

ПВНЗ «ЄВРОПЕЙСЬКИЙ УНІВЕРСИТЕТ»

КАФЕДРА КОМП'ЮТЕРНИХ НАУК ТА ПРОГРАМНОЇ ІНЖЕНЕРІЇ

СПЕЦІАЛЬНІСТЬ 122 КОМП'ЮТЕРНІ НАУКИ

ОСВІТНЬО-ПРОФЕСІЙНА ПРОГРАМА «КОМП'ЮТЕРНІ НАУКИ»

**ВИПУСКНА КВАЛІФІКАЦІЙНА БАКАЛАВРСЬКА
РОБОТА**

на тему:

**«Впровадження CI/CD для безперервної інтеграції та доставки в
розподілених системах»**

Студент:

Чубук Богдан Андрійович

Науковий Керівник:

Саморай Олег Костянтинович

Допущено до захисту “___” _____ 2025 р.

Завідувач кафедри:

Сушинський Орест Євгенович,
доктор технічних наук, доцент

Декан факультету:

Яровий Роман Олександрович,
кандидат технічних наук

Завідувач навчальної частини:

Козинець Анатолій Олександрович

Київ – 2025

ПВНЗ «ЄВРОПЕЙСЬКИЙ УНІВЕРСИТЕТ»

КАФЕДРА КОМП'ЮТЕРНИХ НАУК ТА ПРОГРАМНОЇ ІНЖЕНЕРІЇ
СПЕЦІАЛЬНІСТЬ 122 КОМП'ЮТЕРНІ НАУКИ
ОСВІТНЬО-ПРОФЕСІЙНА ПРОГРАМА «КОМП'ЮТЕРНІ НАУКИ»

ЗАВДАННЯ

на випускню кваліфікаційну бакалаврську роботу студентів:

Чубуку Богдану Андрійовичу

1. Тема роботи: Впровадження CI/CD для безперервної інтеграції та доставки в розподілених системах

2. Затверджена наказом по університету від “___” _____2025 р. №___

3. Цільова установка на вихідні дані до роботи: Технічні стандарти та рекомендації у сфері DevOps та CI/CD, технічна документація з розробки та впровадження мікросервісної архітектури, офіційна документація системи контейнеризації Docker та платформи оркестрації Kubernetes, методологічні рекомендації з побудови CI/CD-пайплайнів для розподілених систем, дослідження DORA (DevOps Research and Assessment) щодо ефективності DevOps-практик, специфікації інструментів CI/CD (GitLab CI/CD, GitHub Actions, Jenkins, TeamCity, CircleCI, Azure DevOps), документація Spring Boot та Spring Cloud для реалізації мікросервісної архітектури, матеріали проєкту Spring PetClinic Microservices, методики оцінювання ефективності CI/CD-процесів

4. Перелік графічного матеріалу (з точним визначенням обов'язкових креслень): Схеми організації процесів розробки ПЗ (каскадна модель, DevOps-підхід), діаграма порівняння рівнів продуктивності команд DevOps за ключовими метриками, схематичні зображення стратегій розгортання (Blue-Green Deployment, Canary Deployment, Rolling Update), схема CI/CD конвеєра та його етапів, архітектурні діаграми контейнеризації (Docker) та оркестрації (Kubernetes), порівняльні таблиці оцінки CI/CD-інструментів за критеріями

гнучкості, продуктивності та безпеки, архітектурна схема мікросервісної системи Spring PetClinic, скриншоти налаштування GitLab CI/CD, конфігурацій Dockerfile, .gitlab-ci.yml та Kubernetes-маніфестів, ілюстрації розгортання інфраструктури в Kubernetes, скриншоти інтерфейсів Argo CD, Prometheus та Grafana, таблиці порівняння метрик CI/CD-процесу до та після оптимізації.

5. Консультанти по роботі, із зазначенням розділів, що їх стосуються:

Розділ	Науковий керівник	Підпис, дата	
		Завдання видав	Завдання отримав
1. Теоретичні основи впровадження CI/CD	Саморай О.К.	23.10.2024	23.10.2024
2. Аналіз існуючих підходів та інструментів CI/CD	Саморай О.К.	12.02.2025	12.02.2025
3. Практичне впровадження CI/CD у розподіленій системі	Саморай О.К.	12.02.2025	12.02.2025

6. Зміст випускної кваліфікаційної бакалаврської роботи (перелік питань): У процесі виконання випускної кваліфікаційної бакалаврської роботи було досліджено теоретичні основи концепції DevOps та принципів CI/CD, проаналізовано особливості розподілених систем та виклики, пов'язані з їх автоматизацією. Розглянуто вплив технологій контейнеризації та оркестрації на реалізацію CI/CD-процесів. Проведено порівняльний аналіз сучасних систем автоматизації CI/CD за критеріями гнучкості, продуктивності та безпеки з використанням розробленої математичної моделі оцінки ефективності. Досліджено методи інтеграції системи контролю версій, інфраструктури як код та автоматизованого тестування. На основі аналізу обрано оптимальний підхід до CI/CD для розподіленого середовища. Практично реалізовано CI/CD-пайплайн для веб-застосунку Spring PetClinic Microservices, що включає етапи

збірки, тестування, створення артефактів та публікації Docker-образів. Впроваджено GitOps-підхід з використанням Argo CD для розгортання та керування мікросервісами, розроблено скрипт для реалізації стратегії Blue-Green Deployment. Налаштовано систему моніторингу на базі Prometheus та Grafana. Проведено аналіз результатів впровадження CI/CD-процесів, що показав значне покращення ключових метрик продуктивності розробки. Розроблено рекомендації щодо подальшого вдосконалення CI/CD-процесів у розподілених системах.

7. Календарний план виконання випускної кваліфікаційної бакалаврської роботи

№ з/п	Назва етапів випускної кваліфікаційної бакалаврської роботи	Термін виконання етапів випускної кваліфікаційної бакалаврської роботи	
		За планом	Фактично
1.	Обрання теми випускної кваліфікаційної магістерської роботи (ВКБР)	05.08.24 – 13.09.24	05.08.24 – 13.09.24
2.	Опрацювання літератури та інших джерел для написання ВКБР	16.09.24 – 01.11.24	16.09.24 – 01.11.24
3.	Виконання 1 розділу ВКБР	04.11.24 – 28.02.25	04.11.24 – 28.02.25
4.	Виконання 2 розділу ВКБР	03.03.25 – 21.03.25	03.03.25 – 21.03.25
5.	Виконання 3 розділу ВКБР, висновки	24.03.25 – 18.04.25	24.03.25 – 18.04.25
6.	Передзахист та проведення перевірки на плагіат	21.04.24 – 02.05.25	21.04.24 – 02.05.25
7.	Відгук керівника, рецензія	01.05.25 – 12.05.25	01.05.25 – 12.05.25

8. Дата видачі завдання “__” _____ 2025 р.

Керівник _____
(підпис)

Завдання прийняв до виконання _____
(підпис)

АНОТАЦІЯ

Дипломна робота присвячена дослідженню та впровадженню процесів безперервної інтеграції та доставки (CI/CD) у розподілених системах. В умовах стрімкого розвитку інформаційних технологій та поширення мікросервісних архітектур зростає необхідність автоматизації процесів розробки, тестування та розгортання програмного забезпечення. Метою роботи є розробка та впровадження ефективного CI/CD-пайплайну для веб-застосунку в розподіленому середовищі, що забезпечить оптимізацію процесів розробки та підвищення якості програмного продукту.

