виконання різних операцій, серед яких:

1. підключення до серверів;

2. автентифікація у реєстрах контейнерів;
3. взаємодія із зовнішніми сервісами.

При цьому зберігання таких даних у відкритому вигляді у репозиторії створює високий рівень ризику, особливо у випадках, коли репозиторій є публічним або має широкий доступ, що зумовлює використання спеціалізованих механізмів захисту, одним із яких є GitHub Secrets як інструмент для безпечного управління обліковими даними [15].

Секрети у цьому контексті представляють собою змінні середовища, значення яких зберігаються у зашифрованому вигляді та не відображаються у відкритому доступі, при цьому вони використовуються у пайплайнах для передачі конфіденційної інформації без ризику її витоку, що дозволяє інтегрувати безпечні механізми автентифікації у процес автоматизації без необхідності включення таких даних у код або конфігураційні файли.

Організація безпечного зберігання секретів базується на декількох фундаментальних принципах, серед яких відокремлення конфіденційних даних від коду дозволяє уникнути їх випадкового розповсюдження, оскільки всі секрети передаються у процес виконання пайплайнів динамічно і не зберігаються у репозиторії, що значно знижує ризик їх компрометації. Додатково застосовується шифрування, яке гарантує, що навіть у разі доступу до системи зберігання неможливо отримати значення секретів у відкритому вигляді, що забезпечує захист від несанкціонованого перегляду.

Контроль доступу до секретів реалізується через систему прав, де лише користувачі або процеси з відповідними дозволами можуть використовувати ці дані, що дозволяє обмежити їх використання лише необхідними сценаріями, одночасно застосовується принцип мінімальних привілеїв, згідно з яким кожен секрет має лише ті права, які необхідні для виконання конкретної задачі, що зменшує потенційний вплив у разі його компрометації. Регулярне оновлення облікових даних також виступає важливим елементом безпеки, оскільки періодична зміна секретів дозволяє знизити ризик їх використання сторонніми особами у разі витоку.

Інтеграція секретів у процеси автоматизації здійснюється через конфігураційні файли пайплайнів, зокрема у середовищах на базі GitHub Actions, де значення секретів підставляються у вигляді змінних середовища під час виконання сценаріїв, при цьому вони автоматично приховуються у логах виконання, що додатково запобігає їх розкриттю. Такий підхід забезпечує централізоване управління конфіденційними даними, спрощує інтеграцію з іншими компонентами інфраструктури та підвищує загальний рівень безпеки автоматизованих процесів (див. рис. 1.9) [15].

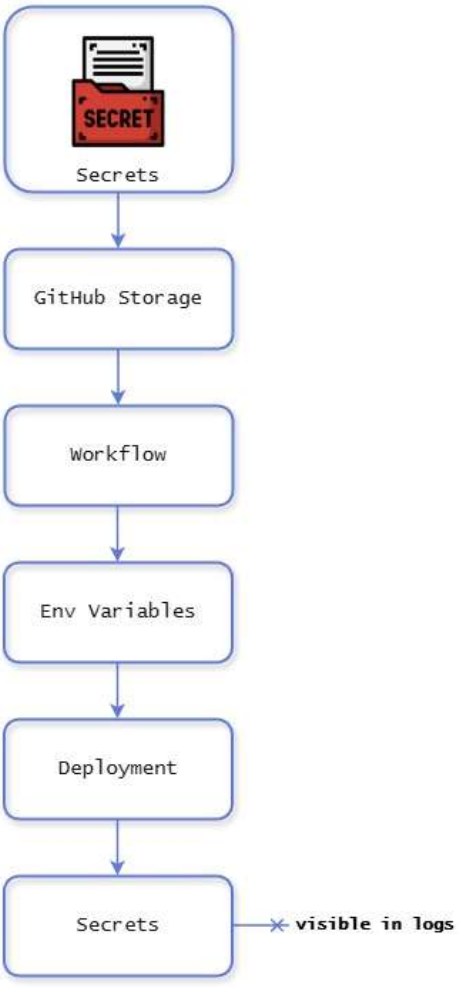


Рисунок 1.9 – Принципи роботи GitHub Secrets

У результаті впровадження практик безпечного управління секретами формується надійна основа для побудови захищених CI/CD процесів, де

конфіденційні дані використовуються контрольовано та без ризику витоку, що дозволяє забезпечити стабільність, безпеку та передбачуваність автоматизованого розгортання програмного забезпечення у сучасних інфраструктурних середовищах.

Обмеження доступу та захист цільової серверної інфраструктури. Окрім захисту секретів, важливим аспектом забезпечення безпеки автоматизованого розгортання є захист цільової серверної інфраструктури, оскільки саме на сервері відбувається виконання застосунку, зберігаються дані та забезпечується безперервність роботи сервісів, а отже будь-які помилки в налаштуванні доступу або відсутність належних механізмів контролю можуть призвести до несанкціонованого доступу, порушення роботи системи або компрометації всієї інфраструктури, при цьому процес розгортання, що виконується через автоматизовані пайплайни, має прямий доступ до серверного середовища, що підвищує вимоги до його захищеності та контролю.

Одним із базових підходів до обмеження доступу є використання автентифікації на основі ключів через SSH, що дозволяє відмовитися від передачі паролів та значно підвищує рівень безпеки за рахунок використання криптографічних алгоритмів, які ускладнюють підбір або перехоплення облікових даних, при цьому застосування ключів забезпечує більш контрольований доступ до серверів та дозволяє інтегрувати механізми автоматизації без збереження чутливої інформації у відкритому вигляді. Додатково практикується повне відключення автентифікації за паролем, що усуває один із найпоширеніших векторів атак і дозволяє залишити лише ключову автентифікацію як єдиний спосіб підключення.

Обмеження доступу також реалізується на мережевому рівні шляхом налаштування фільтрації трафіку, де доступ до серверів дозволяється лише з визначених IP-адрес або діапазонів, що значно зменшує поверхню атаки та унеможливлює підключення з невідомих джерел, при цьому використання міжмережевих екранів дозволяє централізовано контролювати всі вхідні та вихідні з’єднання. Додатковим рівнем захисту виступає створення окремих користувачів для процесів автоматизації, де кожен такий користувач має обмежені права доступу

та не може виконувати критичні операції поза межами визначених сценаріїв, що дозволяє знизити ризики у разі компрометації облікового запису.

Розмежування прав доступу базується на принципі мінімальних привілеїв, відповідно до якого кожному користувачу або процесу надається лише той рівень доступу, який необхідний для виконання конкретних задач, що обмежує можливість виконання небажаних або потенційно небезпечних дій, а також зменшує масштаб можливих наслідків у разі порушення безпеки. Важливим елементом є також контроль команд, які виконуються у процесі розгортання, оскільки автоматизовані пайплайни взаємодіють із сервером через віддалені з'єднання і виконують заздалегідь визначені сценарії, тому необхідно перевіряти їх зміст, уникати виконання неперевіреного коду та забезпечувати контроль джерел змін [16].

Окрему увагу необхідно приділяти захисту контейнеризованого середовища, зокрема при використанні Docker, де важливо обмежити доступ до демона керування контейнерами, оскільки він фактично надає повний контроль над системою, а також контролювати доступ до томів з даними, у яких можуть зберігатися бази даних, конфігурації або інші критичні ресурси, що потребують додаткового рівня захисту, включаючи налаштування прав доступу та ізоляцію середовищ виконання.

Не менш важливим компонентом забезпечення безпеки є моніторинг та аудит, які дозволяють [16]:

1. відслідковувати всі дії у системі;
2. аналізувати журнали доступу;
3. своєчасно виявляти підозрілу активність.

Все це дає змогу оперативно реагувати на потенційні загрози та запобігати їх розвитку, при цьому інтеграція таких механізмів із процесами автоматизації забезпечує постійний контроль за станом інфраструктури (див. рис. 1.10).

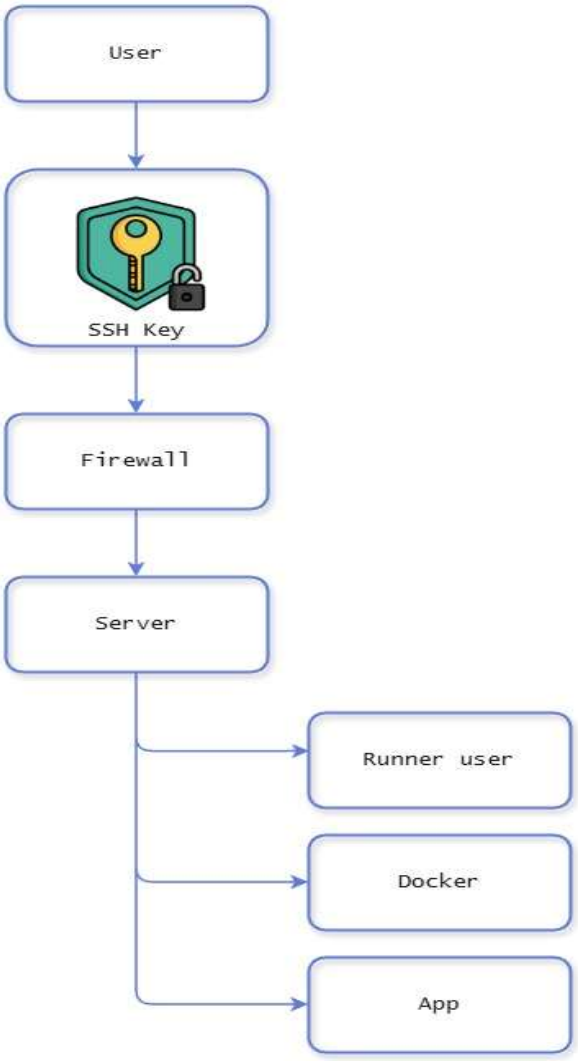


Рисунок 1.10 – Схема обмеження доступу до серверної інфраструктури

У контексті інтеграції з CI/CD пайплайнами доступ до серверів зазвичай реалізується через захищені з’єднання із використанням ключів, які зберігаються у безпечному середовищі, наприклад у GitHub Secrets, а виконання команд обмежується визначеним набором операцій, необхідних для розгортання застосунку, що дозволяє мінімізувати ризики та забезпечити контрольований процес автоматизації, при цьому поєднання всіх зазначених підходів формує надійну основу для побудови захищеного середовища розгортання, де обмеження доступу, контроль дій та захист інфраструктури працюють як єдина система, спрямована на забезпечення стабільності та безпеки сучасних програмних рішень.

2. ПРОЄКТУВАННЯ ТА РОЗРОБКА СИСТЕМИ АВТОМАТИЗОВАНОГО РОЗГОРТАННЯ НА БАЗІ GITHUB ACTIONS

2.1 Архітектура пайплайну CI/CD

У межах побудови системи автоматизованого розгортання контейнеризованого застосунку формується архітектура CI/CD-пайплайну, яка забезпечує безперервну інтеграцію змін та їх автоматизовану доставку до цільового середовища виконання, при цьому основна концепція базується на тому, що будь-яка зміна, внесена до репозиторію, ініціює процес розгортання без ручного втручання, що дозволяє значно підвищити швидкість оновлення застосунку та зменшити залежність від людського фактору.

Така архітектура формується з трьох основних компонентів, серед яких [17]:

1. віддалений репозиторій;
2. середовище виконання автоматизації у вигляді Self-hosted runner;
3. цільовий сервер, де безпосередньо виконується застосунок.

Причому у даній реалізації цільовим середовищем виступає локальна система на базі Windows із використанням підсистеми WSL2, що дозволяє створити ізольоване Linux-середовище для запуску контейнерів і виконання DevOps-процесів без необхідності використання зовнішніх серверів.

Взаємодія між компонентами реалізується за моделлю Pull, у якій сервер самостійно ініціює отримання завдань, а не очікує зовнішніх підключень, що суттєво підвищує рівень безпеки інфраструктури, оскільки відсутня необхідність відкриття вхідних портів, а всі з’єднання здійснюються через вихідні канали, що значно знижує ризики несанкціонованого доступу.

Загальна логіка функціонування виглядає таким чином, що після внесення змін у код застосунку та їх фіксації у репозиторії відбувається автоматичне спрацювання тригера пайплайну, після чого спеціалізований агент у вигляді Self-

hosted runner, розгорнутий безпосередньо на сервері, отримує сигнал про запуск процесу, виконує оновлення коду, збірку контейнерів та їх перезапуск, що забезпечує мінімальний час доставки змін та повну автоматизацію процесів [17].

Схема взаємодії: Репозиторій — Self-hosted runner — Цільовий *сервер.* Цільовим середовищем у побудові системи виступає локальний комп'ютер, на якому встановлено підсистему Windows Subsystem for Linux 2 з дистрибутивом Ubuntu, використання такого підходу дозволяє забезпечити повноцінне Linux-середовище всередині операційної системи Windows, що значно спрощує процеси розгортання та виконання контейнеризованих застосунків, а також дозволяє уникнути використання зовнішніх серверів або хмарних платформ, забезпечуючи повний контроль над інфраструктурою.

Віддалений репозиторій виступає центральним сховищем вихідного коду, у якому зберігаються всі компоненти проєкту, включаючи програмний код, Docker-конфігурації та сценарії автоматизації, причому кожна зміна, що фіксується у вигляді коміту, стає тригером для запуску процесу розгортання, що дозволяє організувати безперервний цикл оновлення застосунку.

Self-hosted runner виступає ключовим елементом системи, який забезпечує виконання процесів безпосередньо на сервері, на відміну від хмарних runner-ів він працює у локальному середовищі та має прямий доступ до файлової системи, Docker-демона та інших ресурсів, що дозволяє виконувати операції розгортання без додаткових мережевих взаємодій, при цьому runner функціонує як фоновий сервіс, що постійно підтримує з'єднання з репозиторієм та очікує на нові завдання [18].

Взаємодія між компонентами реалізується у вигляді послідовного процесу, де після внесення змін у код виконується "push" у гілку "main", репозиторій фіксує цю подію та ініціює запуск runner, який отримує завдання через захищене з'єднання, після чого виконує необхідні команди для оновлення та розгортання застосунку, а контейнери оновлюються і нова версія стає доступною для користувачів (див. табл. 2.1).

Таблиця 2.1 – Основні компоненти архітектури CI/CD пайплайну та їх функції

Компонент	Функція
Репозиторій	Зберігання коду, конфігурацій та Docker-файлів, тригер запуску пайплайну
Workflow	Опис логіки CI/CD процесу у вигляді YAML-конфігурації
Runner (Self-hosted)	Виконання задач пайплайну у локальному середовищі
Сервер (WSL2)	Цільове середовище розгортання застосунку
Docker	Контейнеризація застосунку та управління середовищем виконання
HTTPS-з’єднання	Безпечна передача завдань між репозиторієм і runner
Контейнери	Запуск і виконання застосунку після розгортання

Для представлення цього процесу доцільно використовувати блок-схему, що відображає послідовність дій, яка починається з push-запиту, переходить у тригер пайплайну, далі "runner" виконує процес розгортання, після чого формується результат, що може бути як успішним, так і з помилкою, причому у випадку помилки процес зупиняється на відповідному етапі, що дозволяє контролювати коректність виконання (див. рис. 2.1) [18].

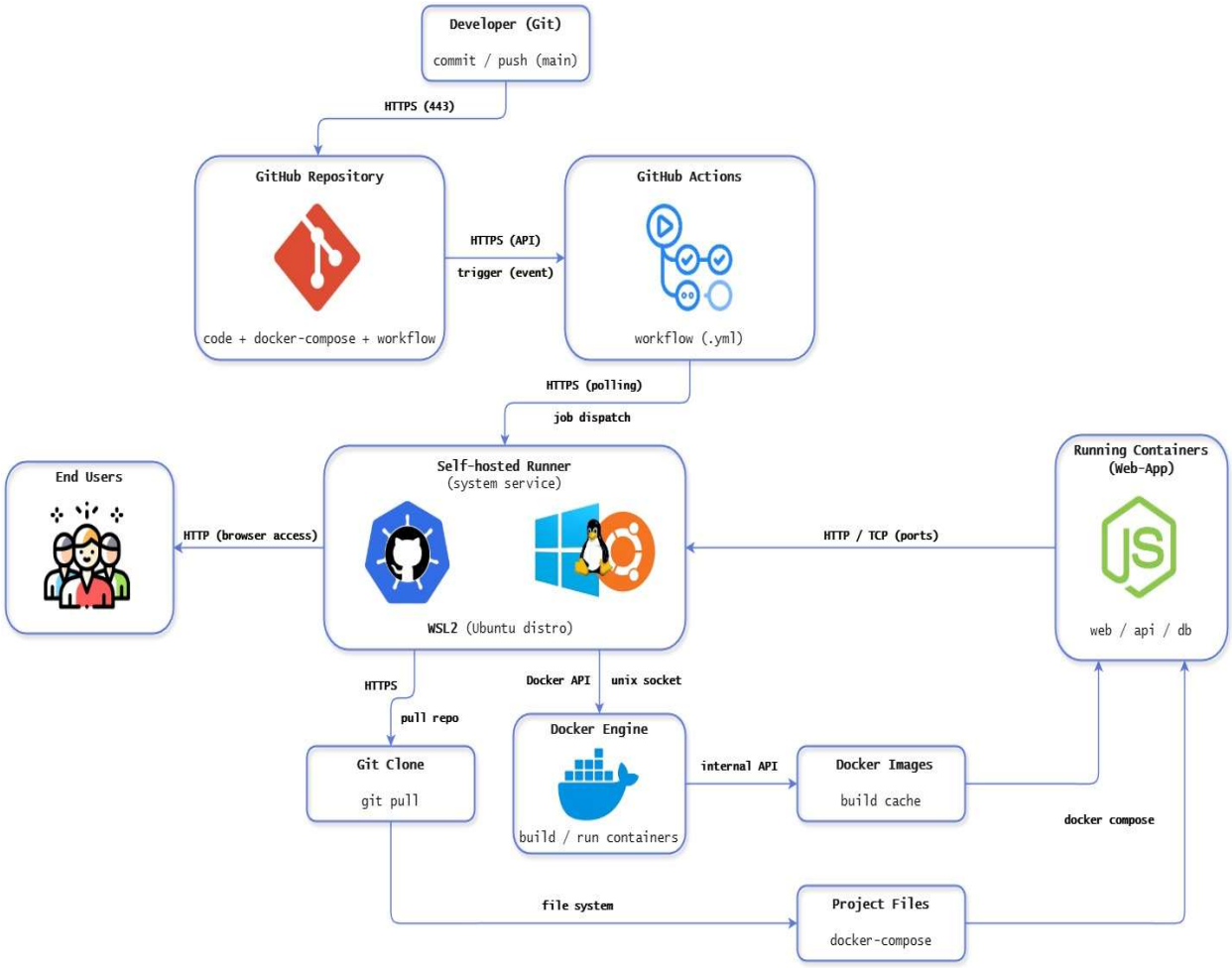


Рисунок 2.1 – Архітектура системи автоматизованого розгортання на базі GitHub Actions

Особливістю такої архітектури є використання моделі Pull, при якій сервер самостійно ініціює отримання завдань, що дозволяє уникнути відкриття додаткових мережевих портів, таких як SSH, і таким чином зменшує ризики несанкціонованого доступу, при цьому взаємодія здійснюється через захищені канали, що підвищує загальний рівень безпеки системи.

Структура проєкту та файлу конфігурації docker-compose.yml. Реалізована система базується на простому веб-застосунку, створеному за допомогою платформи Node.js, де основною метою вибору такої технології стало забезпечення наочної демонстрації принципів автоматизованого розгортання без ускладнення внутрішньої логіки самого застосунку, оскільки акцент зміщується саме на процес функціонування CI/CD-конвеєра та взаємодію його компонентів, а застосунок у

цьому випадку виступає індикатором змін, що дозволяє відслідковувати коректність виконання розгортання шляхом оновлення його версії у відповіді сервера. Структура проєкту включає набір ключових файлів, кожен із яких виконує визначену функцію у процесі побудови, контейнеризації та подальшого автоматизованого запуску системи, де центральним елементом виступає файл app.js, який реалізує базовий HTTP-сервер із використанням фреймворку Express та забезпечує обробку запитів на кореневому маршруті з поверненням HTML-відповіді, що містить інформацію про поточну версію застосунку та службові дані, і така реалізація дозволяє швидко перевіряти результат розгортання без використання складних інструментів моніторингу, оскільки будь-яка зміна у тексті відповіді одразу відображається у браузері після виконання пайплайну (див. рис. 2.2).



Рисунок 2.2 – Логічна структура проєкту

Файл "package.json" визначає метадані застосунку, включаючи його назву, версію та список залежностей, необхідних для коректної роботи, серед яких використовується бібліотека Express, що забезпечує швидке створення веб-сервера та обробку HTTP-запитів, а також дозволяє стандартизувати процес встановлення залежностей під час побудови Docker-образу, де виконання команди npm install відбувається автоматично у процесі збірки контейнера. Контейнеризація застосунку реалізується за допомогою Dockerfile, який описує покроковий процес створення образу, починаючи з використання базового образу " node:18-alpine ", що

обраний через малий розмір та оптимізовану продуктивність, після чого визначається робоча директорія всередині контейнера, копіюються файли залежностей та виконується їх встановлення, а вже потім додається решта проєкту, що дозволяє ефективно використовувати механізм кешування шарів Docker і значно пришвидшує повторні збірки у випадку, коли залежності не змінюються, далі відкривається порт 8080 та задається команда запуску застосунку, що забезпечує автоматичний старт сервера при запуску контейнера.

Управління контейнером та його життєвим циклом здійснюється за допомогою файлу "docker-compose.yml", який визначає конфігурацію сервісу та параметри його виконання, де у поточній реалізації використовується один сервіс із назвою "web", що будується на основі локального Dockerfile, контейнер отримує фіксоване ім’я для спрощення ідентифікації у внутрішній мережі Docker, а також виконується проброс портів із хост-системи на контейнер, де порт 80 відображається на порт 8080, що дозволяє отримувати доступ до застосунку через стандартний HTTP-інтерфейс без додаткової конфігурації з боку користувача, при цьому параметр "restart: always" забезпечує автоматичний перезапуск контейнера у випадку його аварійного завершення або перезавантаження системи, а механізм "healthcheck" дозволяє здійснювати періодичну перевірку доступності сервісу шляхом виконання HTTP-запиту, що формує додатковий рівень контролю за станом застосунку та дозволяє виявляти проблеми у його роботі (див. рис. 2.3) [19].

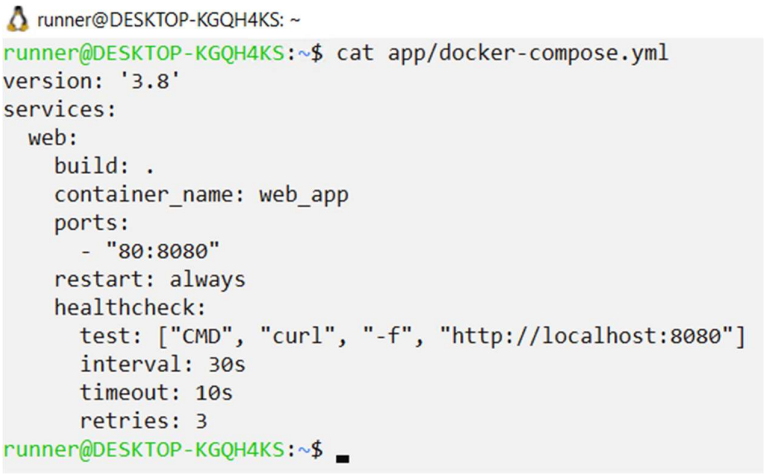


Рисунок 2.3 – Приклад конфігурації docker-compose.yml

Автоматизація процесу розгортання реалізується через файл "deploy.yml", розташований у директорії . "github/workflows", де описується сценарій виконання дій при кожному оновленні "гілки main", і після отримання тригера Self-hosted runner виконує послідовність команд, що починається з переходу до робочої директорії, далі відбувається синхронізація локального репозиторію з віддаленим за допомогою "git pull origin main", після чого запускається процес збірки нового Docker-образу через "docker compose build", що включає всі внесені зміни до коду застосунку, а завершальним етапом стає виконання команди "docker compose up -d", яка забезпечує запуск або оновлення контейнерів у фоновому режимі без зупинки сервісу, і після цього виконується очищення невикористаних образів за допомогою "docker image prune -f", що дозволяє оптимізувати використання дискового простору та уникнути накопичення застарілих версій (див. рис. 2.4).



Рисунок 2.4 – Приклад конфігурації deploy.yml

Логіку роботи такого пайплайну відображає послідовність виконання операцій, де початковим етапом є отримання події "push", після чого виконується "checkout" коду, далі відбувається "git pull" для синхронізації змін, наступним кроком є побудова Docker-образу, після чого запускається контейнер у фоновому режимі, і завершальним етапом виступає очищення системи від невикористаних ресурсів, при цьому у разі виникнення помилки на будь-якому етапі процес може бути перерваний, що формує розгалужену логіку виконання з можливістю обробки помилок та відображення їх у логах системи.

Сформована структура проєкту забезпечує повний цикл автоматизованого розгортання застосунку, починаючи від внесення змін у код і завершуючи їх миттєвим застосуванням у середовищі виконання, а поєднання контейнеризації, систем контролю версій та інструментів автоматизації формує цілісну інфраструктуру, яка відповідає сучасним підходам до розробки програмного забезпечення та дозволяє реалізувати концепцію інфраструктури як коду, що значно підвищує гнучкість, відтворюваність та надійність процесів розгортання.

2.2 Налаштування цільового середовища та конфігурація Actions

Практична реалізація системи автоматизованого розгортання передбачає не лише формування архітектурної моделі, а й детальну підготовку середовища виконання, у якому відбувається запуск сервісів, обробка змін коду та виконання всіх етапів CI/CD-конвеєра, при цьому особлива увага приділяється узгодженості компонентів інфраструктури, їх сумісності та безпеці взаємодії між ними, оскільки саме від коректності налаштування середовища залежить стабільність усієї системи автоматизації, у даному випадку розглядається підхід із використанням локальної інфраструктури на базі WSL2, інтеграції контейнерної платформи Docker та впровадження механізму автоматизації через Self-hosted runner, який забезпечує виконання процесів розгортання безпосередньо на сервері, а не у віддаленому

середовищі, що дозволяє досягти повного контролю над ресурсами, мережевими взаємодіями та політиками доступу, і така модель розгортання демонструє наближеність до реальних DevOps-практик, які застосовуються у production-середовищах, навіть при використанні локального комп'ютера як сервера.

Використання локального середовища замість віддалених серверів дозволяє моделювати повноцінну інфраструктуру розгортання, де всі компоненти — від системи контролю версій до середовища виконання контейнерів — знаходяться під безпосереднім контролем, що забезпечує гнучкість налаштувань і можливість детального аналізу процесів, пов'язаних із розгортанням, тестуванням та оновленням застосунків, а також дає змогу відпрацьовувати сценарії безпеки, управління доступом та оптимізації ресурсів без залежності від зовнішніх сервісів, у результаті чого формується повноцінна лабораторна модель автоматизованої інфраструктури, придатна для подальшого масштабування.

Підготовка сервера: WSL2, Docker Desktop та автентифікація. Цільовим середовищем для розгортання системи виступає локальний комп'ютер із встановленою підсистемою Windows Subsystem for Linux 2, що забезпечує повноцінне Linux-середовище всередині операційної системи Windows і дозволяє запускати сучасні інструменти розробки та контейнеризації без необхідності використання окремих віртуальних машин, при цьому вибір WSL2 обумовлений його високою продуктивністю, підтримкою ядра Linux та можливістю інтеграції з системними ресурсами хост-операційної системи, після інсталяції середовища розгортається дистрибутив Ubuntu, у якому виконується базове налаштування системи, що включає оновлення пакетів, встановлення необхідних утиліт та підготовку робочого середовища для подальшої роботи з контейнерами та системами автоматизації, і така початкова конфігурація формує стабільну основу для подальших етапів розгортання (див. рис. 2.5).

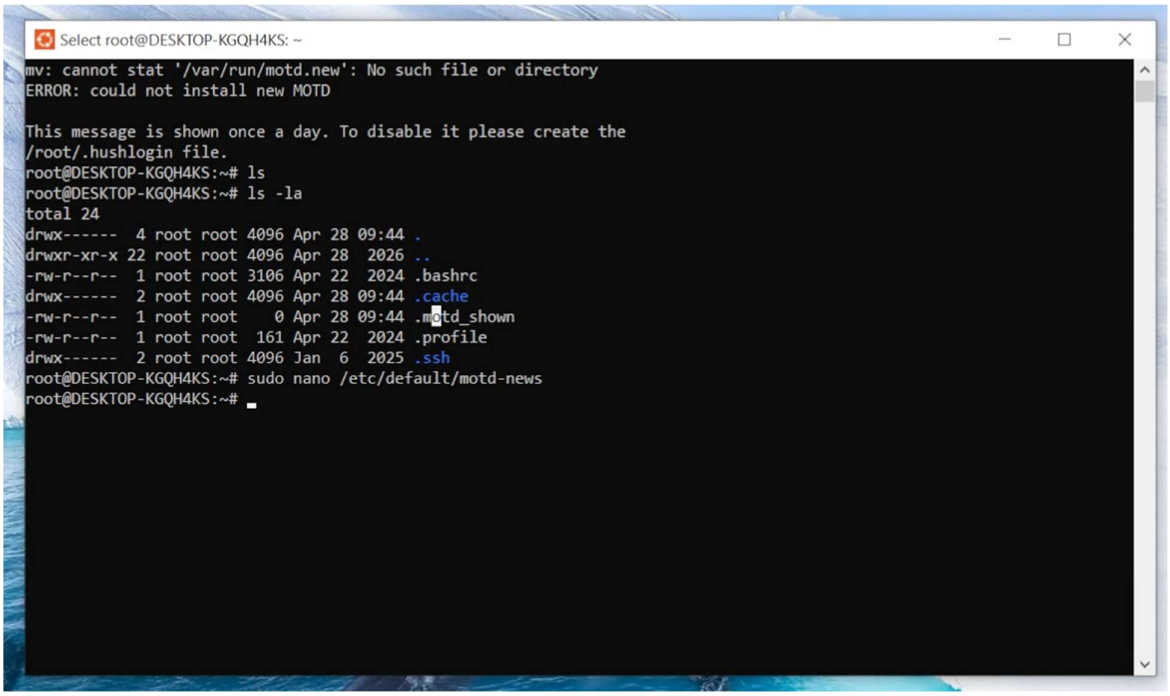


Рисунок 2.5 – Підготовлене середовище WSL2

Наступним етапом виступає інтеграція Docker у середовище WSL2, де Docker Desktop встановлюється у Windows, після чого активується режим інтеграції з обраним Linux-дистрибутивом, що дозволяє використовувати Docker Engine безпосередньо всередині WSL-середовища, забезпечуючи високу продуктивність і повну сумісність із Linux-інструментами, при цьому важливим аспектом є налаштування доступу до Docker daemon, де користувача додають до групи docker, що дозволяє виконувати команди керування контейнерами без використання прав адміністратора, і така конфігурація відповідає принципу мінімальних привілеїв, зменшуючи ризики, пов’язані з виконанням команд від імені суперкористувача, а також підвищує загальний рівень безпеки системи (див. рис. 2.6).

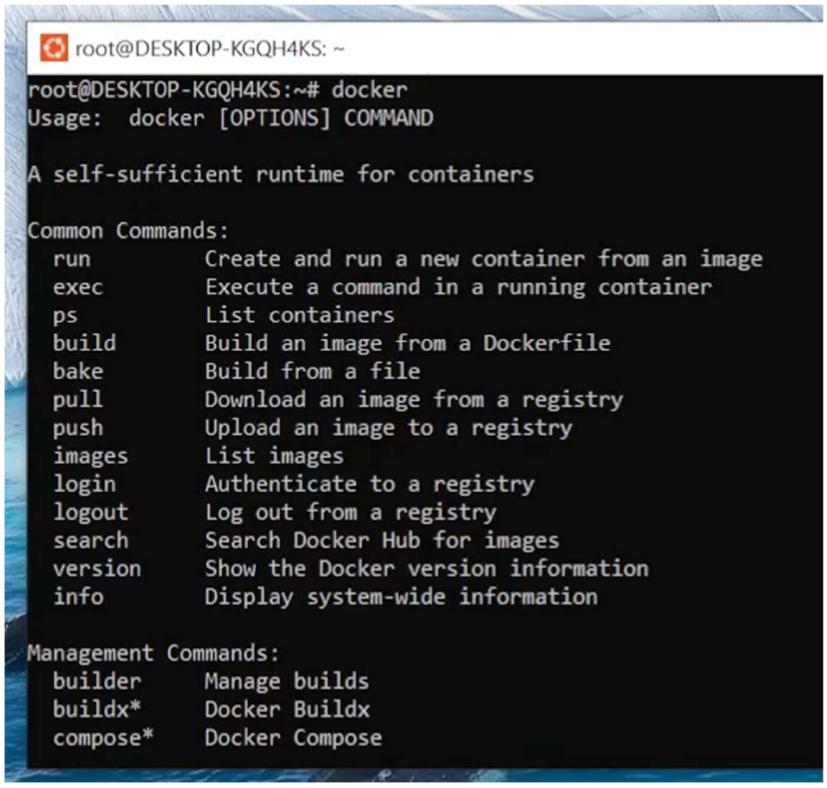


Рисунок 2.6 – CLI інтерфейс Docker у терміналі WSL2

Паралельно з налаштуванням контейнерного середовища виконується конфігурація взаємодії з віддаленим репозиторієм, де традиційний підхід із використанням паролів через HTTPS замінюється на більш безпечні механізми автентифікації, зокрема використовується "Personal Access Token", який дозволяє виконувати операції з репозиторієм із обмеженим набором прав і забезпечує контроль доступу до ресурсів, а додатково впроваджується автентифікація за допомогою SSH-ключів, згенерованих за алгоритмом Ed25519, що забезпечує високий рівень криптографічного захисту при компактному розмірі ключа та швидкому виконанні операцій шифрування, при цьому публічний ключ додається до облікового запису у системі контролю версій, а приватний зберігається на сервері та використовується для встановлення захищеного з’єднання, що дозволяє уникнути передачі конфіденційних даних у відкритому вигляді (див. рис. 2.7).