У процесі дослідження визначено сутність та основні принципи методології DevOps, проаналізовано особливості розподілених систем та виклики їх автоматизації. Особливу увагу приділено впливу технологій контейнеризації та оркестрації на реалізацію CI/CD-процесів. Проведено порівняльний аналіз сучасних систем автоматизації за критеріями гнучкості, продуктивності та безпеки. Розроблено математичну модель оцінки ефективності CI/CD інфраструктури, що дозволила визначити оптимальний підхід для розподіленого середовища.

У рамках дослідження розроблено та впроваджено CI/CD-пайплайн для проєкту Spring PetClinic Microservices з використанням GitLab CI/CD та Kubernetes. Реалізовано GitOps-підхід з використанням Argo CD для автоматизації розгортання. Розроблено спеціалізований скрипт для стратегії Blue-Green Deployment, що забезпечує нульовий простій під час оновлення системи. Налаштовано комплексний моніторинг за допомогою Prometheus та Grafana.

Впровадження розробленого CI/CD-пайплайну дозволило значно покращити ключові метрики процесу розробки: час від коміту до розгортання скоротився на 70%, частота розгортання збільшилася на 200%, кількість відмов

при розгортанні зменшилася на 67%, час відновлення після збоїв скоротився на 83%. Проведена оптимізація пайплайну дозволила зменшити споживання ресурсів CPU на 28% та пам'яті на 26%.

Результати дослідження підкреслюють важливість автоматизації процесів розробки та розгортання у розподілених системах та демонструють ефективні підходи до впровадження практик CI/CD у сучасному IT-середовищі.

Ключові слова: CI/CD, DevOps, GitOps, безперервна інтеграція, безперервна доставка, розподілені системи, мікросервіси, контейнеризація, Kubernetes, автоматизація, Docker, GitLab, Argo CD, Blue-Green Deployment.

SUMMARY

The diploma thesis is dedicated to the research and implementation of Continuous Integration and Continuous Delivery (CI/CD) processes in distributed systems. In the context of rapid information technology development and the spread of microservice architectures, there is an increasing need for automation of software development, testing, and deployment processes. The aim of the work is to develop and implement an effective CI/CD pipeline for a web application in a distributed environment, which will ensure optimization of development processes and improve the quality of the software product.

During the research, the essence and basic principles of DevOps methodology were defined, and the features of distributed systems and challenges of their automation were analyzed. Special attention was paid to the impact of containerization and orchestration technologies on the implementation of CI/CD processes. A comparative analysis of modern automation systems was conducted based on criteria of flexibility, performance, and security. A mathematical model for evaluating the efficiency of CI/CD infrastructure was developed, which allowed for determining the optimal approach for a distributed environment.

Within the research, a CI/CD pipeline was developed and implemented for the Spring PetClinic Microservices project using GitLab CI/CD and Kubernetes. A GitOps approach was implemented using Argo CD for deployment automation. A specialized script for the Blue-Green Deployment strategy was developed, which ensures zero downtime during system updates. Comprehensive monitoring was configured using Prometheus and Grafana.

The implementation of the developed CI/CD pipeline significantly improved key development process metrics: commit-to-deployment time was reduced by 70%, deployment frequency increased by 200%, deployment failure rate decreased by

67%, and recovery time after failures was reduced by 83%. The pipeline optimization reduced CPU resource consumption by 28% and memory usage by 26%.

The research results emphasize the importance of automating development and deployment processes in distributed systems and demonstrate effective approaches to implementing CI/CD practices in the modern IT environment.

Keywords: CI/CD, DevOps, GitOps, continuous integration, continuous delivery, distributed systems, microservices, containerization, Kubernetes, automation, Docker, GitLab, Argo CD, Blue-Green Deployment.

ЗМІСТ

ВСТУП	13
РОДІЛ 1 ТЕОРЕТИЧНІ ОСНОВИ ВПРОВАДЖЕННЯ CI/CD	15
1.1 Сутність та розвиток концепції DevOps	15
1.2 Основні принципи CI/CD і їх значення для безперервної інтеграції та доставки	21
1.3 Особливості розподілених систем та типові виклики автоматизації	31
1.4 Вплив технологій контейнеризації та оркестрації на реалізацію CI/CD	36
РОДІЛ 2 АНАЛІЗ ІСНУЮЧИХ ПІДХОДІВ ТА ІНСТРУМЕНТІВ CI/CD	47
2.1 Огляд сучасних систем автоматизації	47
2.2 Порівняння інструментів за критеріями гнучкості, продуктивності та безпеки	52
2.3 Методи інтеграції системи контролю версій, інфраструктури як код та авто тестування	63
2.4 Вибір оптимального підходу до CI/CD для розподіленого середовища	72
РОДІЛ 3 ПРАКТИЧНЕ ВПРОВАДЖЕННЯ CI/CD У РОЗПОДІЛЕНІЙ СИСТЕМІ	80
3.1 Опис веб-застосунку та його архітектури	80
3.2 Розробка та впровадження CI/CD-пайплайну: побудова, тестування, реліз	84
3.3 Налаштування контейнеризації та керування сервісами в оркестраційному середовищі	97
3.4 Аналіз результатів та оптимізація процесу безперервної доставки	116
ВИСНОВКИ	122
СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ	125
ДОДАТКИ	133

ПЕРЕЛІК УМОВНИХ ПОЗНАЧЕНЬ

API – Application Programming Interface (інтерфейс прикладного програмування) – набір визначень для взаємодії різних компонентів програмного забезпечення.

AWS – Amazon Web Services – платформа хмарних сервісів від компанії Amazon.

Blue-Green Deployment – стратегія розгортання з двома ідентичними середовищами, де одне активне (синє), а інше використовується для нового релізу (зелене).

CALMS – Culture, Automation, Lean, Measurement, Sharing (культура, автоматизація, ощадливість, вимірювання, спільне використання) – фреймворк, що описує ключові принципи DevOps.

Canary Deployment – стратегія поступового розгортання, при якій нова версія спочатку застосовується для невеликої групи користувачів.

CD – Continuous Delivery/Continuous Deployment (безперервна доставка/безперервне розгортання) – практика автоматизованої доставки змін до виробничого середовища.

CI – Continuous Integration (безперервна інтеграція) – практика автоматизованої інтеграції змін коду в спільне сховище.

DAST – Dynamic Application Security Testing (динамічне тестування безпеки додатків) – метод тестування безпеки запущеного додатку.

DevOps – Development + Operations – методологія розробки програмного забезпечення, що поєднує розробку і експлуатацію.

DevSecOps – Development, Security, and Operations – розширення концепції DevOps з інтегрованою безпекою.

DinD – Docker-in-Docker – технологія, що дозволяє запускати Docker-контейнери всередині інших Docker-контейнерів.

Docker – платформа для розробки, доставки та запуску додатків у контейнерах.

DORA – DevOps Research and Assessment – організація, що проводить дослідження ефективності DevOps-практик.

DSL – Domain Specific Language (предметно-орієнтована мова) – мова програмування, спеціалізована для конкретної галузі.

E2E – End-to-End (наскрізне тестування) – метод тестування повного процесу від початку до кінця.

Feature Flags – техніка розробки, що дозволяє вмикати або вимикати функціональність без розгортання нового коду.

GitOps – підхід до автоматизації розгортання програмного забезпечення, який використовує Git як єдине джерело істини.

IaC – Infrastructure as Code (інфраструктура як код) – підхід до управління інфраструктурою за допомогою програмного коду.

JPA – Java Persistence API – специфікація для об'єктно-реляційного відображення в Java.

K8s – Kubernetes – платформа з відкритим кодом для автоматизації розгортання, масштабування та керування контейнерними додатками.