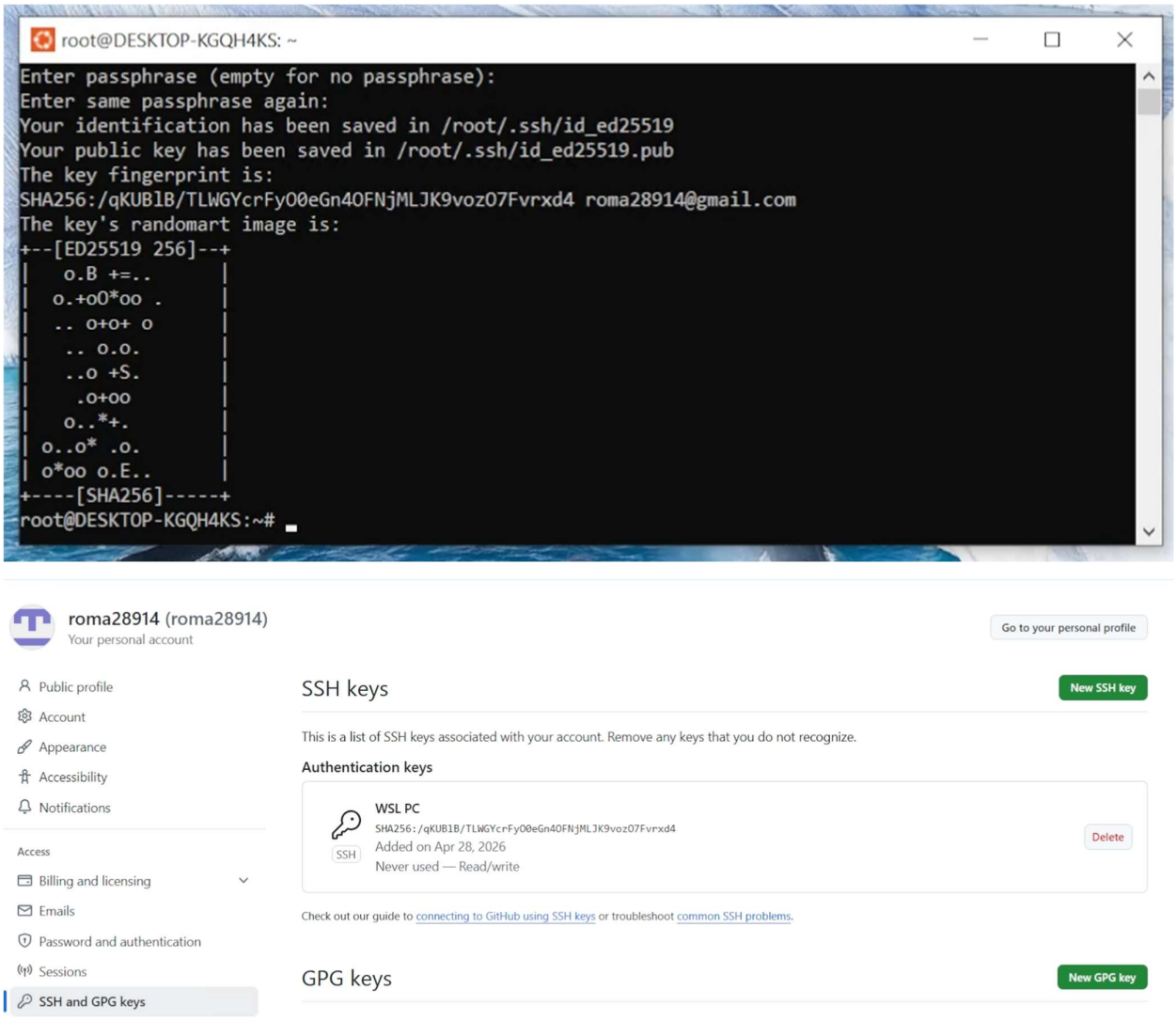


Рисунок 2.7 – Процес автентифікації у системі

У процесі підготовки середовища також виконується стандартизація структури репозиторію, де основна гілка визначається як main, що відповідає сучасним практикам розробки та забезпечує сумісність із інструментами автоматизації, які за замовчуванням орієнтовані саме на цю гілку, і така уніфікація спрощує налаштування пайплайнів та підвищує їх передбачуваність, а логіку підготовки середовища доцільно представити у вигляді послідовної схеми, де початковим етапом є встановлення WSL2, далі виконується розгортання дистрибутиву Ubuntu, після чого налаштовується Docker із підтримкою інтеграції, наступним кроком виступає конфігурація користувача та прав доступу, а завершальним етапом є налаштування механізмів автентифікації через SSH та

токени, що формує завершене середовище, готове до інтеграції з системою автоматизації.

Розгортання Self-hosted runner як системної служби. Після завершення підготовки серверного середовища наступним етапом стає розгортання механізму виконання CI/CD-процесів, де центральним елементом виступає Self-hosted runner, використання якого дозволяє виконувати автоматизовані сценарії безпосередньо у середовищі сервера та забезпечує повний контроль над ресурсами, файловою системою і контейнерною інфраструктурою, а сам процес розгортання починається із завантаження інсталяційного пакета runner із відповідного репозиторію, після чого виконується його розпакування у виділену директорію та запуск конфігураційного скрипта, який здійснює прив’язку до конкретного репозиторію, на цьому етапі задаються параметри підключення, включаючи URL репозиторію та реєстраційний токен, що генерується автоматично під час створення "runner" у системі керування, у результаті чого "runner" реєструється як доступний виконавець завдань і стає видимим у загальній інфраструктурі автоматизації (див. рис. 2.8).

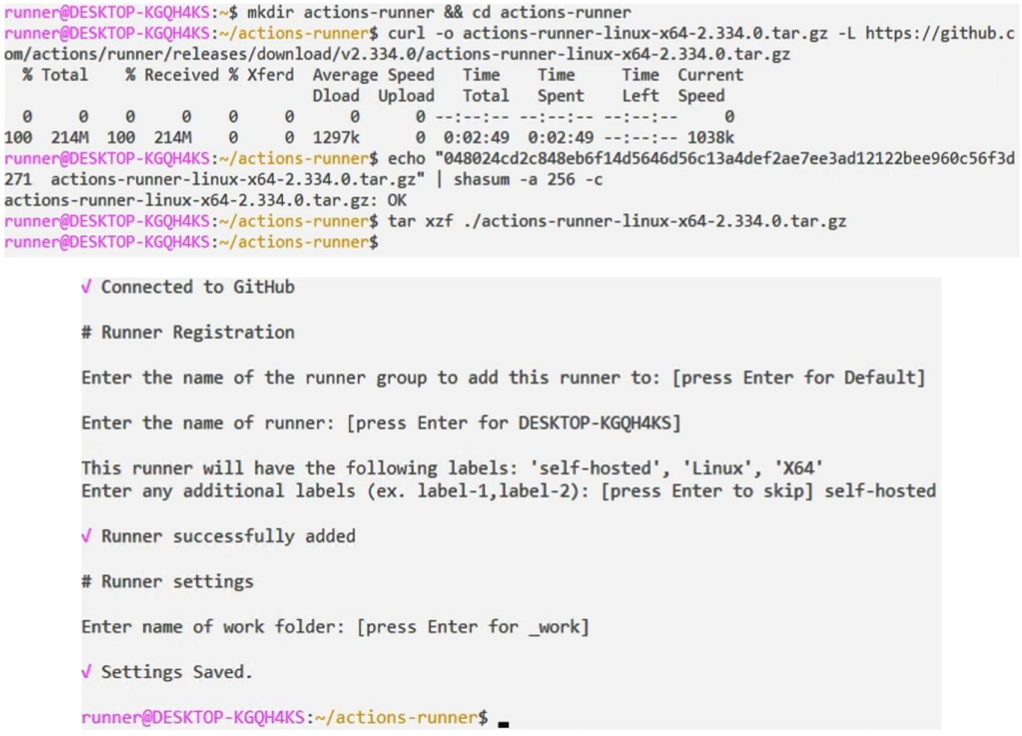


Рисунок 2.8 – Встановлення Self-Hosted Runner

Особлива увага приділяється способу запуску "runner", оскільки замість ручного виконання через shell-скрипти застосовується підхід із використанням системної служби, що забезпечує стабільність і безперервність роботи, де за допомогою скрипта "svc.sh" виконується інсталяція служби в систему ініціалізації, а подальший запуск активує її роботу у фоновому режимі, і така конфігурація дозволяє автоматично запускати runner після перезавантаження системи, забезпечує постійне функціонування без необхідності ручного втручання та гарантує виконання CI/CD-завдань у будь-який момент часу, що є критично важливим для підтримки безперервності процесів розгортання (див. рис. 2.9) [22].

```
drwxrwxr-x  7 runner runner      4096 Apr 28 12:26 _work/
-rw-rw-r--  1 runner runner 225016209 Apr 28 12:06 actions-runner-linux-x64-2.334.0.tar.gz
drwxr-xr-x 17 runner runner     20480 Apr 21 11:50 bin/
-rwxr-xr-x  1 runner runner      2458 Apr 21 11:49 config.sh*
-rwxr-xr-x  1 runner runner       646 Apr 21 11:49 env.sh*
drwxr-xr-x  6 runner runner      4096 Apr 21 11:50 externals/
-rw-r--r--  1 runner runner      1824 Apr 21 11:49 run-helper.cmd.template
-rwxr-xr-x  1 runner runner      2866 Apr 28 12:10 run-helper.sh*
-rwxr-xr-x  1 runner runner      2866 Apr 21 11:49 run-helper.sh.template*
-rwxr-xr-x  1 runner runner      2885 Apr 21 11:49 run.sh*
-rwxr-xr-x  1 runner runner       751 Apr 21 11:49 safe_sleep.sh*
-rwxr-xr-x  1 runner runner      5238 Apr 28 12:09 svc.sh*
runner@DESKTOP-KGQH4KS:~/actions-runner$ sudo ./svc.sh install
Creating launch runner in /etc/systemd/system/actions.runner.roma28914-my-web-app.DESKTOP-KGQH4KS.service
Run as user: runner
Run as uid: 1000
gid: 1000
Created symlink /etc/systemd/system/multi-user.target.wants/actions.runner.roma28914-my-web-app.DESKTOP-KGQH
4KS.service → /etc/systemd/system/actions.runner.roma28914-my-web-app.DESKTOP-KGQH4KS.service.
runner@DESKTOP-KGQH4KS:~/actions-runner$
```

Рисунок 2.9 – Запуск runner як системної служби

У процесі реалізації враховуються аспекти безпеки, зокрема заборона виконання "runner" від імені користувача "root", що відповідає сучасним вимогам захисту систем, замість чого створюється окремий технічний користувач із обмеженими правами доступу, якому надаються необхідні дозволи для взаємодії з Docker та виконання команд автоматизації, і такий підхід дозволяє мінімізувати ризики компрометації системи, оскільки навіть у випадку потенційної вразливості доступ до критичних ресурсів залишається обмеженим.

Взаємодію Self-hosted runner із системою автоматизації доцільно розглядати у вигляді логічної схеми, де початковою точкою виступає віддалений репозиторій,

який через захищене HTTPS-з’єднання по порту 443 передає сигнал про наявність нового завдання, після чого runner, що працює у фоновому режимі як "daemon", отримує це завдання та ініціює виконання визначених команд, взаємодіючи з Docker-середовищем для побудови образів та запуску контейнерів, і вся ця логіка реалізується у вигляді сценаріїв, що виконуються послідовно у межах workflow-файлу.

Алгоритм роботи "runner" може бути представлений як циклічний процес, у якому початковим станом виступає режим очікування, після чого відбувається отримання завдання, далі виконується послідовність команд, визначених у конфігурації пайплайну, після завершення яких формується результат виконання, що передається назад у систему керування, і завершальним етапом стає повернення до режиму очікування наступного завдання, при цьому така модель забезпечує безперервність роботи та швидке реагування на зміни у репозиторії.

Важливим аспектом функціонування виступає мережева модель взаємодії, яка базується на використанні вихідного HTTPS-з’єднання, що дозволяє уникнути відкриття вхідних портів на сервері, у результаті чого значно зменшуються ризики несанкціонованого доступу та мережевих атак, а також усуваються проблеми, пов’язані з конфігурацією NAT або firewall, і такий підхід забезпечує високий рівень безпеки без необхідності додаткових налаштувань мережевої інфраструктури (див. рис. 2.10).



Рисунок 2.10 – Робота runner у фоновому режимі

У результаті виконання всіх етапів формується повноцінна система автоматизації, у якій Self-hosted runner виступає центральним виконавчим компонентом, що забезпечує безперервне виконання CI/CD-процесів, інтеграцію з Docker-середовищем та автоматичне розгортання контейнеризованих застосунків після кожної зміни у репозиторії, при цьому оновлення відбувається на рівні контейнерів, де створюються нові образи, старі версії видаляються, а сервіси перезапускаються без втручання ззовні, і така організація процесу дозволяє реалізувати замкнений цикл автоматизованого розгортання, який функціонує безперервно доти, доки підтримується активність репозиторію та доступність серверного середовища, формуючи надійну основу для стабільної, безпечної та ефективної роботи всієї інфраструктури.

2.3 Алгоритми функціонування конвеєра розгортання

У межах побудованої системи автоматизованого розгортання особлива увага приділяється формалізації алгоритмів, які забезпечують повністю автоматичне оновлення застосунку після внесення змін до репозиторію, де основою функціонування виступає конфігураційний файл "workflow", розташований у директорії ".github/workflows", що визначає послідовність дій для виконання процесів безперервної інтеграції та доставки змін, завдяки чому усувається потреба у ручному втручанні та забезпечується висока швидкість реакції системи на будь-які оновлення коду, при цьому конвеєр працює у режимі постійного очікування подій від віддаленого репозиторію, а в момент фіксації змін у вигляді push-запиту автоматично ініціюється сценарій розгортання, який складається з чітко визначених етапів, серед яких отримання актуального стану коду, побудова нового Docker-образу, перезапуск контейнерів із новою версією застосунку та очищення невикористаних ресурсів, причому кожен із зазначених етапів має власну внутрішню логіку виконання та взаємодіє з іншими компонентами системи,

формуючи єдиний узгоджений механізм функціонування конвеєра, що забезпечує стабільність, відтворюваність і передбачуваність процесів розгортання (див. табл. 2.2) [31].

Таблиця 2.2 – Послідовність виконання етапів CI/CD пайплайну

Крок	Дія	Результат
1	Push у репозиторій	Ініціація тригера
2	Запуск workflow	Активується пайплайн
3	Підключення runner	Отримання задачі
4	git pull	Оновлення коду
5	Перехід у директорію	Підготовка середовища
6	docker compose build	Створення нового образу
7	docker compose up -d	Перезапуск контейнерів
8	Перевірка виконання	Логи та статус
9	Завершення	Оновлений застосунок

Алгоритм ініціації збірки Docker-образів при оновленні коду. Перший ключовий алгоритм у структурі конвеєра розгортання пов’язаний з ініціацією процесу збірки Docker-образів після оновлення вихідного коду, де початковою точкою виступає внесення змін до репозиторію та їх подальше відправлення у гілку main, що автоматично активує тригер у системі GitHub Actions, який у свою чергу запускає Self-hosted runner, розгорнутий у локальному середовищі, після чого відбувається передача сигналу через захищене HTTPS-з’єднання, і "Runner" переходить у стан виконання задачі, починаючи з оновлення локальної копії проєкту шляхом виконання команди "git pull origin main", що гарантує синхронізацію робочої директорії із віддаленим репозиторієм та усуває ризик використання застарілої версії коду, далі виконується перехід до відповідної директорії, у якій розташований проєкт, оскільки правильне визначення контексту виконання команд є критично важливим для коректної роботи Docker Compose, який орієнтується на поточне розташування файлу конфігурації.

Після завершення етапу синхронізації розпочинається процес побудови Docker-образу за допомогою команди "docker compose build", де відбувається аналіз "Dockerfile", визначення базового образу, у даному випадку "Node.js" версії 18 на

базі Alpine Linux, створення робочої директорії всередині контейнера, копіювання файлів із залежностями, встановлення необхідних бібліотек через пакетний менеджер npm, копіювання основного коду застосунку та визначення параметрів запуску, включаючи порт і команду старту, при цьому важливою особливістю є використання механізму кешування шарів Docker, що дозволяє суттєво скоротити час повторної збірки, оскільки при внесенні змін лише у прикладний код перевиконуються лише фінальні шари, тоді як попередні залишаються незмінними, що значно оптимізує продуктивність пайплайну та зменшує витрати ресурсів (див. рис. 2.11) [23].

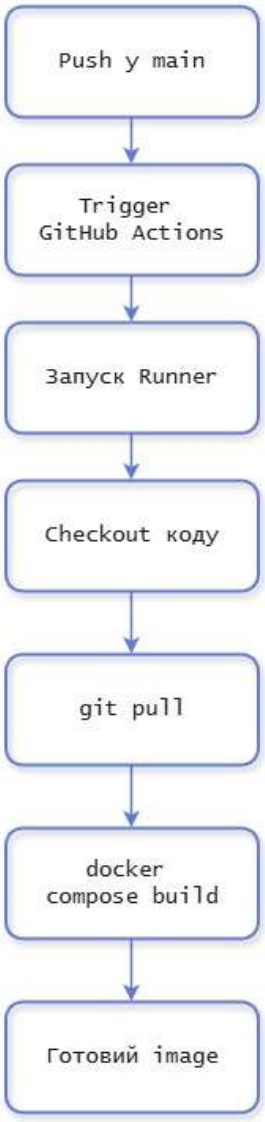


Рисунок 2.11 – Алгоритм ініціації збірки Docker-образу

Завдяки такій організації процесу забезпечується повна відтворюваність збірки, оскільки кожна нова версія застосунку формується як окремий Docker-образ із чітко визначеним середовищем виконання, що виключає залежність від зовнішніх факторів і гарантує однакову поведінку системи незалежно від умов розгортання.

Алгоритм безшовного перезапуску контейнерів без простою. Після успішного завершення етапу збірки система переходить до наступного критично важливого процесу, пов’язаного з оновленням уже запущеного застосунку, де основною вимогою виступає мінімізація простою сервісу або його повна відсутність, що досягається завдяки використанню механізмів оркестрації контейнерів через Docker Compose, який аналізує зміни у конфігурації або у щойно зібраному образі та приймає рішення щодо необхідності перезапуску відповідних сервісів, у межах цього процесу виконується перевірка актуального стану контейнерів, визначення наявності нового образу, після чого у разі необхідності відбувається зупинка застарілих контейнерів, створення нових екземплярів на основі оновленого образу, запуск контейнерів у фоновому режимі із використанням "detached mode" та повторне підключення до мережевої інфраструктури із збереженням налаштованих портів і залежностей між сервісами, причому особливістю даного підходу виступає те, що Docker Compose виконує оновлення лише для тих сервісів, у яких дійсно відбулися зміни, завдяки чому уникається повний перезапуск усієї системи та суттєво зменшується вплив на доступність застосунку [24].