Microservices – архітектурний стиль, при якому додаток складається з невеликих, незалежних сервісів.

MTTR – Mean Time To Recovery (середній час відновлення) – метрика, що вимірює середній час, необхідний для відновлення після збою.

ПЗ – програмне забезпечення.

QoS – Quality of Service (якість обслуговування) – загальна продуктивність мережі або комп'ютерної системи.

REST – Representational State Transfer – архітектурний стиль для розподілених гіперсередовищ.

Rolling Update – стратегія розгортання, при якій екземпляри додатка оновлюються поступово, один за одним.

SaaS – Software as a Service (програмне забезпечення як послуга) – модель поставки програмного забезпечення.

SAST – Static Application Security Testing (статичне тестування безпеки додатків) – метод аналізу вихідного коду для виявлення вразливостей.

Spring Boot – фреймворк для створення автономних, продакшн-готових додатків на Spring.

SRE – Site Reliability Engineering (інженерія надійності сайту) – дисципліна, що поєднує аспекти розробки програмного забезпечення і IT-операцій.

UI – User Interface (користувацький інтерфейс) – середовище взаємодії людини з комп'ютерною системою.

VCS – Version Control System (система контролю версій) – програмне забезпечення для управління змінами в документах, програмах та інших даних.

VM – Virtual Machine (віртуальна машина) – програмне середовище, яке імітує фізичний комп'ютер.

YAML – YAML Ain't Markup Language – людиночитабельний формат серіалізації даних, що часто використовується для конфігураційних файлів.

ВСТУП

Актуальність теми. У сучасному світі інформаційних технологій, де швидкість та надійність розробки програмного забезпечення є критичними, впровадження практик безперервної інтеграції та доставки (CI/CD) стає невід'ємною складовою успішних проєктів. Особливо це стосується розподілених систем, де складність архітектури та географічна розподіленість команд розробки створюють додаткові виклики для забезпечення узгодженості та якості продукту. Впровадження CI/CD у таких умовах дозволяє автоматизувати процеси розробки, тестування та розгортання, мінімізуючи людський фактор та підвищуючи ефективність командної роботи. Це дослідження є актуальним, оскільки спрямоване на розробку та впровадження ефективних CI/CD-процесів у розподілених системах, що відповідає сучасним тенденціям розвитку галузі та потребам індустрії.

Метою дослідження є розробка та впровадження ефективного CI/CD-пайплайну для веб-застосунку в розподіленій системі, що забезпечить автоматизацію процесів інтеграції, тестування та доставки програмного забезпечення.

Для досягнення цієї мети поставлено такі **завдання дослідження**:

- Дослідити теоретичні основи концепції DevOps та принципів CI/CD;
- Проаналізувати особливості розподілених систем та виклики, пов'язані з їх автоматизацією;
- Оцінити сучасні інструменти та підходи для реалізації CI/CD у розподілених середовищах;
- Розробити архітектуру веб-застосунку з урахуванням вимог до автоматизації процесів;
- Реалізувати та налаштувати CI/CD-пайплайн для забезпечення безперервної інтеграції та доставки;

- Впровадити автоматизоване тестування та контроль якості на всіх етапах розробки;
- Проаналізувати результати впровадження та оптимізувати процеси для підвищення ефективності.

Об'єктом дослідження є процеси безперервної інтеграції та доставки програмного забезпечення у розподілених системах.

Предметом дослідження є методи та інструменти впровадження CI/CD для автоматизації безперервної інтеграції та доставки веб-застосунків у розподілених середовищах.

Практичне значення одержаних результатів полягає у створенні CI/CD-пайплайну, який автоматизує рутинні процеси розробки, зменшує кількість помилок, прискорює випуск нових версій програмного забезпечення та покращує співпрацю між командами. Використання сучасних інструментів, таких як контейнеризація та оркестрація, забезпечує гнучкість та масштабованість рішень, що є особливо важливим для розподілених систем. Рекомендації та методи, викладені в дипломній роботі, можуть бути використані в ІТ-компаніях, які прагнуть удосконалити процеси DevOps та перевести існуючі моно- чи багатокомпонентні системи на сучасну мікросервісну й контейнеризовану архітектуру.

Апробація результатів. Результати дослідження були впроваджені та протестовані в рамках внутрішніх проєктів, що підтверджує їх практичну цінність та ефективність у реальних умовах розробки розподілених систем.

РОДІЛ 1

ТЕОРЕТИЧНІ ОСНОВИ ВПРОВАДЖЕННЯ CI/CD

1.1 Сутність та розвиток концепції DevOps

Упродовж останніх десятиліть у галузі інформаційних технологій відбувся суттєвий зсув парадигм від традиційних, здебільшого лінійних, моделей організації розробки програмного забезпечення (ПЗ) до методологій, що базуються на гнучкості, тісній взаємодії всіх учасників процесу та максимальній автоматизації рутинних операцій. Однією з найпомітніших і найвпливовіших таких методологій стала DevOps [1, с. 22]. Її поява й поширення значною мірою зумовлені тим, що сучасні бізнес-умови вимагають безперервного вдосконалення продукту, а також швидкого й надійного випуску нових версій додатків.

Ранні етапи промислової розробки ПЗ базувалися переважно на каскадних (Waterfall) методах, які передбачали сувору послідовність фаз: від аналізу вимог і проєктування — до реалізації та підтримки. Такий підхід зображено на рисунку 1.1.

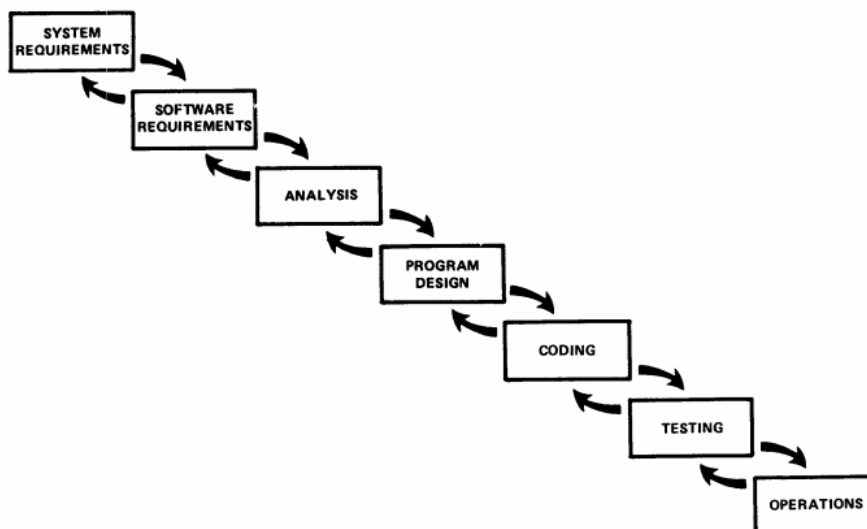


Рисунок 1.1 – Каскадна модель життєвого циклу розроблення програмного забезпечення [2]

Така організація процесу мала низку переваг для великих проєктів, однак водночас характеризувалася довгим періодом між релізами, а також браком швидкої реакції на зміни у вимогах споживачів. Досвід 1980–1990-х років засвідчив, що підхід із жорстким поділом розробки на окремі стадії стає неефективним в умовах, коли ринок ПЗ починає вимагати гнучкості та регулярних оновлень.

У відповідь на ці виклики на початку 2000-х років з'явилися гнучкі методології (Agile), сформульовані у «Маніфесті Agile» (2001 р.) [3]. Agile змістив акцент у бік коротких ітерацій розробки, тісної взаємодії з клієнтом та швидкої адаптації до змін. Проте команда експлуатації (Operations) здебільшого залишалася осторонь, тоді як основна увага зосереджувалася на взаємодії між розробниками, тестувальниками та стейкхолдерами. У результаті між «Dev» і «Ops» формувалася певна «стіна непорозуміння»: розробники прагнули швидкого випуску нового функціоналу, а фахівці з обслуговування систем опиралися стрімким змінам, відстоюючи стабільність продуктивних середовищ.

Саме ця суперечність сприяла появі руху, який у 2008–2009 роках оформився під назвою DevOps (Development + Operations). Початківцями у його поширенні стали практики, що організовували перші конференції DevOps Days у Бельгії (2009 р.) [4] та популяризували ідею об'єднання команд розробки й експлуатації в межах єдиного процесу. За таких умов швидкість впровадження змін, надійність роботи застосунку та зменшення витрат на підтримку інфраструктури стали пріоритетами, досягнути яких можна лише тоді, коли Dev і Ops діють спільно.

У найзагальнішому вигляді DevOps можна визначити як методологію, філософію та культуру, що підкреслює необхідність тісної співпраці між розробниками, тестувальниками, операційним персоналом та іншими учасниками життєвого циклу ПЗ. У книзі «Continuous Delivery: Reliable Software Releases through Build, Test, and Deployment Automation» Девід Фарлі та Джек Хамбл акцентують увагу на необхідності усунення бар'єрів між командами для підвищення ефективності розробки та експлуатації програмних

продуктів [5, с. 279].

Така співпраця реалізується насамперед через:

- автоматизацію більшості рутинних процесів — збирання, тестування, розгортання, моніторингу
- спільну відповідальність за стабільність і якість продукту на всіх етапах, від написання коду до його введення в експлуатацію
- екосистему безперервної інтеграції (CI) і безперервної доставки (CD), що передбачає якнайшвидше виявлення дефектів, а також швидко й безпечно появу нових версій на продуктивних серверах
- відкриту комунікацію та культуру прозорості, коли будь-який збій чи відмова аналізується всією командою з метою запобігання повторенням таких помилок
- організаційні зміни, за яких Dev і Ops, попри традиційний розподіл обов'язків, утворюють єдину функціональну команду або, принаймні, максимально синхронізують свої процеси завдяки інструментам керування конфігураціями, спільним репозиторіям інфраструктурних скриптів тощо.

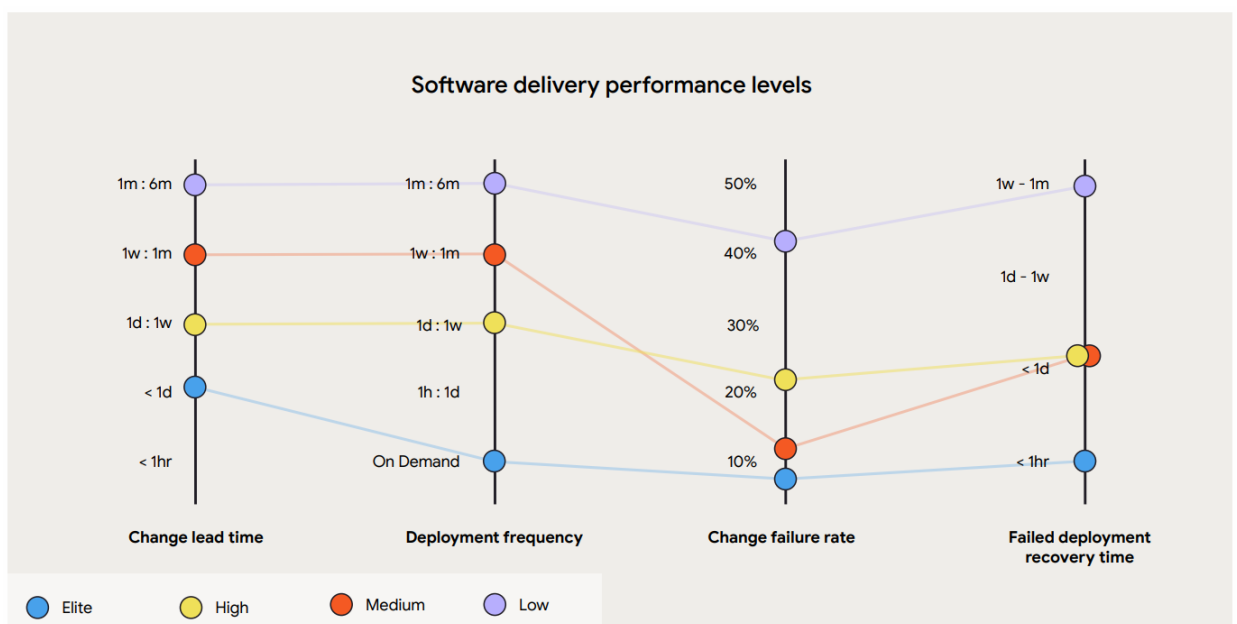
Для структурування цих принципів часто застосовують модель CALMS (Culture, Automation, Lean, Measurement, Sharing). У її межах DevOps постає як культура (Culture), що базується на автоматизації (Automation) та принципах ощадливого виробництва (Lean); у процесі докладно вимірюють (Measurement) усі показники, а результатами діляться (Sharing) між усіма учасниками проєкту.

Активне впровадження хмарних платформ та контейнеризації надало DevOps нового поштовху. Поява Docker (2013 р.) та систем оркестрації (Kubernetes, Docker Swarm, Apache Mesos) забезпечила стандартизацію середовищ виконання та полегшила перенесення застосунків між різними стадіями розробки й продуктивом. Водночас хмарні провайдери (AWS, Azure, Google Cloud) запропонували вбудовані інструменти для налаштування пайплайнів CI/CD (напр., AWS CodePipeline, Azure DevOps, Google Cloud Build), що зробило підхід DevOps доступним навіть для невеликих команд.

Згідно з дослідженням DORA (DevOps Research and Assessment), представленим у «State of DevOps Report 2024», організації, які активно впроваджують DevOps-практики, демонструють значно вищі показники продуктивності та стабільності роботи. Дослідження виявило, що високоефективні команди випереджають конкурентів у 127 разів за швидкістю впровадження змін, мають на 8 разів нижчий рівень відмов у розгортанні та на 182 рази частіше випускають оновлення в продакшн. Такі компанії також відновлюються після збоїв у 2293 рази швидше порівняно з менш ефективними командами [6, с. 15].

На рисунку 1.2 представлено порівняння чотирьох рівнів продуктивності команд DevOps (Elite, High, Medium, Low) за ключовими метриками, що характеризують ефективність процесів розробки та впровадження змін:

- Change Lead Time (час між комітом коду та його виходом у продакшн)
- Deployment Frequency (частота розгортання змін у продакшн)
- Change Failure Rate (відсоток змін, що спричинили збої у продакшні)
- Failed Deployment Recovery Time (час відновлення після невдалого розгортання).



Окрім прискорення випуску оновлень, DevOps-орієнтовані компанії

досягають кращої якості програмного забезпечення, знижують рівень дефектів і забезпечують кращий користувацький досвід. Дослідження Senapathi, Buchan & Osman (2019) [7] також показало, що впровадження DevOps-принципів у новозеландській компанії дозволило збільшити частоту релізів із 30 до 120 на місяць, покращити комунікацію між розробниками та операційними командами та підвищити автономність команд завдяки автоматизації розгортання та моніторингу.