Додатковий контроль працездатності забезпечується через механізм healthcheck, який виконує періодичні перевірки доступності сервісу шляхом надсилання HTTP-запитів до контейнера, і у випадку виявлення помилок або недоступності застосунку система може автоматично ініціювати повторний запуск контейнера, що підвищує загальну надійність інфраструктури та дозволяє підтримувати стабільний стан сервісу навіть за умов часткових збоїв, при цьому логіка виконання алгоритму доцільно описується через розгалужену блок-схему, де початковим етапом виступає отримання нового Docker-образу після завершення

попереднього алгоритму, далі виконується перевірка його наявності, після чого у разі існування активних контейнерів відбувається їх коректна зупинка та створення нових екземплярів, а у випадку відсутності контейнерів одразу ініціюється створення нових, наступним кроком виконується запуск через команду "docker compose up -d", після чого активується механізм перевірки стану через "healthcheck", де у разі успішного проходження перевірки процес завершується, а у випадку помилки відбувається повторна спроба запуску або відновлення працездатності сервісу.

Після запуску оновлених контейнерів виконується завершальний етап оптимізації, пов’язаний із очищенням невикористаних Docker-образів за допомогою команди "docker image prune -f", що дозволяє уникнути накопичення застарілих версій образів та є особливо важливим в умовах обмежених ресурсів локального середовища, зокрема при використанні WSL2 або інших серверних конфігурацій, де контроль використання дискового простору має критичне значення для стабільної роботи системи, таким чином підтримується ефективне управління ресурсами та запобігається деградація продуктивності внаслідок перевантаження файлової системи (див. рис. 2.12).

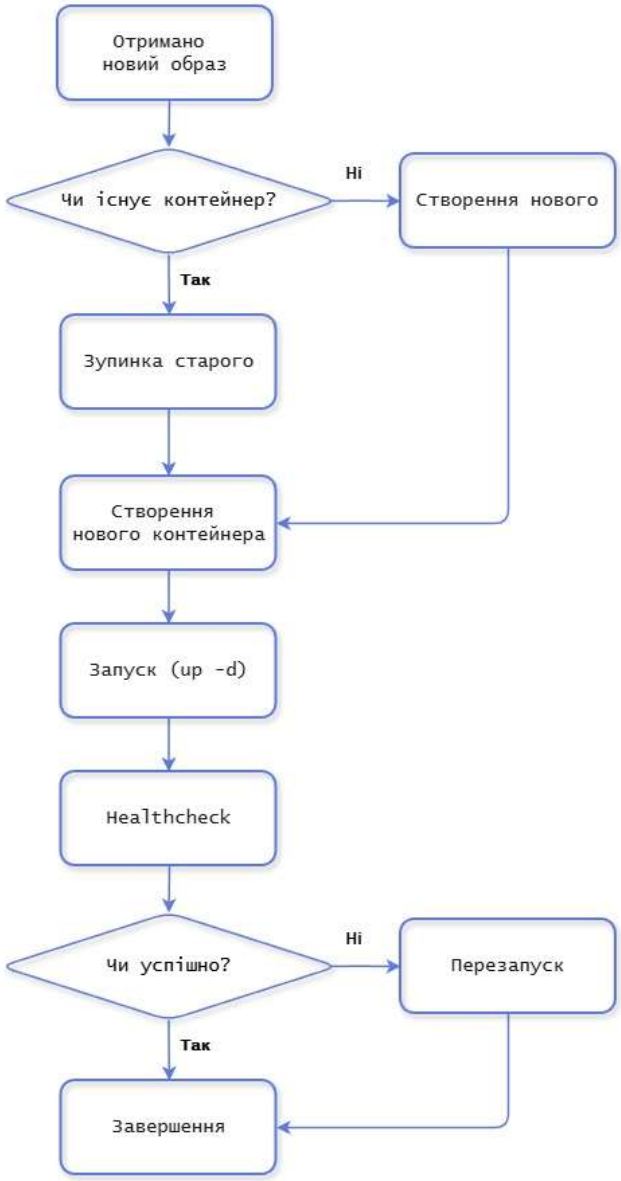


Рисунок 2.12 – Алгоритм оновлення контейнерів без простою

У підсумку реалізація алгоритму безшовного перезапуску контейнерів дозволяє забезпечити відсутність необхідності ручного втручання у процес розгортання, мінімізувати час недоступності застосунку, автоматизувати відновлення після збоїв та забезпечити постійний контроль стану контейнерів через механізм "healthcheck", а поєднання цього алгоритму з попереднім етапом збірки формує завершений цикл безперервної доставки, у якому кожна зміна коду автоматично проходить шлях від репозиторію до робочого середовища без участі адміністратора, що відповідає сучасним підходам DevOps та забезпечує високу

відтворюваність, стабільність і швидкість виконання процесів розгортання контейнеризованих застосунків.

2.4 Тестування процесу автоматичного розгортання

Після завершення етапів проєктування архітектури системи та формалізації алгоритмів функціонування конвеєра розгортання виникає необхідність практичної перевірки працездатності, стабільності та відповідності системи поставленим вимогам, при цьому тестування спрямовується не лише на підтвердження коректності виконання окремих операцій, а й на оцінку цілісності всього циклу автоматизованого розгортання від моменту внесення змін у вихідний код до їх фактичного відображення у працюючому застосунку, у межах якого використовується підхід імітації реального сценарію експлуатації, коли розробник вносить зміни до проєкту, після чого система без будь-якого ручного втручання виконує всі необхідні етапи, включаючи отримання оновленого коду, побудову нового Docker-образу, перезапуск контейнерів та публікацію оновленої версії сервісу, такий підхід дозволяє оцінити ефективність функціонування конвеєра у максимально наближених до реальних умовах, водночас особлива увага приділяється двом основним аспектам, що характеризують якість реалізації системи, а саме перевірці факту оновлення застосунку після внесення змін та аналізу внутрішніх процесів виконання пайплайну через журнали, що дає змогу не лише підтвердити коректність роботи, але й виявити потенційні помилки конфігурації або вузькі місця в алгоритмах [25].

Валідація успішного оновлення застосунку після push-запиту. У межах даного етапу тестування основною метою виступає перевірка того, що будь-яка зміна у вихідному коді автоматично призводить до оновлення застосунку без необхідності ручного втручання, для цього формується тестовий сценарій, максимально наближений до реального процесу розробки, де у вихідному коді

застосунку, зокрема у файлі, що відповідає за обробку HTTP-запитів, змінюється відображуваний текст, наприклад версія системи змінюється з "0.1.0" на "0.1.1". Така зміна має мінімальний вплив на логіку застосунку, однак є достатньою для перевірки повного циклу автоматичного розгортання, після внесення змін виконується стандартна послідовність команд системи контролю версій, де спочатку всі модифікації додаються до індексу, далі створюється коміт із відповідним повідомленням, після чого зміни відправляються до віддаленого репозиторію у гілку "main", саме операція "push" виступає тригером для запуску CI/CD пайплайну, у цей момент відбувається передача змін до системи керування репозиторієм, яка автоматично ініціює виконання "workflow", визначеного у GitHub Actions, далі Self-hosted runner, що функціонує у фоновому режимі, отримує сигнал про нову подію через захищене з'єднання та розпочинає виконання послідовності команд, визначених у сценарії розгортання, включаючи оновлення коду, побудову нового образу та перезапуск контейнерів, при цьому характерною особливістю є майже миттєва реакція системи, що підтверджує ефективність використання pull-моделі взаємодії, де сервер самостійно відстежує зміни та ініціює їх обробку (див. рис. 2.13).

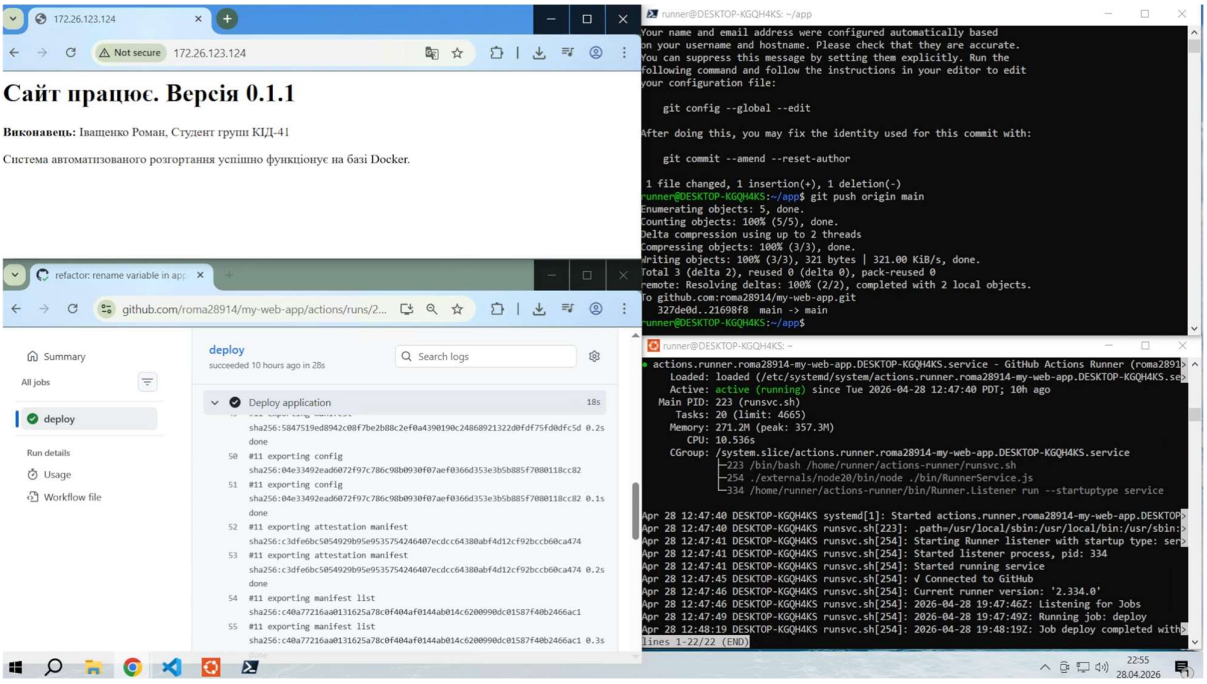


Рисунок 2.13 – Відображення оновленого застосунку у браузері (повний CI/CD)

Логіка виконання даного процесу може бути представлена у вигляді лінійної послідовності, у якій початковим етапом виступає зміна коду, далі виконується push до гілки main, після чого спрацьовує тригер GitHub Actions, запускається Self-hosted runner, виконується пайплайн розгортання, оновлюються контейнери та відбувається відображення нової версії застосунку у браузері, завершальним етапом тестування виступає перевірка результату, що здійснюється шляхом відкриття веб-сторінки сервісу, де відображається оновлений текст із новою версією, що підтверджує успішність виконання всіх етапів конвеєра (див. рис. 2.14).



Рисунок 2.14 – Сценарій автоматичного оновлення після push

Додатково аналізується час виконання процесу, який у середньому становить від кількох десятків секунд до однієї хвилини залежно від продуктивності системи та тривалості побудови Docker-образу, отримані результати демонструють суттєве скорочення часу розгортання у порівнянні з ручними підходами, де аналогічні

операції можуть займати значно більше часу та вимагати безпосередньої участі адміністратора, що дозволяє зробити висновок про ефективність реалізованої системи та її відповідність сучасним підходам автоматизації процесів розробки і доставки програмного забезпечення.

Перевірка відпрацювання пайплайну та управління ресурсами. Другим важливим етапом тестування виступає аналіз внутрішніх процесів виконання пайплайну, який дозволяє оцінити не лише кінцевий результат у вигляді оновленого застосунку, а й коректність виконання кожного окремого етапу розгортання, що формує цілісне уявлення про стабільність та передбачуваність роботи всієї системи автоматизації, при цьому використовується механізм журналювання, що надається платформою GitHub Actions, де після запуску workflow відкривається інтерфейс виконання задачі з відображенням усіх етапів у вигляді послідовних кроків, кожен з яких супроводжується детальними логами, що містять інформацію про виконані команди, їх результати та можливі помилки (див. рис. 2.15).

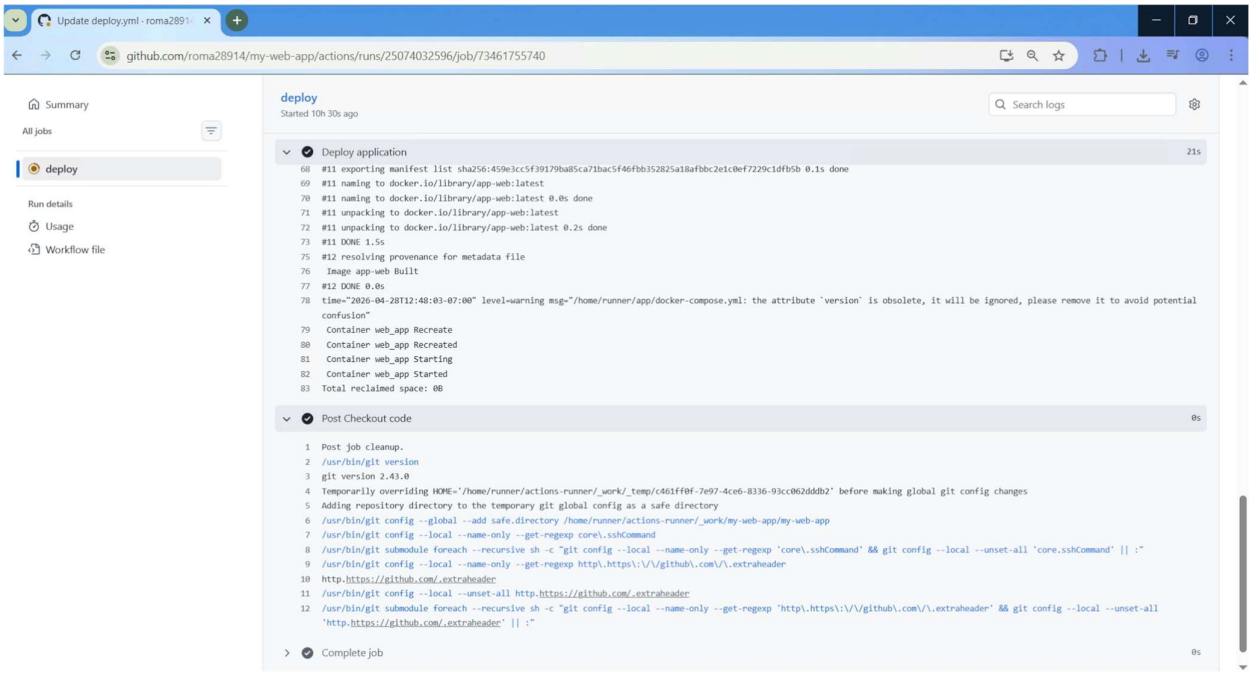


Рисунок 2.15 – Інтерфейс виконання workflow (GitHub Actions)

Особлива увага приділяється таким ключовим етапам, як успішність отримання вихідного коду та виконання команди "git pull", коректність збірки Docker-образу, створення і оновлення контейнерів, а також виконання фінальної команди очищення ресурсів, при цьому у випадку успішного виконання всі етапи позначаються як завершені без помилок, що візуально відображається у вигляді відповідних статусів, тоді як наявність помилок супроводжується сигналізацією про збій, що дозволяє оперативно локалізувати проблемний етап, а самі логи містять повідомлення про створення нового образу або запуск контейнера, що виступає прямим підтвердженням виконання алгоритмів, описаних у попередніх підрозділах (див. рис. 2.16) [26].



Рисунок 2.16 – Кроки виконання пайплайну

Окремий акцент робиться на управлінні ресурсами, оскільки контейнеризоване середовище, особливо в умовах локальної інфраструктури на базі WSL2, має обмежені обчислювальні та дискові ресурси, що з часом можуть вичерпуватися внаслідок накопичення застарілих Docker-образів, у зв’язку з чим у пайплайні передбачено використання команди "docker image prune -f", яка автоматично видаляє невикористані образи після кожного розгортання, забезпечуючи стабільність роботи системи та запобігаючи деградації продуктивності через нестачу дискового простору.

У процесі тестування також аналізуються потенційні збої, що можуть виникати під час виконання пайплайну, серед яких поширеними є помилки

синтаксису у workflow-файлі, некоректно задані шляхи до робочих директорій, відсутність необхідних прав доступу у користувача, від імені якого працює Self-hosted runner, а також зміни у конфігураційних файлах, що можуть порушити логіку роботи Docker Compose, при цьому система журналювання надає детальну інформацію щодо кожної помилки, що суттєво спрощує процес діагностики та прискорює її усунення (див. рис. 2.17) [26].