Таким чином, активне використання DevOps сприяє швидкому реагуванню на бізнес-вимоги, зменшенню кількості дефектів у продуктах та підвищенню здатності компанії ефективно працювати під навантаженням у пікові періоди.

На початкових етапах методологію DevOps здебільшого використовували «прогресивні» стартапи або ІТ-гіганти (як-от Netflix, Amazon, Google). Утім, згодом її стали застосовувати у сферах електронної комерції, телекомунікацій, банківського сектора та навіть державних установ, де безвідмовність і безпека сервісів мають визначальне значення. Для успішного впровадження DevOps необхідно враховувати:

1. Зміну організаційної структури. Часто виникає потреба у створенні крос-функціональних команд, які включають у себе розробників, аналітиків, тестувальників та фахівців з експлуатації, що спільно відповідають за результат.
2. Культуру відкритості. Керівництву важливо створити умови, за яких кожен член команди має доступ до інформації про інфраструктуру, логи, метрики тощо. Прозорість сприяє швидшому пошуку «вузьких місць» і демократизує ухвалення технічних рішень.
3. Інструменти та автоматизацію. Багато в чому успіх DevOps визначається якістю інструментарію: системи контролю версій (Git), сервіси безперервної інтеграції (Jenkins, GitLab CI/CD, GitHub Actions), платформи оркестрації (Kubernetes), інструменти інфраструктури як коду (Terraform, Ansible) тощо.

4. Безпеку (DevSecOps). Останнім часом набуває поширення практика «зсуву безпеки вліво» (Shift Left): етапи аналізу безпеки, перевірки контейнерів та сканування на вразливості інтегрують безпосередньо в CI/CD-процес, а не додають уже після того, як основна робота виконана.

Основні вигоди від застосування DevOps-практик можна сформулювати так:

- швидші релізи і зменшення часу на вихід продукту на ринок. регулярні, невеликі оновлення легше тестувати та інтегрувати, а проблеми виявляються швидше, що знижує загальні ризики проєкту
- зменшення операційних витрат і підвищення ефективності. завдяки автоматизації ручних операцій команди можуть приділяти більше уваги творчим завданням, а відповідно — отримують додатковий простір для інновацій
- поліпшення якості коду та стабільності середовища. постійні тести, статичний аналіз, перевірка сумісності пакетів та швидке зворотне повідомлення скорочують імовірність «проблемних» релізів
- задоволеність команди. фахівці відчувають більший контроль над кінцевим продуктом і скорочення бюрократичних бар'єрів, що позитивно впливає на мотивацію.

На практиці це призводить до поліпшення конкурентоспроможності, адже компанія може «випускати» новий функціонал швидше й адаптивніше реагувати на зворотний зв'язок із ринку чи від користувачів.

На сьогодні DevOps вважається однією з ключових ІТ-методологій і дедалі частіше входить до переліку «обов'язкових» підходів у середніх і великих організаціях. Усе частіше DevOps розширюють до концепції DevSecOps, інтегруючи безпеку на всіх етапах життєвого циклу, а також до DataOps (з акцентом на роботу з даними). Щорічні опитування свідчать, що команди, котрі застосовують DevOps-практики, суттєво випереджають за показниками продуктивності й часу відновлення після збоїв тих, хто й досі працює за традиційними схемами.

З урахуванням викликів глобальної цифрової трансформації та розширення масштабів систем (зокрема, перехід на мікросервісні архітектури, збільшення обсягів даних), розвиток DevOps є не тільки технічним, але й організаційним трендом. Це надає можливість уніфікувати робочі процеси, посилювати контроль якості та швидше випускати інноваційні рішення, задовольняючи потреби високо-конкурентного ринку.

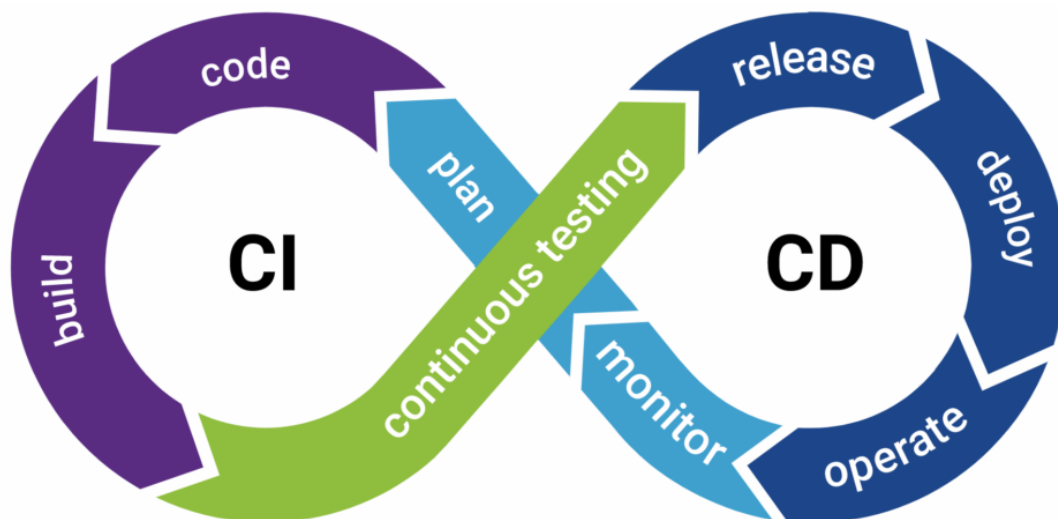
1.2 Основні принципи CI/CD і їх значення для безперервної інтеграції та доставки

Розвиток ІТ-індустрії та зростаючі вимоги до швидкості випуску якісного програмного забезпечення (ПЗ) зумовили популяризацію системи безперервної інтеграції та доставки, що отримала загальноживану назву CI/CD (Continuous Integration / Continuous Delivery або Continuous Deployment). Цей підхід тісно переплетений із методологією DevOps і виступає одним із ключових інструментів досягнення її мети: швидкого, надійного та передбачуваного розгортання нових версій застосунків у виробничому середовищі. У цьому підрозділі розглянуто основні принципи CI/CD, а також їхнє значення для організації безперервної інтеграції та доставки, особливо в умовах розподілених систем.

Термін Continuous Integration (CI) у буквальному перекладі означає «безперервну інтеграцію». Його суть полягає в тому, що розробники регулярно (бажано кілька разів на день) зливають зміни до спільного репозиторію коду, після чого автоматично запускається процес складання та первинного тестування. Головною метою CI є якомога раніше виявлення конфліктів або дефектів, що виникають під час об'єднання змін від різних учасників команди. Якщо збірка або тести завершуються з помилками, відповідальним за зміни миттєво надходить зворотний зв'язок, що дає змогу виправити проблему, доки вона не поширилася на інші модулі [8].

Наступною логічною ланкою є Continuous Delivery (CD) — безперервна доставка, згідно з якою продукт у будь-який момент часу має бути готовий до випуску у виробниче середовище. Для цього з автоматизованого процесу CI (збирання та тестування) «вирастає» ширший цикл, що включає додаткові види тестів (інтеграційні, навантажувальні, безпекові), формування релізних артефактів (наприклад, контейнерних образів) та підготовку до деплоюменту. При цьому вирішальний крок — власне розгортання — може залишатися під контролем людини: команда, переконавшись, що все пройшло перевірку, «натискає кнопку», щоб випустити оновлення на продуктив.

Ще глибшим рівнем автоматизації є Continuous Deployment, коли розгортання в середовищі користувача відбувається автоматично, за умови успішного проходження всіх тестів. Такий підхід використовується тоді, коли рівень упевненості у процесах тестування та відкату (roll-back) достатньо високий, аби виключити потребу ручного затвердження. Згідно з опитуваннями, Continuous Deployment найчастіше застосовується у компаніях, що активно працюють із веб-сервісами чи мікросервісною архітектурою, де невеликі оновлення виходять по кілька разів на день.



Для кращого розуміння процесу CI/CD на рисунку 1.3 представлено основні етапи CI/CD конвеєра:

Крім того, важливо розуміти різницю між традиційною доставкою

програмного забезпечення, CI, CD та повним CI/CD конвеєром. Нижче на таблиці 1.1. наведено порівняння цих підходів.

Таблиця 1.1 – Порівняння підходів CI, CD та повним CI/CD конвеєром [9]

Підхід	Традиційна доставка ПЗ	Continuous Integration	Continuous Delivery	CI/CD конвеєр
Мета	загальний процес доставки програмного забезпечення для кінцевих користувачів або виробничого середовища	процес автоматичної інтеграції змін коду в спільне сховище та запуск автоматизованих тестів	процес автоматичного розгортання коду до виробничого середовища після проходження всіх тестів	набір практик та інструментів, які автоматизують весь процес доставки програмного забезпечення, включаючи інтеграцію, тестування та розгортання
Автоматизація	загальний процес може включати ручні кроки та не мати повної автоматизації	зосереджується на автоматизації інтеграції та тестування змін коду	автоматизує процес розгортання, але може зажадати схвалення людини, перш ніж випустити виробництво	автоматизує весь конвеєр доставки програмного забезпечення, від інтеграції коду до тестування та розгортання
Людське втручання	вимагає більше втручання людини, особливо на фазі розгортання	автоматизована інтеграція та тестування коду з мінімальним втручанням людини	може передбачати схвалення людини перед розгортанням виробництва в CD	можуть включати в себе ручні контрольні шлюзи, але цей процес в значній мірі автоматизований

Основні елементи та етапи CI/CD-процесу:

1. Система контролю версій (VCS). Кожен розробник вносить зміни до коду за допомогою гілок (branches). Сучасним стандартом де-факто є Git, що надає гнучкий і надійний інструментарій для колективної роботи. Саме в репозиторії зберігаються всі зміни та ініціюються подальші автоматичні дії.
2. Автоматизоване збирання (build). Після того, як зміни потрапляють до основної чи інтеграційної гілки, система безперервної інтеграції

(Jenkins, GitLab CI/CD, GitHub Actions, CircleCI тощо) виконує процедуру збирання. Залежно від мови та середовища розробки це можуть бути компіляція, збирання пакетів, підготовка контейнерних образів тощо.

3. Автоматизоване тестування. В ідеалі, на етапі CI має запускатися якомога більше тестів, зокрема:

- юніт-тести (unit tests): перевіряють окремі функції чи класи
- інтеграційні тести (integration tests): забезпечують правильну взаємодію між декількома модулями
- функціональні тести (functional tests): перевіряють відповідність функціоналу вимогам
- статичний аналіз коду (static code analysis): виявлення потенційних вразливостей, стилістичних відхилень, недоліків продуктивності.

Якщо хоча б один тест дає негативний результат, процес вважається «зламаним», а розробники отримують повідомлення про необхідність усунення помилки.

4. Формування релізних артефактів і середовище staging. У разі успішного збирання й тестування система формує остаточний продукт (артефакт): виконуваний файл, архів або контейнерний образ (наприклад, Docker image). Цей артефакт потім можуть розгорнути в проміжному середовищі (staging) чи «середовищі, максимально наближеному до бойового», де проводять додаткові перевірки, такі як інтеграційні тести між різними мікросервісами, навантажувальні тести та перевірки безпеки.

5. Розгортання (deployment) у виробничому середовищі. Залежно від рівня автоматизації (Continuous Delivery чи Continuous Deployment) новий реліз може:

- вийти автоматично, якщо вся послідовність перевірок пройдена

успішно

- очікувати ручного підтвердження, коли відповідальна особа переглядає результати і дає «зелене світло» на перенесення змін у продакшн.

6. Моніторинг і зворотний зв'язок. Після розгортання сервіс або застосунок працюють у бойовому режимі, тому важливо мати належні засоби моніторингу й логування. Метрики на кшталт відсотка помилок, часу відповіді, завантаженості процесора тощо свідчать про якість нового релізу. У разі погіршення показників команда або система автоматично ініціює відклик релізу (rollback).

Значення CI/CD для безперервної інтеграції та доставки:

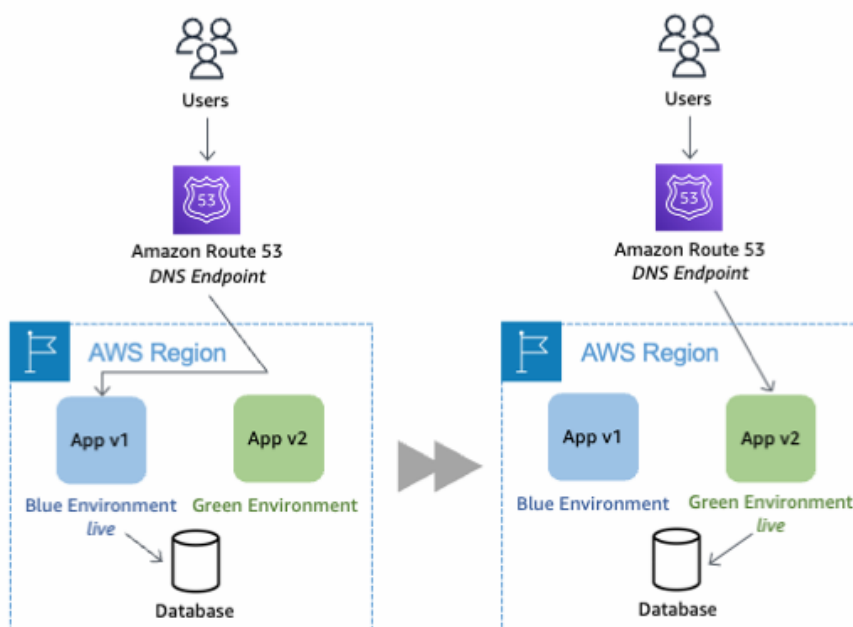
1. Раннє виявлення помилок і підвищення якості. Завдяки CI/CD дефекти коду виявляються на рівні коротких, локальних змін, а не накопичуються місяцями. Це знижує вартість виправлень: відомо, що чим раніше знайдено помилку, тим меншими зусиллями вона виправляється.
2. Прискорення циклу розробки. Автоматизація ключових етапів (збирання, тестування, реліз) дає змогу командам частіше випускати оновлення, підтримуючи при цьому високу стабільність. Це особливо критично для бізнесу, де швидкість виходу нового функціоналу часто визначає конкурентоспроможність продукту.
3. Спрощення командної взаємодії. Розробники, тестувальники й DevOps-фахівці мають єдиний «прозорий» процес, у якому зрозуміло, на якому етапі перебуває проєкт, що з ним відбувається, та які тести «падають» у разі збоїв. Замість взаємних претензій щодо якості збірки, вся команда залучена до покращення процесу.
4. Контрольованість випуску оновлень. Якщо продукт містить декілька компонентів або мікросервісів, CI/CD дає змогу розгортати оновлення вибірково, поетапно чи навіть використовувати техніки «синьо-зеленого» (blue-green) або «канаркового» (canary) деплою, деплойменту,

мінімізуючи ризики для кінцевих користувачів [10 с. 166].