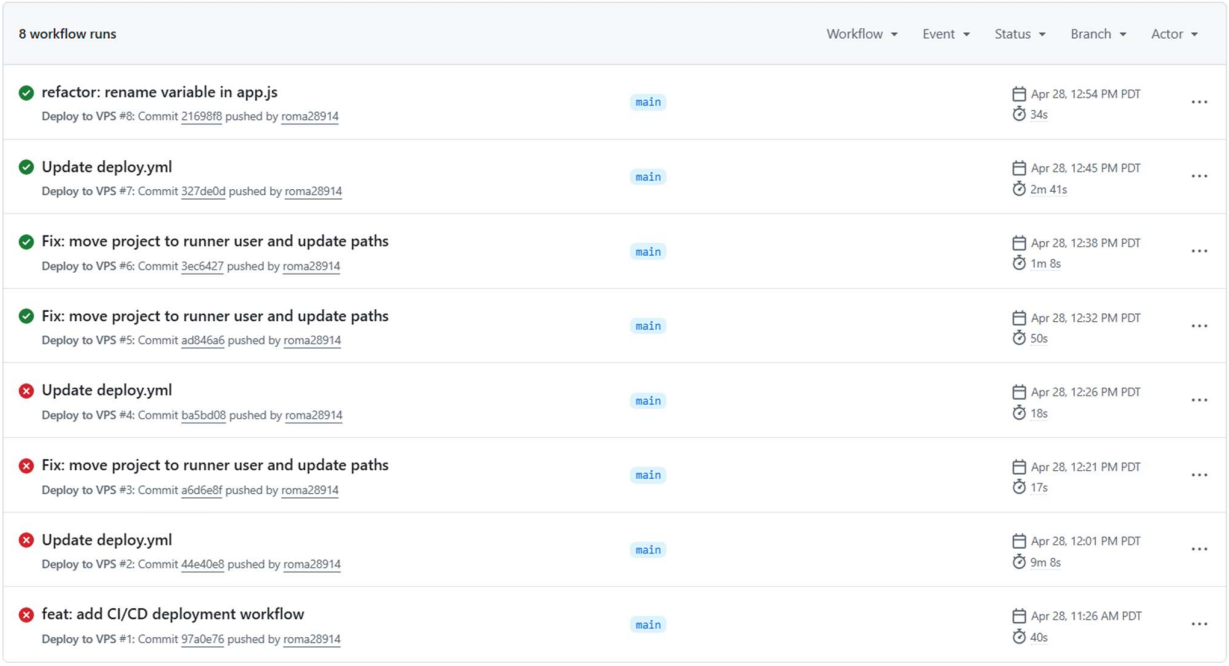


Рисунок 2.17 – Логування всіх комітів під час пайплайну

У підсумку проведене тестування демонструє, що система забезпечує стабільне, прозоре та контрольоване виконання всіх етапів автоматизованого розгортання, а поєднання механізмів журналювання, автоматичного очищення ресурсів та обробки помилок формує надійну інфраструктуру, здатну до ефективної роботи у реальних умовах експлуатації без необхідності постійного втручання адміністратора, що підтверджує відповідність реалізованого підходу сучасним практикам побудови CI/CD систем.

3. ОПТИМІЗАЦІЯ КОНВЕЄРА РОЗГОРТАННЯ ТА ПЕРСПЕКТИВИ РОЗВИТКУ

3.1 Методи оптимізації процесів CI/CD та вирішення проблем безпеки

У процесі побудови системи автоматизованого розгортання важливим етапом стає не лише початкова реалізація архітектури та налаштування конвеєра, а й подальша оптимізація, що базується на аналізі реальних помилок, обмежень і потенційних ризиків, які проявляються під час практичної експлуатації, при цьому накопичений досвід використання дозволяє виявити критичні аспекти, пов’язані з безпекою, управлінням ресурсами та загальною стабільністю функціонування системи, а впровадження сучасних підходів DevOps разом із дотриманням "best practices" забезпечує підвищення надійності інфраструктури та ефективності конвеєра розгортання, що в підсумку формує більш стійку та контрольовану систему, орієнтовану на безперервну інтеграцію і доставку змін.

У даному контексті основна увага зосереджується на трьох взаємопов’язаних напрямах, серед яких [31-32]:

— ізоляція середовища виконання;

— забезпечення мережевої безпеки;

— оптимізація використання ресурсів.

Причому кожен із цих напрямів розглядається як відповідь на конкретні проблеми, що виникали під час реалізації, і водночас як крок до подальшого вдосконалення архітектури системи, яка поступово наближається до рівня промислових рішень.

Ізоляція середовища: перехід від root до технічного користувача (Least Privilege). Однією з перших проблем, що виникла під час налаштування системи, стала неможливість запуску Self-hosted runner від імені користувача "root", причому така заборона є не випадковою, а закладеною у механізмах безпеки, оскільки виконання автоматизованих процесів із максимальними привілеями може

призвести до повного контролю над системою у випадку помилки або компрометації, у зв’язку з чим виникла необхідність переходу до більш безпечної моделі, що базується на принципі найменших привілеїв, відповідно до якого кожен процес отримує лише ті права, які необхідні для виконання його функцій [27].

Рішенням у даному випадку стало створення окремого технічного користувача, наприклад "runner", який використовується виключно для виконання CI/CD процесів, включаючи розгортання Docker-контейнерів та взаємодію з репозиторієм, при цьому сам процес налаштування включає створення нового користувача в системі, надання йому обмежених прав доступу, додавання до групи "docker" для можливості керування контейнерами без використання привілеїв адміністратора, а також налаштування доступу до необхідних директорій, що забезпечує коректну роботу всіх компонентів системи (див. рис. 3.1) [27].

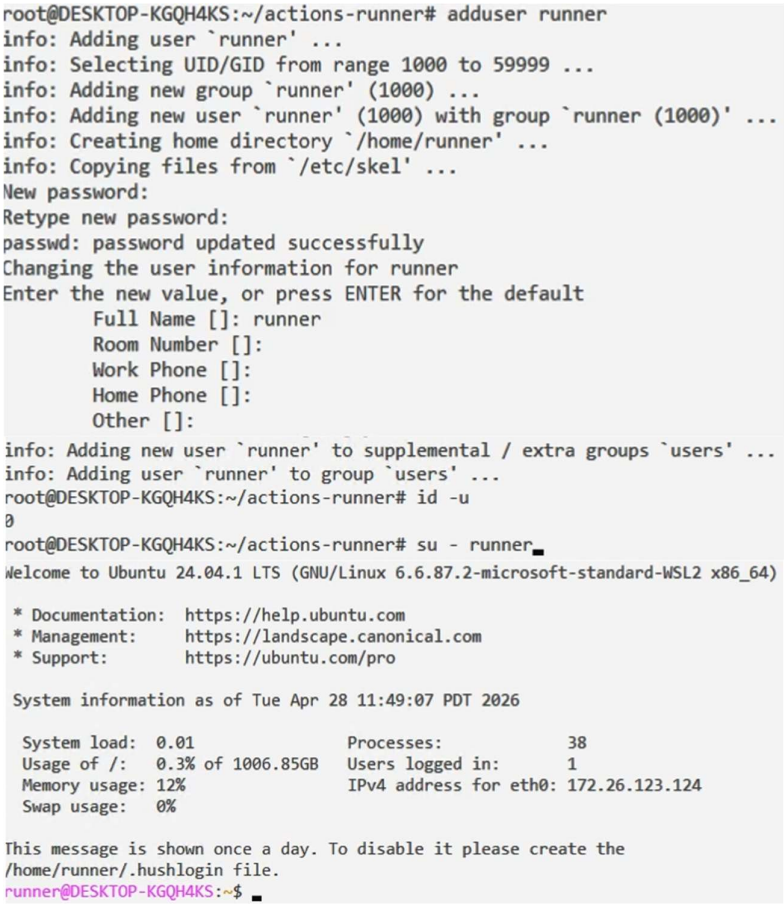


Рисунок 3.1 – Перехід від root до технічного користувача runner

Окрему увагу приділено міграції проєкту з директорії "/root/" до домашньої директорії нового користувача, наприклад "/home/runner/", причому важливим аспектом такого перенесення є збереження не лише основних файлів застосунку, а й прихованих директорій, зокрема ".git", які містять історію змін та забезпечують можливість подальшої синхронізації з віддаленим репозиторієм без повторної ініціалізації, що дозволяє зберегти цілісність проєкту та безперервність процесу розгортання (див. рис. 3.2).

```
root@DESKTOP-KGQH4KS:~# mv /root/app/* /home/runner/app/
root@DESKTOP-KGQH4KS:~# mv /root/app/.git /home/runner/app/
root@DESKTOP-KGQH4KS:~# chown -R runner:runner /home/runner/app
root@DESKTOP-KGQH4KS:~# su - runner
runner@DESKTOP-KGQH4KS:~$ cd /home/runner/app/
runner@DESKTOP-KGQH4KS:~/app$ ls -la
total 28
drwxrwxr-x 3 runner runner 4096 Apr 28 11:53 .
drwxr-x--- 4 runner runner 4096 Apr 28 11:52 ..
drwxr-xr-x 8 runner runner 4096 Apr 28 11:25 .git
-rw-r--r-- 1 runner runner  115 Apr 28 20:40 Dockerfile
-rw-r--r-- 1 runner runner  559 Apr 28 20:40 app.js
-rw-r--r-- 1 runner runner  254 Apr 28 20:40 docker-compose.yml
-rw-r--r-- 1 runner runner  119 Apr 28 20:40 package.json
runner@DESKTOP-KGQH4KS:~/app$
```

Рисунок 3.2 – Ізоляція середовища виконання шляхом мігрування

Логіку переходу до нової моделі доцільно уявити у вигляді послідовного процесу, де початково використовується root-користувач, після чого виявляються обмеження та ризики, далі створюється окремий користувач runner, йому надаються необхідні права та доступ до ресурсів, виконується міграція проєкту до нового середовища, після чого запускається Self-hosted runner вже від імені цього користувача, що в підсумку формує ізольоване та контрольоване середовище виконання.

У результаті впровадження такого підходу досягається чітка ізоляція процесів, де кожен компонент системи функціонує в межах визначених прав доступу, що значно знижує ризик пошкодження системи або витоку даних, а також

підвищує загальний рівень безпеки інфраструктури, забезпечуючи відповідність сучасним вимогам до побудови захищених CI/CD систем.

Мережева безпека: використання моделі Pull (вихідні з'єднання). Наступним важливим аспектом оптимізації виступає забезпечення мережевої безпеки середовища, що безпосередньо впливає на доступність і захищеність серверної інфраструктури, при цьому традиційний підхід до розгортання застосунків часто базується на використанні SSH-з'єднань, які передбачають відкриття порту 22 на сервері для прийому вхідних запитів, що автоматично збільшує поверхню атаки та створює додаткові ризики, серед яких можливість несанкціонованого доступу, підвищена ймовірність атак типу brute force, необхідність складного налаштування firewall, а також потенційні проблеми, пов'язані з NAT і обмеженнями мережевої маршрутизації.

Альтернативною моделлю виступає підхід, реалізований через використання Self-hosted runner, у якому сервер не очікує вхідних підключень, а самостійно ініціює вихідне з'єднання до віддаленого сервісу через захищений канал HTTPS по порту 443, у такій моделі взаємодії Runner встановлює постійне з'єднання із сервісом GitHub, після чого періодично перевіряє наявність нових завдань і у випадку їх появи отримує інструкції з "workflow" та виконує їх локально, що дозволяє повністю змінити підхід до організації мережевої взаємодії.

У результаті використання такої моделі усувається необхідність відкриття вхідних портів, що суттєво знижує ризики несанкціонованого доступу та мінімізує потенційні вектори атак, одночасно з цим зникають проблеми, пов'язані з NAT та складною конфігурацією мережевих правил, а використання HTTPS забезпечує стабільність та надійність з'єднання завдяки стандартним механізмам шифрування та автентифікації (див. рис. 3.3) [28].



Рисунок 3.3 – Метод захищеної взаємодії Self-hosted runner із зовнішнім сервісом

У підсумку застосування моделі вихідних з’єднань дозволяє сформувати більш захищену інфраструктуру автоматизованого розгортання, у якій зменшено кількість відкритих точок доступу, спрощено мережеву конфігурацію та забезпечено надійне виконання CI/CD процесів навіть у середовищах із обмеженим або складним мережевим доступом.

Автоматичне очищення невикористаних ресурсів. У процесі експлуатації системи було виявлено проблему накопичення невикористаних Docker-образів, оскільки кожна нова збірка створює окремий образ, який після оновлення контейнерів може залишатися в системі, поступово займаючи значний обсяг дискового простору, що особливо критично для середовищ із обмеженими ресурсами, таких як WSL2, де перевантаження файлової системи може призвести до зниження продуктивності або навіть до збоїв у роботі пайплайну.

Для вирішення цієї проблеми у процес автоматизованого розгортання інтегровано механізм автоматичного очищення ресурсів, реалізований через використання команди "docker image prune -f", яка дозволяє видаляти всі невикористані образи без необхідності ручного втручання, при цьому

використання параметра "-f" забезпечує виконання команди без додаткового підтвердження, що є критично важливим для повністю автоматизованих сценаріїв, де відсутня взаємодія з користувачем (див. рис. 3.4) [29].

```
runner@DESKTOP-KGQH4KS:~/actions-runner$ sudo docker images
                                                    i Info →  U  In Use
IMAGE            ID             DISK USAGE   CONTENT SIZE   EXTRA
app-web:latest   fb6ebd85186c        197MB         47.5MB    U
runner@DESKTOP-KGQH4KS:~/actions-runner$
```

Рисунок 3.4 –Актуальний Docker-образ після деплою

Логіку роботи такого механізму доцільно представити у вигляді послідовного процесу, де після завершення етапу розгортання виконується перевірка наявних Docker-образів, визначаються ті, що більше не використовуються контейнерами, після чого запускається команда очищення, яка видаляє зайві ресурси і звільняє дисковий простір, забезпечуючи стабільність подальшої роботи системи (див. рис. 3.5).



Рисунок 3.5 – Метод автоматичного очищення Docker-образів після кожної збірки

У результаті впровадження автоматичного очищення ресурсів досягається підтримка стабільного стану інфраструктури без накопичення зайвих даних, зменшується потреба у ручному адмініструванні та підвищується ефективність використання доступних ресурсів, а поєднання цього механізму з іншими оптимізаційними підходами, включаючи ізоляцію середовища та безпечну мережеву модель, формує комплексне рішення, орієнтоване на побудову надійної, безпечної та масштабованої системи автоматизованого розгортання.

3.2 Аналіз ефективності розробленої системи

Проведений аналіз ефективності реалізованої системи автоматизованого розгортання дозволяє оцінити не лише технічні характеристики, але й практичну доцільність застосування у порівнянні з традиційними підходами адміністрування, при цьому розглянута архітектура базується на інтеграції CI/CD-конвеєра із використанням Self-hosted runner у середовищі WSL2, що формує замкнутий цикл доставки змін від репозиторію до середовища виконання, у межах якого проведено:

1. тестовий сценарій оновлення застосунку;
2. аналіз журналів виконання пайплайнів;
3. виявлення потенційних вузьких місць.

Все це може впливати на стабільність і продуктивність системи, а також визначення факторів, що впливають на загальну ефективність інфраструктури.

У процесі аналізу особлива увага приділяється таким аспектам, як швидкість виконання операцій розгортання, стабільність роботи при повторних ітераціях, рівень автоматизації процесів, а також ступінь залежності від людського фактору, що дозволяє сформувати комплексне уявлення про ефективність запропонованого рішення, при цьому важливим елементом оцінки виступає аналіз логів виконання, який дає змогу відстежити кожен етап пайплайну та визначити потенційні точки затримок або помилок, що виникають під час виконання сценаріїв розгортання.

Окремо розглядається питання ресурсоефективності, оскільки використання контейнеризації разом із автоматизованими механізмами очищення дозволяє підтримувати стабільний стан системи навіть у середовищах з обмеженими ресурсами, таких як WSL2, де надмірне накопичення Docker-образів може призводити до деградації продуктивності, тому інтеграція механізмів оптимізації забезпечує довготривалу стабільність роботи системи без необхідності ручного втручання, що в свою чергу підвищує загальний рівень надійності інфраструктури.

Таким чином, аналіз ефективності демонструє, що побудована система відповідає сучасним вимогам до автоматизації процесів розгортання, забезпечує високу швидкість доставки змін, мінімізує кількість помилок та дозволяє масштабувати інфраструктуру без суттєвого ускладнення її архітектури, що створює основу для подальшого розвитку та інтеграції більш складних механізмів управління сервісами.

Порівняння автоматизованого підходу з ручним адмініструванням. Традиційна модель адміністрування передбачає виконання більшості операцій вручну, що включає:

1. підключення до сервера через SSH.
2. перехід у робочу директорію;
3. оновлення коду;
4. збірку застосунку;
5. перезапуск сервісів;
6. подальшу перевірку їх працездатності.

При цьому кожен із цих етапів залежить від дій адміністратора та може виконуватися з різною точністю і швидкістю, що створює значний вплив людського фактору на кінцевий результат, а також підвищує ризик виникнення помилок.

Автоматизований підхід, реалізований у межах даної системи, принципово змінює цю модель, оскільки після внесення змін у код та виконання push-запиту всі наступні дії виконуються без участі адміністратора, де CI/CD-пайплайн ініціює процеси оновлення коду, збірки контейнерів та їх перезапуску, забезпечуючи

повністю автоматизований цикл розгортання, що дозволяє значно скоротити час виконання операцій та підвищити їх передбачуваність.

З точки зору часових витрат ручний підхід може займати від кількох хвилин до десятків хвилин залежно від складності системи та досвіду адміністратора, при цьому автоматизована модель забезпечує виконання тих самих операцій у значно коротший термін, який зазвичай обмежується кількома хвилинами або навіть секундами у випадку незначних змін, причому цей час залишається стабільним незалежно від кількості ітерацій, що є важливою перевагою у контексті безперервної інтеграції (див. табл. 3.1).