5. Легкість масштабування та адаптації. Для розподілених систем, що можуть складатися із сотень сервісів, налагоджений процес CI/CD є чи не єдиним способом утримати продуктивне середовище в стабільному стані й уникати конфліктів між версіями. Мікросервісна архітектура у поєднанні з CI/CD дає гнучкі можливості швидкого розгортання в хмарі, масштабування під навантаження та повернення до попередньої версії у разі потреби.

Для підвищення надійності та зменшення ризиків при оновленні програмних продуктів у складних розподілених системах застосовують низку сучасних стратегій:

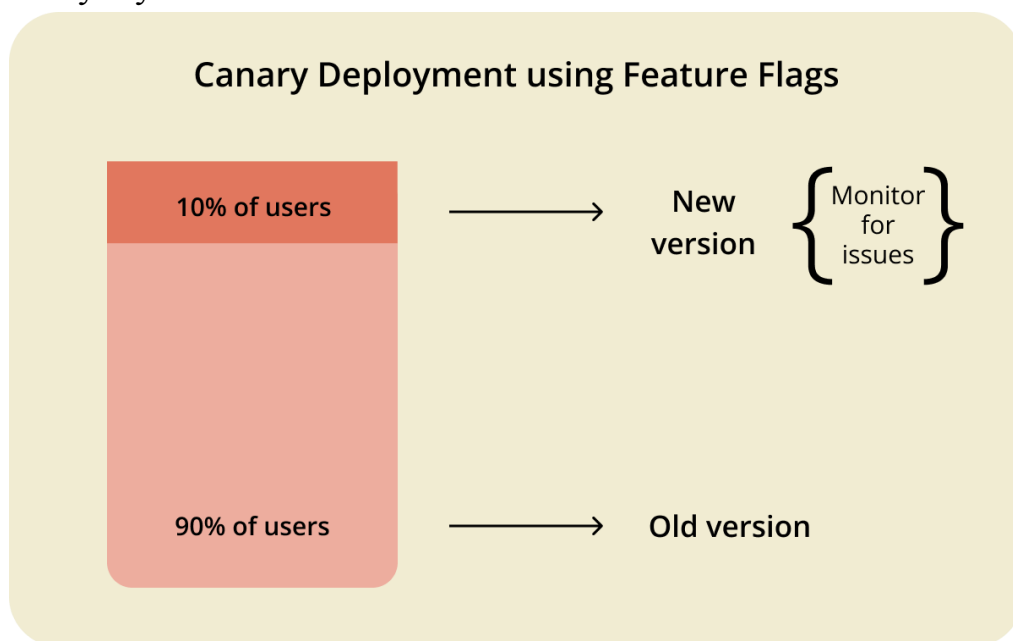
- Blue-green deployment – це метод при якому створюються два паралельні середовища – «синє» (активне) та «зелене» (резервне). Нова версія застосунку спочатку розгортається у «зеленому» середовищі, після чого відбувається перемикавання користувацького трафіку з «синього» на «зелене». У разі виявлення недоліків можливий швидкий відкат до попередньої версії [11]. На рисунку 1.4 наведено зображення blue-green deployment:



- Canary deployment – це метод розгортання програмного забезпечення,

при якому нова функція або версія випускається для невеликої підмножини користувачів у процесі виробництва перед випуском для більшої підмножини або всіх користувачів. Це також іноді називають поетапним впровадженням або інкрементним випуском. Крім зниження ризику випадкового випуску коду з помилками, він надає можливість протестувати нову версію функції в робочому середовищі і подивитися, як відреагують користувачі. При цьому для більшої гнучкості та керованості процесом часто використовують функціональні прапорці (Feature Flags) (див. Рис. 1.5).

Вони дозволяють динамічно вмикати або вимикати нові функції для певних груп користувачів без зміни коду або повторного розгортання застосунку.



Завдяки Canary deployment у поєднанні з функціональними прапорцями можна поступово розширювати аудиторію нової версії, оперативно виявляти потенційні дефекти та, у разі потреби, швидко їх нейтралізувати, мінімізуючи вплив на загальну роботу системи.

- Rolling update – це підхід, за якого оновлення здійснюється поступово: окремі вузли або контейнери замінюються новою версією поетапно, що дозволяє підтримувати безперервну роботу системи та знижувати ризик масштабного збою [13].

Наприклад розглянемо як Rolling update реалізований в Kubernetes. Як показано на рисунку 1.6, під час оновлення Kubernetes створює нові поди на основі оновленого контейнера, а старі екземпляри видаляються тільки після того, як нові поди успішно проходять перевірку готовності (readiness probe). Це дозволяє рівномірно розподілити навантаження на систему та уникнути збоїв у роботі

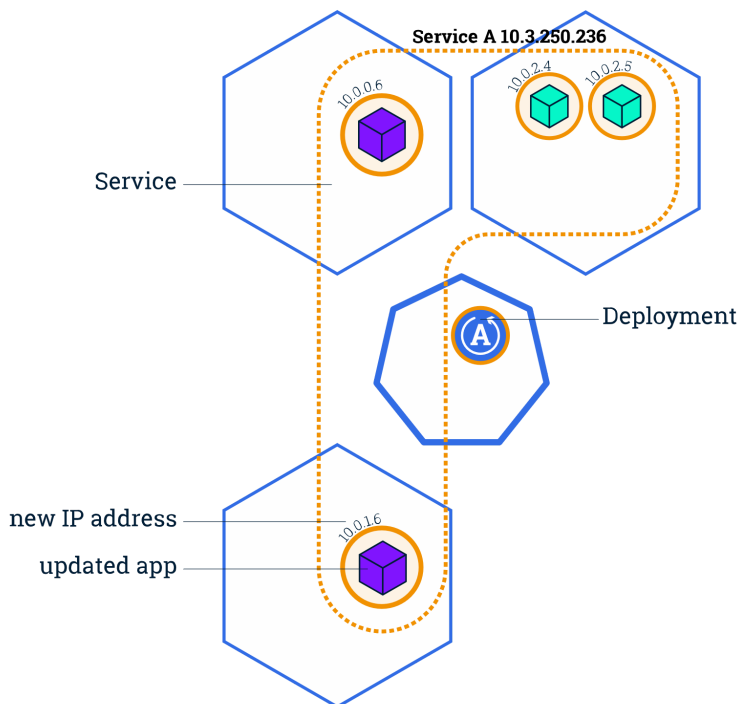


Рисунок 1.6 – Схема оновлення подів під час Rolling Update у Kubernetes [13]

сервісів.

На схемі видно, що Service A (IP: 10.3.250.236) спочатку спрямовує трафік на старі поди (позначені пурпурним кольором). У процесі оновлення розгортання (Deployment) додає нові поди (бірюзового кольору), а старі поступово видаляються. При цьому нові поди отримують нові IP-адреси.

Один із ключових аспектів Rolling update – це можливість швидкого відкату (rollback) у разі виявлення проблем. Kubernetes дозволяє повернутися до попередньої стабільної версії без зупинки сервісу.