Таблиця 3.1 – Порівняння ручного та автоматизованого розгортання

Критерій	Ручне розгортання	Автоматизоване розгортання
Час виконання	Хвилини–десятки хвилин	Секунди–хвилини
Ймовірність помилок	Висока (людський фактор)	Мінімальна
Відтворюваність	Нестабільна	Повна
Складність	Висока	Нижча після налаштування
Масштабованість	Обмежена	Висока
Контроль процесу	Ручний	Автоматизований
Залежність від користувача	Висока	Відсутня

Порівняння логіки виконання процесів демонструє суттєву різницю між підходами, де у ручному сценарії послідовність виглядає як підключення до сервера, перехід у директорію, оновлення коду, збірка застосунку, перезапуск сервісів та перевірка результату, тоді як автоматизований сценарій обмежується виконанням push-запиту, після якого відбувається автоматичний запуск пайплайну, виконання всіх необхідних операцій Self-hosted runner та отримання готового результату у вигляді оновленого сервісу (див. рис. 3.6).

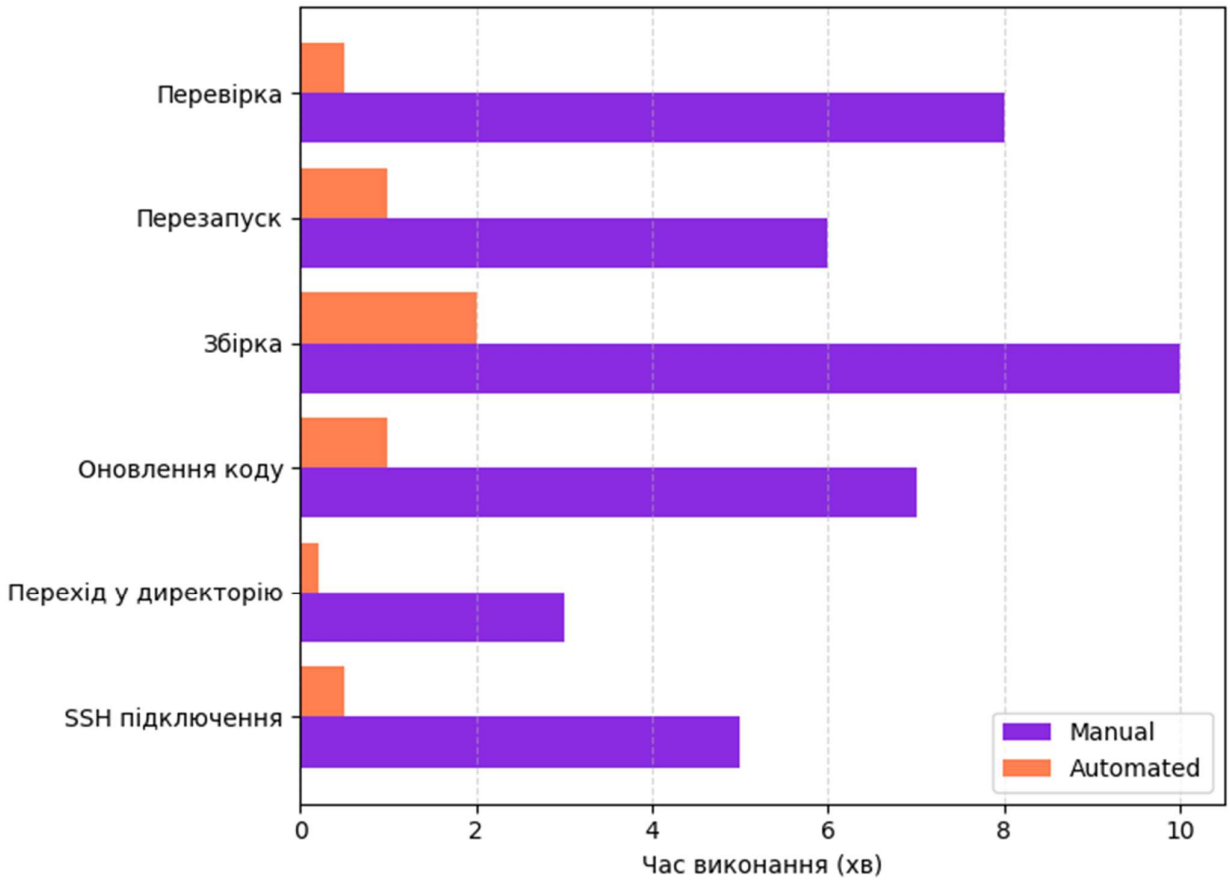


Рисунок 3.6 – Гістограма виконання процесів розгортання

Важливою характеристикою автоматизованого підходу є відтворюваність процесів, оскільки всі дії описані у конфігураційному файлі "workflow" та виконуються однаково при кожному запуску, що виключає варіативність результатів, характерну для ручного адміністрування, де різні користувачі можуть виконувати ті самі операції по-різному, що призводить до непередбачуваних наслідків.

Зниження ризику помилок також виступає суттєвою перевагою автоматизації, оскільки у ручному режимі навіть незначні неточності, такі як неправильна команда або пропущений етап, можуть призвести до некоректної роботи системи, тоді як автоматизований підхід базується на заздалегідь визначеному сценарії, що мінімізує можливість виникнення таких ситуацій та забезпечує стабільність виконання процесів.

Додатковим фактором є підвищення зручності експлуатації, оскільки відсутня необхідність постійного підключення до сервера, а взаємодія з системою здійснюється через репозиторій, що значно спрощує процес управління інфраструктурою та дозволяє зосередитися на розробці функціоналу замість виконання рутинних операцій.

У контексті масштабування автоматизований підхід також демонструє переваги, оскільки додавання нових сервісів або зміна конфігурації не потребує суттєвих змін у процесі розгортання, адже всі дії описані декларативно у конфігураційних файлах, що забезпечує гнучкість та спрощує розширення системи, формуючи основу для подальшого переходу до більш складних оркестраційних рішень та розподілених інфраструктур.

Аналіз використання GitHub-hosted проти Self-hosted runner-ів. Одним із визначальних рішень у процесі проєктування системи автоматизованого розгортання стало використання Self-hosted runner замість стандартного GitHub-Hosted Runner, при цьому обґрунтування такого вибору базується на детальному аналізі особливостей кожного підходу та їх впливу на продуктивність, гнучкість і безпеку інфраструктури, де GitHub-hosted runner представляє собою віртуальне середовище, що надається платформою та автоматично масштабується залежно від навантаження, але водночас має ряд обмежень, серед яких:

1. фіксовані ліміти часу виконання задач;
2. обмеження на використання обчислювальних ресурсів;
3. відсутність прямого доступу до локальної інфраструктури;
4. необхідність передачі даних через зовнішні канали.

Все це складнює інтеграцію з середовищем виконання застосунку.

Self-hosted runner у даному випадку розгортається безпосередньо у цільовому середовищі, зокрема у середовищі WSL2, що дозволяє забезпечити повний контроль над виконанням процесів та використанням ресурсів системи, при цьому відсутні обмеження на використання CPU, оперативної пам’яті та дискового простору, оскільки runner працює на локальній машині, що є критично важливим

для контейнеризованих застосунків, де процеси збірки Docker-образів можуть бути ресурсоємними та вимагати значних обчислювальних потужностей.

Важливим аспектом виступає інтеграція з локальною інфраструктурою, де GitHub-hosted runner не має прямого доступу до внутрішніх ресурсів системи, що потребує додаткових механізмів передачі даних та налаштування мережевої взаємодії, тоді як Self-hosted runner працює безпосередньо у середовищі, де виконується застосунок, що дозволяє уникнути зайвих мережевих операцій та спрощує процес розгортання, забезпечуючи швидке виконання команд та мінімальні затримки.

З точки зору безпеки Self-hosted runner демонструє додаткові переваги завдяки використанню моделі pull, при якій ініціація з'єднання здійснюється з боку сервера через захищений HTTPS-канал, що виключає необхідність відкриття вхідних портів, на відміну від традиційних підходів, де використання зовнішніх підключень може збільшувати поверхню атаки, таким чином забезпечується більш контрольоване та ізольоване середовище виконання.

Окрему увагу привертає можливість кастомізації середовища виконання, де Self-hosted runner дозволяє встановлювати необхідні залежності, налаштовувати конфігурацію системи та адаптувати її під конкретні потреби проєкту, що забезпечує високу гнучкість та сумісність із різними інструментами, тоді як у випадку GitHub-hosted runner така гнучкість обмежена стандартними конфігураціями, що можуть не відповідати специфічним вимогам застосунку.

Архітектурна різниця між підходами проявляється у структурі взаємодії компонентів, де у випадку GitHub-hosted runner репозиторій взаємодіє з хмарним середовищем виконання, яке у свою чергу встановлює з'єднання із сервером, що додає додатковий рівень взаємодії та потенційні точки відмови, тоді як Self-hosted runner розташовується безпосередньо на сервері, забезпечуючи локальне виконання всіх операцій, що спрощує архітектуру та зменшує затримки, оскільки виключається необхідність передачі даних через зовнішні мережі.

Разом із перевагами використання Self-hosted runner накладає певні вимоги до підтримки інфраструктури, включаючи:

1. необхідність забезпечення постійної доступності;
2. оновлення програмного забезпечення;
3. контролю безпеки.

Все те, що у середовищі, WSL2 вирішується шляхом налаштування runner як системної служби, що забезпечує його автоматичний запуск та стабільну роботу без ручного втручання, при цьому залежність від локального середовища означає, що доступність системи пов’язана зі станом фізичного пристрою, на якому розгорнуто runner (див. табл. 3.2).

Таблиця 3.2 – Порівняння GitHub-hosted та Self-hosted runner-ів

Критерій	GitHub-hosted runner	Self-hosted runner
Доступ до ресурсів	Обмежений	Повний
Продуктивність	Обмежена середовищем	Залежить від локальної машини
Безпека	Хмарна ізоляція	Контрольована локально
Мережева модель	Вхідні/зовнішні взаємодії	Pull через HTTPS
Інтеграція з інфраструктурою	Обмежена	Повна
Кастомізація	Обмежена	Повна
Затримки	Можливі	Мінімальні
Підтримка	Не потребує адміністрування	Потребує обслуговування
Масштабування	Автоматичне	Залежить від ресурсів

У підсумку вибір Self-hosted runner виступає обґрунтованим для інфраструктур, де пріоритетами є контроль, продуктивність та безпека, оскільки такий підхід дозволяє реалізувати автономну систему автоматизованого розгортання без залежності від обмежень сторонніх платформ, забезпечуючи високу швидкість виконання, гнучкість налаштування та інтеграцію з локальним середовищем, що у сукупності демонструє перевагу над традиційними та хмарними підходами у більшості ключових аспектів функціонування CI/CD-конвеєра.

3.3 Вектори масштабування інфраструктури

Розроблена система автоматизованого розгортання, що базується на контейнеризації та використанні CI/CD-конвеєра із Self-hosted runner, демонструє високу ефективність у межах невеликих проєктів або локальних і тимчасових середовищ, однак із зростанням складності інфраструктури поступово проявляються обмеження, пов’язані з відсутністю повноцінного горизонтального масштабування, автоматичного відновлення сервісів, балансування навантаження та централізованого управління великою кількістю контейнерів, що ускладнює підтримку стабільності системи при зростанні кількості користувачів або сервісів, у зв’язку з чим виникає необхідність переходу до більш складних підходів оркестрації контейнерів, серед яких найбільш поширеним виступає Kubernetes, що дозволяє реалізувати кластерну модель управління інфраструктурою.

Аналіз векторів масштабування передбачає розгляд можливості еволюційного переходу від локальної архітектури до розподіленої кластерної моделі, де основна ідея полягає у збереженні базових принципів автоматизації розгортання з одночасним розширенням функціональних можливостей системи, що дозволяє адаптувати інфраструктуру до більш складних умов експлуатації без повної перебудови її логіки.

Перспективи міграції від Docker Compose до кластерів Kubernetes. Docker Compose у поточній реалізації виступає зручним інструментом для опису багатоконтейнерних застосунків у межах одного конфігураційного файлу, що дозволяє швидко розгортати сервіси, керувати їх взаємодією та забезпечувати базову доступність застосунку, при цьому використання обмежується локальним середовищем та не передбачає складних сценаріїв масштабування, оскільки зі зростанням навантаження або кількості компонентів системи з’являються суттєві обмеження, серед яких:

1. відсутність вбудованих механізмів автоматичного масштабування;
2. обмежена підтримка балансування навантаження;

3. відсутність централізованого управління кластером.

У випадку збільшення кількості користувачів або сервісів ручне масштабування через дублювання контейнерів стає неефективним і складним в адмініструванні, що створює передумови для переходу до систем оркестрації, де Kubernetes забезпечує автоматизацію розгортання, масштабування та управління контейнерами у межах кластера, який складається з декількох вузлів, що дозволяє суттєво підвищити гнучкість і відмовостійкість інфраструктури [30].

Архітектурна різниця між підходами проявляється у структурі системи, де у випадку Docker Compose використовується один сервер із локально запущеними контейнерами, що керуються через CI/CD runner, тоді як у кластерній моделі Kubernetes застосовується група серверів, що включає керуючий вузол та виконавчі вузли, між якими розподіляється навантаження, у результаті чого система отримує здатність продовжувати роботу навіть у випадку відмови окремих компонентів [30].

Однією з ключових переваг Kubernetes виступає механізм горизонтального масштабування, який дозволяє автоматично змінювати кількість екземплярів застосунку залежно від поточного навантаження, що забезпечує ефективне використання ресурсів та стабільну продуктивність системи, тоді як у Docker Compose така функціональність відсутня і потребує ручного втручання.

Додатково важливим аспектом є механізм самовідновлення, де у Kubernetes кожен контейнер запускається у межах спеціалізованої сутності, і у випадку збою система автоматично перезапускає або створює новий екземпляр, що дозволяє мінімізувати простої та підтримувати безперервну доступність сервісу, на відміну від локального підходу, де відновлення потребує зовнішнього контролю або додаткових скриптів.

Балансування навантаження також реалізується на рівні платформи через використання сервісів та ingress-механізмів, що дозволяє рівномірно розподіляти запити між доступними екземплярами застосунку, забезпечуючи стабільну роботу при високому навантаженні та ефективну обробку великої кількості клієнтів [30].

У контексті міграції важливим виступає поступовий підхід, що включає використання вже існуючого "Dockerfile" для контейнеризації застосунку, трансформацію конфігурації Docker Compose у Kubernetes-маніфести, налаштування кластера, інтеграцію CI/CD-процесів із використанням нових інструментів розгортання та подальше тестування і оптимізацію системи, при цьому загальна логіка пайплайну зберігається, але змінюється набір команд, де замість Docker Compose використовуються інструменти взаємодії з кластером.

Інтеграція CI/CD у такій архітектурі передбачає перехід від локального розгортання до публікації контейнерних образів у реєстр та подальшого застосування конфігурацій до кластера, що дозволяє зберегти автоматизований характер процесів та адаптувати їх до розподіленого середовища (див. рис. 3.7).



Рисунок 3.7 – Процес міграції від Docker Compose до Kubernetes-кластера

Додатковою перевагою Kubernetes виступає підвищений рівень ізоляції та безпеки завдяки використанню механізмів управління доступом, простору імен та секретів, що забезпечує більш детальний контроль над інфраструктурою та її компонентами, що є особливо важливим у складних системах із великою кількістю сервісів.

Разом із цим перехід до Kubernetes супроводжується певними складнощами, серед яких складність початкового налаштування, необхідність глибшого розуміння архітектури, підвищені вимоги до ресурсів та потреба у використанні додаткових інструментів для управління та моніторингу, однак у довгостроковій перспективі такі витрати компенсуються значним підвищенням гнучкості, масштабованості та надійності системи [30].

ВИСНОВКИ

У результаті проєктування та розробки сформовано цілісну архітектуру автоматизованого розгортання, що поєднує репозиторій, механізм виконання на виділеному середовищі та контейнерну платформу. Взаємодія між компонентами організована таким чином, що передача змін ініціює виконання пайплайну без додаткового втручання, а використання конфігураційних файлів дозволяє формалізувати процеси розгортання та забезпечити їх відтворюваність.

Налаштування цільового середовища забезпечило стабільну основу для виконання автоматизованих сценаріїв, де використання ізольованого середовища та розгортання виконавчого механізму у вигляді системної служби дозволило досягти безперервності роботи та автономності виконання. Організація доступів, підключення до контейнерної платформи та інтеграція з системою керування версіями забезпечили коректну взаємодію між усіма елементами інфраструктури.

Реалізовані алгоритми функціонування конвеєра дозволили автоматизувати ключові етапи розгортання, включаючи ініціацію збірки контейнерних образів та їх подальше оновлення без простою. Проведене тестування підтвердило коректність роботи системи, зокрема автоматичне оновлення застосунку після внесення змін, стабільне виконання пайплайну та ефективне управління ресурсами, що дозволяє підтримувати працездатність середовища без накопичення зайвих даних.

Подальший аналіз дозволив визначити напрями оптимізації та розвитку системи, зокрема впровадження принципу найменших привілеїв, використання безпечної мережевої моделі із вихідними з’єднаннями та автоматизацію очищення ресурсів. Порівняння з ручним підходом підтвердило переваги автоматизації за швидкістю та надійністю, а вибір локального виконавчого механізму забезпечив більший контроль над інфраструктурою. Розглянуті вектори масштабування демонструють можливість переходу до кластерних рішень.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Continuous Integration Explained. Atlassian. URL: https://www.atlassian.com/continuous-delivery/continuous-integration (дата звернення: 25.02.2026).
2. What is CI/CD? Red Hat. URL: https://www.redhat.com/en/topics/devops/what-is-ci-cd (дата звернення: 01.03.2026).
3. Software Development Life Cycle (SDLC). IBM. URL: https://www.ibm.com/topics/sdlc (дата звернення: 05.03.2026).
4. CI/CD Pipeline Overview. GitLab Docs. URL: https://docs.gitlab.com/ee/ci/introduction (дата звернення: 09.03.2026).
5. GitHub Actions Documentation. GitHub. URL: https://docs.github.com/en/actions (дата звернення: 13.03.2026).
6. GitLab CI/CD Overview. GitLab. URL: https://about.gitlab.com/stages-devops-lifecycle/continuous-integration (дата звернення: 17.03.2026).
7. What is Containerization? Docker. URL: https://www.docker.com/resources/what-container (дата звернення: 21.03.2026).
8. Docker Architecture. Docker Docs. URL: https://docs.docker.com/get-started/overview (дата звернення: 25.03.2026).
9. Docker Engine Components. Docker. URL: https://docs.docker.com/engine (дата звернення: 29.03.2026).
10. Docker Compose Overview. Docker Docs. URL: https://docs.docker.com/compose (дата звернення: 02.04.2026).
11. What is Git? Git SCM. URL: https://git-scm.com/docs/gittutorial (дата звернення: 06.04.2026).
12. Infrastructure as Code. HashiCorp. URL: https://www.hashicorp.com/resources/what-is-infrastructure-as-code (дата звернення: 10.04.2026).
13. Git Branching Strategies. Atlassian. URL: https://www.atlassian.com/git/tutorials/comparing-workflows (дата звернення: 14.04.2026).
14. GitHub Workflow Triggers. GitHub Docs. URL: https://docs.github.com/en/actions/using-workflows/events-that-trigger-workflows (дата звернення: 18.04.2026).
15. GitHub Secrets Security. GitHub Docs. URL: https://docs.github.com/en/actions/security-guides/encrypted-secrets (дата звернення: 22.04.2026).
16. DevOps Security Best Practices. AWS. URL: https://aws.amazon.com/devops/security (дата звернення: 26.04.2026).
17. CI/CD Architecture Patterns. Microsoft. URL: https://learn.microsoft.com/en-us/azure/devops/learn/devops-at-microsoft (дата звернення: 30.04.2026).

18. Self-hosted Runners Guide. GitHub Docs. URL: https://docs.github.com/en/actions/hosting-your-own-runners (дата звернення: 03.05.2026).
19. Docker Compose File Reference. Docker Docs. URL: https://docs.docker.com/compose/compose-file (дата звернення: 05.05.2026).
20. WSL2 Overview. Microsoft. URL: https://learn.microsoft.com/en-us/windows/wsl (дата звернення: 25.02.2026).
21. Docker Desktop Setup. Docker Docs. URL: https://docs.docker.com/desktop (дата звернення: 01.03.2026).
22. Runner as Service. GitHub Docs. URL: https://docs.github.com/en/actions/hosting-your-own-runners/configuring-the-self-hosted-runner-application-as-a-service (дата звернення: 05.03.2026).
23. Docker Build Process. Docker Docs. URL: https://docs.docker.com/build (дата звернення: 09.03.2026).
24. Docker Compose Up Command. Docker Docs. URL: https://docs.docker.com/compose/reference/up (дата звернення: 13.03.2026).
25. CI/CD Testing Pipelines. CircleCI. URL: https://circleci.com/docs/pipelines (дата звернення: 17.03.2026).
26. Debugging GitHub Actions. GitHub Docs. URL: https://docs.github.com/en/actions/monitoring-and-troubleshooting-workflows (дата звернення: 21.03.2026).
27. Principle of Least Privilege. NIST. URL: https://csrc.nist.gov/glossary/term/least_privilege (дата звернення: 25.03.2026).
28. HTTPS Security Overview. Cloudflare. URL: https://www.cloudflare.com/learning/ssl/what-is-https (дата звернення: 29.03.2026).
29. Docker Cleanup Commands. Docker Docs. URL: https://docs.docker.com/config/pruning (дата звернення: 02.04.2026).
30. Kubernetes Overview. Kubernetes Docs. URL: https://kubernetes.io/docs/concepts/overview (дата звернення: 05.05.2026).
31. Іващенко Р. М., Лащевська Н. О. Система автоматизованого розгортання та багатофакторної автентифікації контейнеризованих застосунків із використанням Docker Compose та GitHub Actions. «Сучасний стан та перспективи розвитку IoT» : VII Науково-техн. конф., м. Київ, 20 квітня 2026 р. С.294-296.
32. Іващенко Р. М., Лащевська Н. О. Безпечна автоматизація розгортання контейнеризованих застосунків із використанням CI/CD та управління секретами. «Застосування програмного забезпечення в ІКТ» : VII Всеукр. науково-техн. конф., м. Київ, 23 квітня 2026 р. (подано до публікації).

Додаток А

Вихідний код файлу app.js

```
const express = require('express');
const app = express();
const PORT = 8080;

app.get('/', (req, res) => {
    res.send(`
        <h1>Сайт працює. Версія 0.1.1</h1>
        <p><strong>Виконавець:</strong> Іващенко Роман, Студент групи КІД-41</p>
        <p>Система автоматизованого розгортання успішно функціонує на базі
Docker.</p>
    `);
});

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

Додаток Б

Конфігурація залежностей package.json

```
{
  "name": "my-web-app",
  "version": "1.0.0",
  "main": "app.js",
  "dependencies": {
    "express": "^4.18.2"
  }
}
}
```

Додаток В

Сценарій збірки Dockerfile

```
FROM node:18-alpine
WORKDIR /app
COPY package*.json ./
RUN npm install
COPY . .
EXPOSE 8080
CMD ["node", "app.js"]
}
```

Додаток Г

Конфігурація середовища docker-compose.yml

```
version: '3.8'
services:
  web:
    build: .
    container_name: web_app
    ports:
      - "80:8080"
    restart: always
    healthcheck:
      test: ["CMD", "curl", "-f", "http://localhost:8080"]
      interval: 30s
      timeout: 10s
      retries: 3
}
```

Додаток Д

Пайплайн розгортання .github/workflows/deploy.yml

```
name: Deploy to VPS

on:
  push:
    branches: [ main ]

jobs:
  deploy:
    runs-on: self-hosted
    steps:
      - name: Checkout code
        uses: actions/checkout@v3

      - name: Deploy application
        run: |
          # Specify path to folder
          cd /home/runner/app

          # Updating the code
          git pull origin main

          # Launching containers
          docker compose build
          docker compose up -d

          # Clearing cache
          docker image prune -f
}
```

Додаток Е

ДЕМОНСТРАЦІЙНІ МАТЕРІАЛИ

(Презентація)

ДЕРЖАВНИЙ УНІВЕРСИТЕТ
ІНФОРМАЦІЙНО-КОМУНІКАЦІЙНИХ ТЕХНОЛОГІЙ
НАВЧАЛЬНО-НАУКОВИЙ ІНСТИТУТ
ІНФОРМАЦІЙНИХ ТЕХНОЛОГІЙ
Кафедра комп'ютерної інженерії



КВАЛІФІКАЦІЙНА РОБОТА

на здобуття освітнього ступеня бакалавра

НА ТЕМУ: «СИСТЕМА АВТОМАТИЗОВАНОГО РОЗГОРТАННЯ КОНТЕЙНЕРИЗОВАНИХ ЗАСТОСУНКІВ»

Виконав студент групи КІД-41
Іващенко Роман Миколайович
Керівник кваліфікаційної роботи:
Лащевська Наталія Олександрівна,
канд. техн. наук, доц.

Київ – 2026

Мета, об’єкт, предмет дослідження

Мета роботи – мінімізація часу розгортання та підвищення надійності програмних застосунків за рахунок використання принципів CI/CD, контейнеризації та автоматизованих конвеєрів доставки

Об’єкт дослідження – процес автоматизованого розгортання програмних застосунків у серверному середовищі

Предмет дослідження – засоби та технології побудови CI/CD конвеєрів, контейнеризації та автоматизації розгортання із використанням Docker і GitHub Actions

Актуальність роботи

У роботі представлено теоретичні основи автоматизації розгортання застосунків, включаючи концепції CI/CD, життєвий цикл програмного забезпечення, принципи контейнеризації та використання систем контролю версій. Розглянуто архітектуру Docker, підходи до оркестрації контейнерів через Docker Compose, а також моделі організації пайплайнів і механізми забезпечення безпеки, зокрема управління доступами та секретами у процесі автоматизації.

Особливу увагу приділено розробці системи автоматизованого розгортання із використанням GitHub Actions та Self-hosted runner, налаштуванню середовища виконання, реалізації алгоритмів збірки образів і безшовного оновлення контейнерів, а також тестуванню працездатності пайплайну. Додатково розглянуто методи оптимізації, питання безпеки, ефективності та можливості масштабування інфраструктури з переходом до кластерних рішень.

Аналіз існуючих рішень

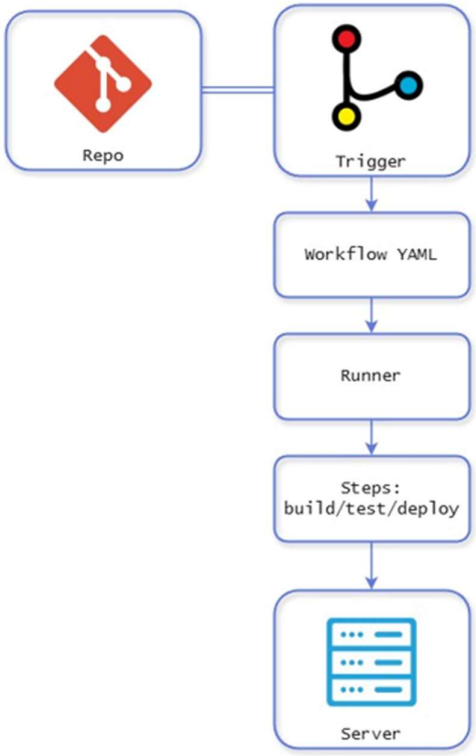


Рис. 1.4 – Взаємодія репозиторію та пайплайну розгортання

Табл. 1.1 – Порівняння інструментів автоматизації CI/CD (GitHub Actions, GitLab CI)

Критерій	GitHub Actions	GitLab CI/CD
Тип середовища	Хмарне та підтримка self-hosted	Хмарне та можливість повністю локального розгортання
Інтеграція	Глибока інтеграція з GitHub	Повна інтеграція в екосистему GitLab
Конфігурація	YAML (workflow)	YAML (.gitlab-ci.yml)
Тригери	Push, pull request, schedule, events	Push, merge request, schedule
Runner-и	GitHub-hosted та self-hosted	Shared та self-hosted
Робота з контейнерами	Вбудована підтримка Docker	Вбудована підтримка Docker та registry
Управління секретами	GitHub Secrets	GitLab CI/CD Variables
Масштабування	Автоматичне (хмарне)	Автоматичне та власна інфраструктура
Обмеження	Ліміти часу та ресурсів	Залежить від тарифу або інфраструктури

Загальна архітектура системи

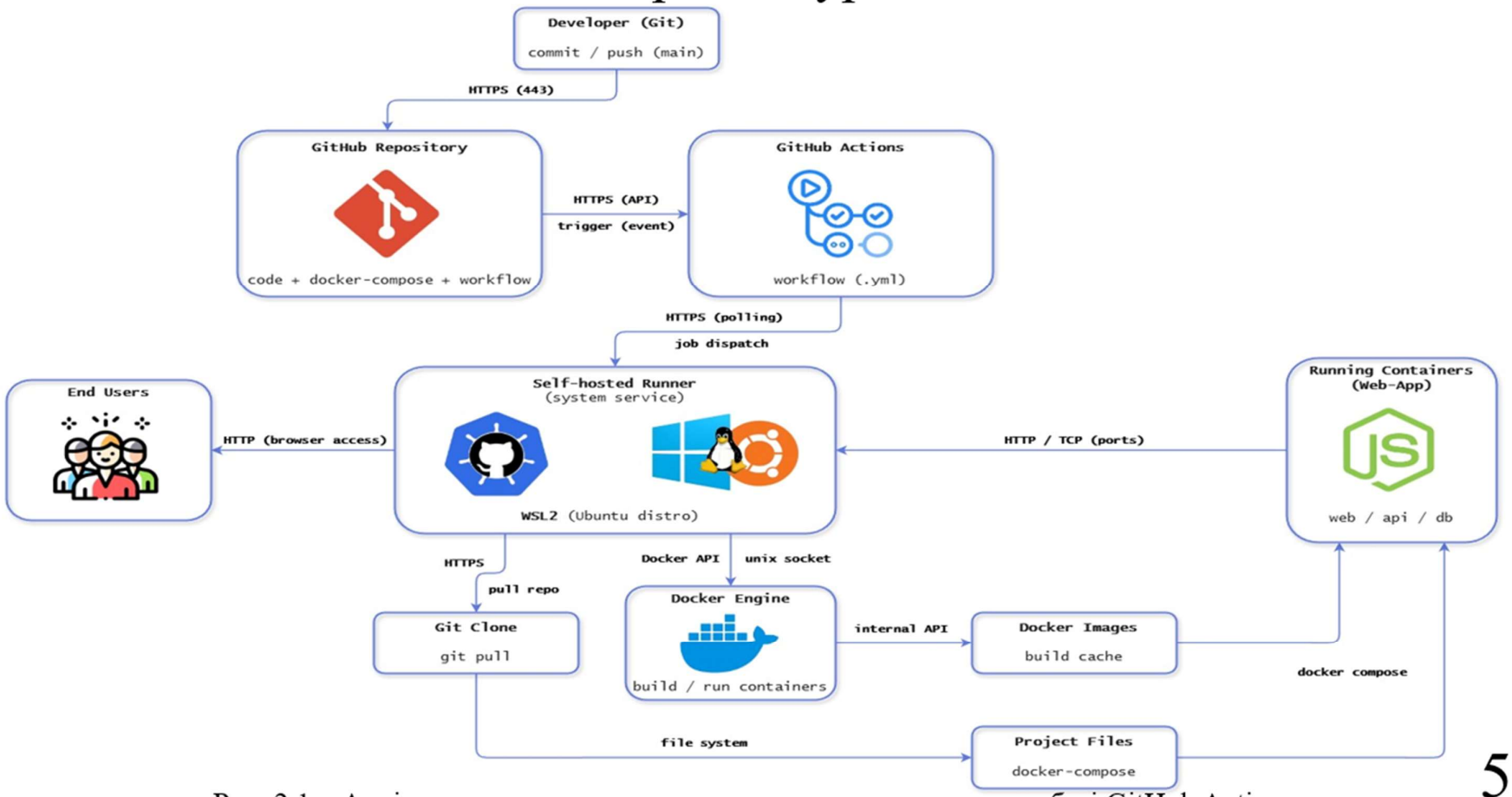


Рис. 2.1 – Архітектура системи автоматизованого розгортання на базі GitHub Actions

5

Конфігурація системи

runner@DESKTOP-KGQH4KS: ~

```
runner@DESKTOP-KGQH4KS:~$ tree -f app app/.github/
app
├── app/Dockerfile
├── app/app.js
├── app/docker-compose.yml
└── app/package.json
app/.github
└── app/.github/workflows
    └── app/.github/workflows/deploy.yml

3 directories, 5 files
runner@DESKTOP-KGQH4KS:~$
```

Рис. 2.2 – Логічна структура проєкту

runner@DESKTOP-KGQH4KS: ~

```
runner@DESKTOP-KGQH4KS:~$ cat app/docker-compose.yml
version: '3.8'
services:
  web:
    build: .
    container_name: web_app
    ports:
      - "80:8080"
    restart: always
    healthcheck:
      test: ["CMD", "curl", "-f", "http://localhost:8080"]
      interval: 30s
      timeout: 10s
      retries: 3
runner@DESKTOP-KGQH4KS:~$
```

Рис. 2.3 – Приклад конфігурації docker-compose.yml

Select runner@DESKTOP-KGQH4KS: ~

```
runner@DESKTOP-KGQH4KS:~$ cat app/.github/workflows/deploy.yml
name: Deploy to VPS

on:
  push:
    branches: [ main ]

jobs:
  deploy:
    runs-on: self-hosted
    steps:
      - name: Checkout code
        uses: actions/checkout@v3

      - name: Deploy application
        run: |
          # Specify path to folder
          cd /home/runner/app

          # Updating the code
          git pull origin main

          # Launching containers
          docker compose build
          docker compose up -d

          # Clearing cache
          docker image prune -f
runner@DESKTOP-KGQH4KS:~$
```

Рис. 2.4 – Приклад конфігурації deploy.yml

Налаштування середовища

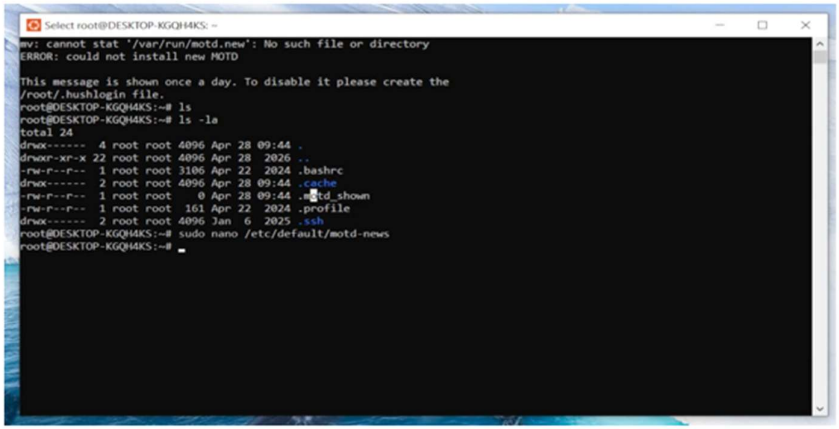


Рис. 2.5 – Підготовлене середовище WSL2

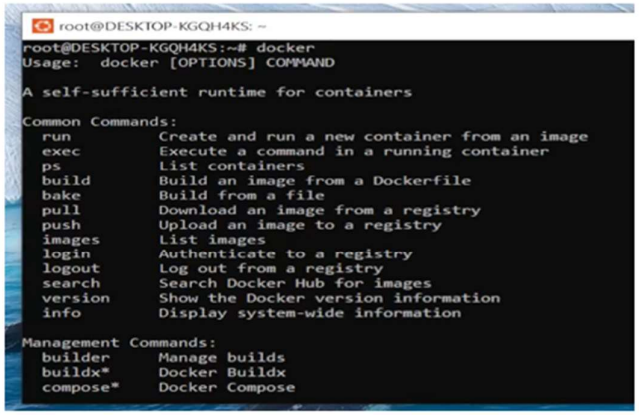


Рис. 2.6 – CLI інтерфейс Docker у терміналі WSL2

```
runner@DESKTOP-KGQH4KS:~$ mkdir actions-runner && cd actions-runner
runner@DESKTOP-KGQH4KS:~/actions-runner$ curl -o actions-runner-linux-x64-2.334.0.tar.gz -L https://github.c
om/actions/runner/releases/download/v2.334.0/actions-runner-linux-x64-2.334.0.tar.gz
  % Total    % Received % Xferd  Average Speed   Time    Time     Time  Current
                                 Dload  Upload   Total   Spent    Left  Speed
  0     0    0     0    0     0      0      0 --:--:-- --:--:-- --:--:--     0
100  214M  100  214M    0     0  1297k      0  0:02:49  0:02:49 --:--:-- 1038k
runner@DESKTOP-KGQH4KS:~/actions-runner$ echo "048024cd2c848eb6f14d5646d56c13a4def2ae7ee3ad12122bee960c56f3d
271  actions-runner-linux-x64-2.334.0.tar.gz" | shasum -a 256 -c
actions-runner-linux-x64-2.334.0.tar.gz: OK
runner@DESKTOP-KGQH4KS:~/actions-runner$ tar xzf ./actions-runner-linux-x64-2.334.0.tar.gz
runner@DESKTOP-KGQH4KS:~/actions-runner$
```

```
√ Connected to GitHub

# Runner Registration

Enter the name of the runner group to add this runner to: [press Enter for Default]

Enter the name of runner: [press Enter for DESKTOP-KGQH4KS]

This runner will have the following labels: 'self-hosted', 'Linux', 'X64'
Enter any additional labels (ex. label-1,label-2): [press Enter to skip] self-hosted

√ Runner successfully added

# Runner settings

Enter name of work folder: [press Enter for _work]

√ Settings Saved.

runner@DESKTOP-KGQH4KS:~/actions-runner$
```

Рис. 2.8 – Встановлення Self-Hosted Runner

Алгоритм роботи CI/CD

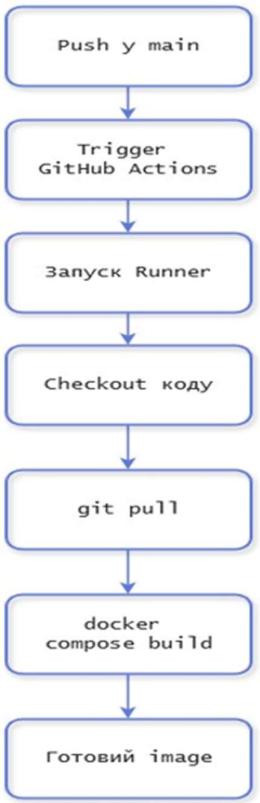


Рис. 2.11 – Алгоритм ініціації збірки Docker-образу

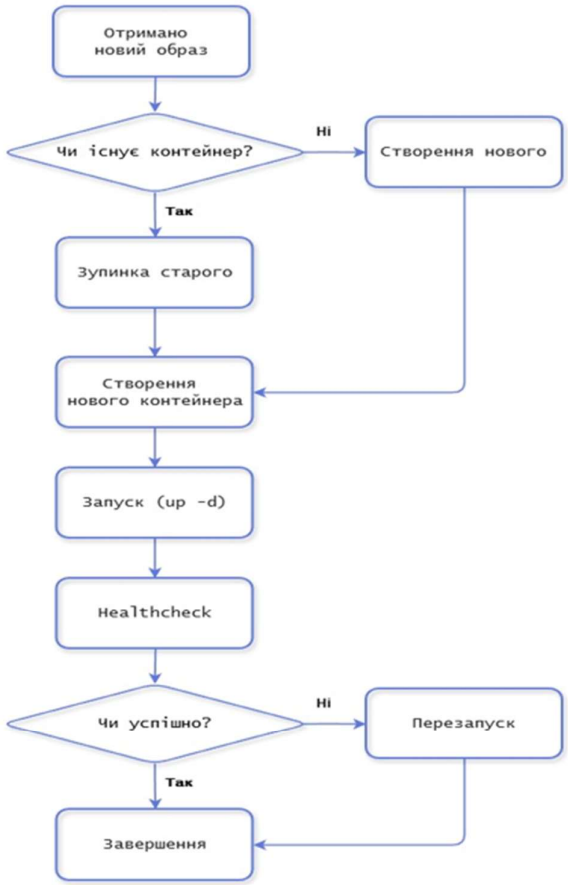


Рис. 2.12 – Алгоритм оновлення контейнерів без простою

Табл. 2.2 – Послідовність виконання етапів CI/CD пайплайну

Крок	Дія	Результат
1	Push у репозиторій	Ініціація тригера
2	Запуск workflow	Активується пайплайн
3	Підключення runner	Отримання задачі
4	git pull	Оновлення коду
5	Перехід у директорію	Підготовка середовища
6	docker compose build	Створення нового образу
7	docker compose up -d	Перезапуск контейнерів
8	Перевірка виконання	Логи та статус
9	Завершення	Оновлений застосунок

Демонстрація роботи

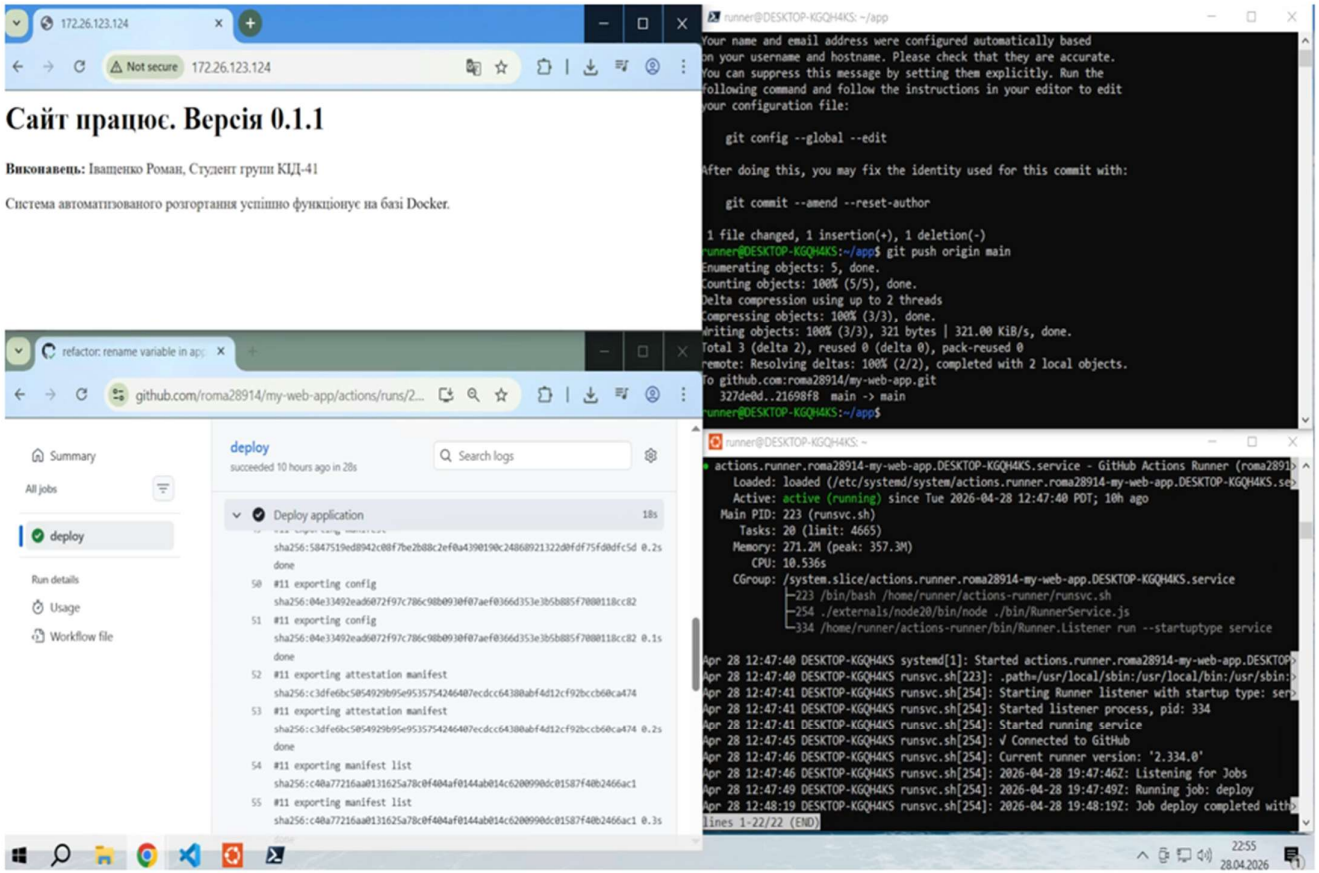


Рис. 2.13 – Відображення оновленого застосунку у браузері (повний CI/CD)



Рис. 2.14 – Сценарій автоматичного оновлення після push

Демонстрація роботи



Рис. 2.16 – Кроки виконання пайплайну

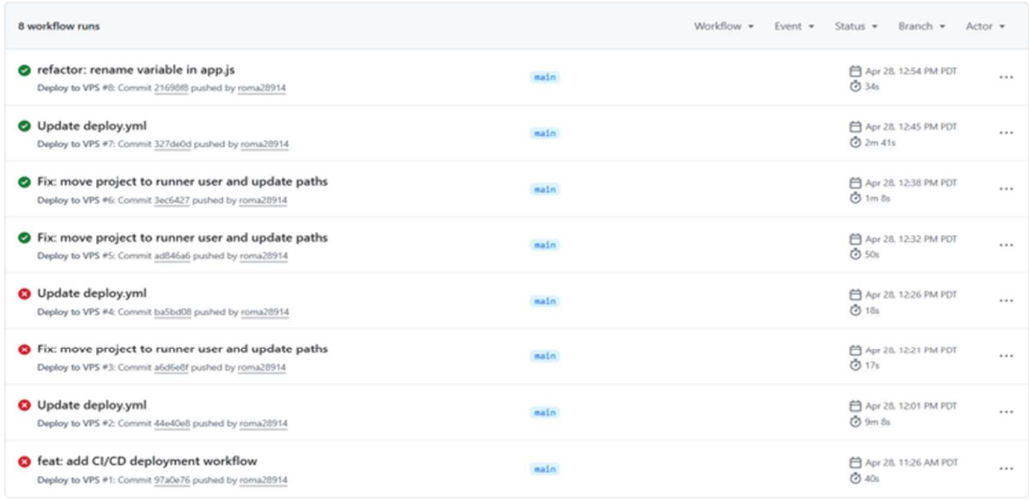


Рис. 2.17 – Логування всіх комітів під час пайплайну

Оптимізація системи

```
root@DESKTOP-KGQH4KS:~/actions-runner# adduser runner
info: Adding user `runner' ...
info: Selecting UID/GID from range 1000 to 59999 ...
info: Adding new group `runner' (1000) ...
info: Adding new user `runner' (1000) with group `runner (1000)' ...
info: Creating home directory `/home/runner' ...
info: Copying files from `/etc/skel' ...
New password:
Retype new password:
passwd: password updated successfully
Changing the user information for runner
Enter the new value, or press ENTER for the default
        Full Name []: runner
        Room Number []:
        Work Phone []:
        Home Phone []:
        Other []:
info: Adding new user `runner' to supplemental / extra groups `users' ...
info: Adding user `runner' to group `users' ...
root@DESKTOP-KGQH4KS:~/actions-runner# id -u
0
root@DESKTOP-KGQH4KS:~/actions-runner# su - runner
Welcome to Ubuntu 24.04.1 LTS (GNU/Linux 6.6.87.2-microsoft-standard-WSL2 x86_64)

 * Documentation:  https://help.ubuntu.com
 * Management:     https://landscape.canonical.com
 * Support:        https://ubuntu.com/pro

 System information as of Tue Apr 28 11:49:07 PDT 2026

  System load:  0.01               Processes:             38
  Usage of /:   0.3% of 1006.85GB  Users logged in:       1
  Memory usage: 12%                IPv4 address for eth0: 172.26.123.124
  Swap usage:   0%

This message is shown once a day. To disable it please create the
/home/runner/.hushlogin file.
runner@DESKTOP-KGQH4KS:~$
```

Рис. 3.1 – Перехід від root до технічного користувача runner



Рис. 3.3 – Метод захищеної взаємодії Self-hosted runner із зовнішнім сервісом



Рис. 3.5 – Метод автоматичного очищення Docker-образів після кожної збірки

Аналіз ефективності

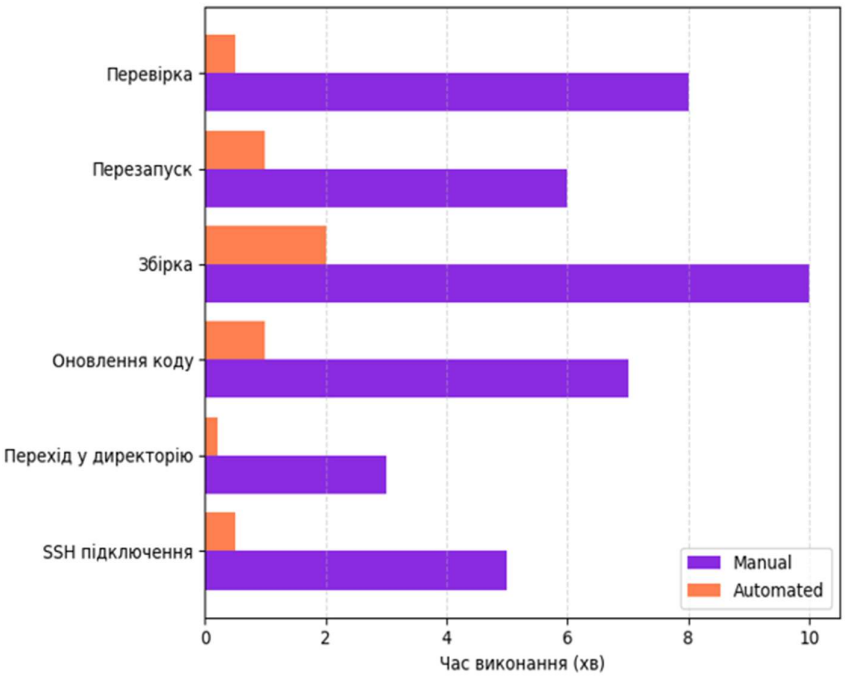


Рис. 3.6 – Гістограма виконання процесів розгортання

Табл. 3.1 – Порівняння ручного та автоматизованого розгортання

Критерій	Ручне розгортання	Автоматизоване розгортання
Час виконання	Хвилини–десятки хвилин	Секунди–хвилини
Ймовірність помилок	Висока (людський фактор)	Мінімальна
Відтворюваність	Нестабільна	Повна
Складність	Висока	Нижча після налаштування
Масштабованість	Обмежена	Висока
Контроль процесу	Ручний	Автоматизований
Залежність від користувача	Висока	Відсутня

Табл. 3.2 – Порівняння GitHub-hosted та Self-hosted runner-ів

Критерій	GitHub-hosted runner	Self-hosted runner
Доступ до ресурсів	Обмежений	Повний
Продуктивність	Обмежена середовищем	Залежить від локальної машини
Безпека	Хмарна ізоляція	Контрольована локально
Мережева модель	Вхідні/зовнішні взаємодії	Pull через HTTPS
Інтеграція з інфраструктурою	Обмежена	Повна
Кастомізація	Обмежена	Повна
Затримки	Можливі	Мінімальні
Підтримка	Не потребує адміністрування	Потребує обслуговування
Масштабування	Автоматичне	Залежить від ресурсів

Перспективи розвитку



Рис. 3.7 – Процес міграції від Docker Compose до Kubernetes-кластера

Висновки

У результаті розробки сформовано цілісну архітектуру автоматизованого розгортання, яка об’єднує репозиторій, виконавче середовище та контейнерну платформу. Процес організовано таким чином, що внесення змін автоматично ініціює виконання пайплайну без ручного втручання, а використання конфігураційних файлів забезпечує відтворюваність і стабільність розгортання. Налаштоване середовище та інтеграція компонентів гарантують безперервність роботи і коректну взаємодію всієї інфраструктури.

Автоматизація ключових етапів, зокрема збірки та оновлення контейнерів без простою, підтвердила ефективність підходу під час тестування. Проведений аналіз показав переваги автоматизованого розгортання над ручним за швидкістю та надійністю, а також визначив напрями подальшого розвитку, включаючи підвищення безпеки, оптимізацію ресурсів і можливість масштабування до кластерних рішень.

Результати впровадження

1. Іващенко Р. М., Лащевська Н. О. Система автоматизованого розгортання та багатофакторної автентифікації контейнеризованих застосунків із використанням Docker Compose та GitHub Actions. «Сучасний стан та перспективи розвитку IoT» : VII Науково-техн. конф., м. Київ, 20 квітня 2026 р. С.294-296.
2. Іващенко Р. М., Лащевська Н. О. Безпечна автоматизація розгортання контейнеризованих застосунків із використанням CI/CD та управління секретами. «Застосування програмного забезпечення в ІКТ» : VII Всеукр. науково-техн. конф., м. Київ, 23 квітня 2026 р. (подано до публікації).