В таблиці 1.2 наведено порівняльну характеристику стратегій

розгортання на основі статті від команди Unleash [14]:

Таблиця 1.2 – Порівняльна характеристика стратегій розгортання

Параметр	Blue-Green	Canary	Rolling Update
Швидкість розгортання	швидке перемикання між двома ідентичними середовищами забезпечує мінімальний час простою	поступове впровадження нової версії дозволяє виявити проблеми на ранніх етапах, але повне розгортання займає більше часу	інкрементне оновлення забезпечує безперервну роботу, але процес може бути повільнішим через поступовий характер
Ризики	можливість швидкого повернення до попередньої версії знижує ризики, але вимагає подвійних ресурсів	обмеження впливу нової версії на невелику групу користувачів дозволяє виявити та виправити проблеми до повного розгортання, знижуючи ризики	поступове оновлення зменшує ризик одночасного збою всієї системи, але може ускладнити управління версіями та відстеження помилок
Можливість відкату	швидке перемикання назад на попереднє середовище забезпечує миттєвий відкат у разі проблем з новою версією	легкий відкат для невеликої групи користувачів мінімізує вплив на більшість користувачів у разі виявлення проблем	відкат може бути складнішим, оскільки потребує поетапного повернення кожного інстансу до попередньої версії

Таким чином, застосування сучасних стратегій розгортання у CI/CD дозволяє забезпечити високий рівень автоматизації оновлень, оперативне виявлення дефектів і швидкий відкат у разі виникнення проблем, що особливо важливо для розподілених систем із складною архітектурою.

Для CI/CD існують такі поширені інструменти:

- Jenkins. Один із найстаріших і найпоширеніших інструментів із відкритим вихідним кодом для організації процесів збирання та тестування. Завдяки численним плагінам Jenkins інтегрується з багатьма мовами, системами контролю версій та платформами.

- GitLab CI/CD. Пропонує вбудовані конвеєри (pipelines) прямо в репозиторії проекту, тісно інтегруючи процес релізу із системою керування вихідним кодом і засобами код-рев'ю.
- GitHub Actions. Схожий за філософією на GitLab CI, але розгортається в екосистемі GitHub, що надає гнучкі «workflow» (робочі процеси) та швидку інтеграцію з рештою продуктів GitHub.
- CircleCI, Travis CI, Bamboo, TeamCity та інші сервіси, що налаштовуються під конкретні потреби команди й можуть працювати як у хмарі, так і «on-premises».

Усі вони впроваджують подібну концепцію: при надходженні змін у репозиторій запускати визначені кроки (build, test, package, deploy), надавати розробникам детальний звіт про успішність кожного етапу та формувати релізний артефакт.

DevOps як культура й методологія підкреслює важливість безперервності (continuous everything): інтеграції, тестування, доставлення, моніторингу та поліпшення. CI/CD-процес фактично стає «технічним стрижнем» DevOps, оскільки реалізує на практиці кілька базових принципів:

1. Автоматизація. Процеси, які раніше виконувалися вручну, стають запрограмованими сценаріями. Це істотно знижує ризик «людського чинника» та пришвидшує випуск оновлень.
2. Швидкий зворотний зв'язок. При виявленні дефектів усі причетні до коду отримують миттєве сповіщення. Це стимулює частіший обмін інформацією й запобігає виникненню великих «технічних боргів».
3. Спільна відповідальність. Тестування й випуск продукту — це не справа тільки відділу контролю якості чи операційної команди; кожен розробник робить свій внесок і відповідає за готовність коду до релізу.

З погляду DevOps, CI/CD — один із фундаментальних «конвеєрів», що забезпечує невинний потік змін, від етапу написання коду до розгортання в середовищі реальних користувачів.

Із переходом індустрії до мікросервісної та хмарної архітектури потреба у

CI/CD набуває критичного значення. Коли застосунок складається з десятків або й сотень мікросервісів, кожен із яких має власний цикл оновлення, саме CI/CD забезпечує:

- Уніфіковані пайплайни. Кожен сервіс може мати окремий або спільний пайплайн, у якому прописано, як збирати, тестувати і розгортати оновлення.
- Контейнеризацію та оркестрацію. Інструменти CI/CD тісно інтегруються з Docker, Kubernetes, Helm та іншими технологіями, щоб автоматично розміщувати нові версії в кластері (on-premises чи в хмарі) й масштабувати сервіси під навантаження.
- Механізми безпечного оновлення. Використання «блакитно-зеленого» (blue-green) або «канаркового» (canary) деплойменту дає змогу мінімізувати простій і ризики невдалого розгортання. Зокрема, малу частку трафіку перенаправляють на нову версію сервісу, а якщо все працює коректно, поступово переводять усіх користувачів.

У такій системі кожен сервіс може оновлюватися незалежно, не блокуючи роботу інших, а головне — зміни стають прозорими для всієї команди. Це сприяє своєчасному виявленню конфліктних версій або взаємних несумісностей.

1.3 Особливості розподілених систем та типові виклики автоматизації

Розвиток інформаційних технологій призвів до того, що велика кількість програмних застосунків набуває розподіленої архітектури: сервіси чи окремі компоненти працюють на різних вузлах (фізичних або віртуальних серверах, контейнерах, хмарних платформах) і взаємодіють між собою через мережу. Такий підхід дає змогу покращувати масштабованість, відмовостійкість і гнучкість системи, але водночас створює додаткові складнощі для автоматизації, тестування та впровадження процесів CI/CD. У цьому підрозділі

розглянуто ключові характеристики розподілених систем, а також проблеми і ризики, пов'язані з організацією безперервної інтеграції та доставки в подібних середовищах.

Термін «розподілена система» охоплює широкий спектр архітектурних моделей, від простого варіанта «клієнт—сервер» до складних кластерів мікросервісів, які взаємодіють між собою й можуть бути розгорнуті на тисячах вузлів. Серед основних ознак, що характеризують розподілені системи, варто виділити:

1. Фізичну й логічну розподіленість компонентів. Окремі частини системи (сервери додатків, бази даних, кеші, сервіси мікросервісної архітектури) можуть розташовуватися в різних дата-центрах або хмарних середовищах і спілкуватися через мережеві протоколи (HTTP, gRPC, message queues тощо).
2. Можливість горизонтального масштабування. Замість нарощування потужності одного сервера, розподілена архітектура дозволяє додавати нові вузли, збільшуючи продуктивність системи загалом [15 с. 441].
3. Децентралізоване зберігання та оброблення даних. Кожен вузол може мати власну базу даних, кеш або sharded-репліку. Це, з одного боку, дає переваги у відмовостійкості, з іншого — створює виклик для узгодженості даних (data consistency).
4. Гетерогенність і незалежність компонентів. У межах однієї системи можуть функціонувати сервіси, написані різними мовами програмування та з різними середовищами виконання, що додатково ускладнює інфраструктуру та інтеграцію [15 с. 3–6].

Із розповсюдженням хмарних платформ (AWS, Azure, Google Cloud) та контейнерних технологій (Docker, Kubernetes) розподілені системи стали ще доступнішими. Тепер команди можуть без великих капітальних витрат розміщувати свої мікросервіси у хмарі, швидко масштабуватися та використовувати вбудовані сервіси оркестрації. Такі рішення суттєво полегшують упровадження DevOps-практик, але водночас створюють нові

виклики в налаштуванні безперервної інтеграції та доставки.

Типові виклики автоматизації у розподілених середовищах:

1. Складність взаємодії компонентів

В умовах, коли застосунок поділено на безліч мікросервісів чи сервісів, кожен із яких має власний цикл оновлення, зростає ризик несумісності версій. Навіть невеликі зміни в інтерфейсі одного сервісу можуть спричинити збій в іншому, якщо вони не були синхронізовані. Це означає, що:

- необхідне централізоване управління версіями арі та контрактними тестами (contract testing), які перевіряють взаємодію між сервісами
- стає критично важливим застосування сі/cd-процесів, що дають змогу автоматизовано будувати та тестувати кожен сервіс і перевіряти його сумісність із рештою системи.

2. Оркестрація та динамічність середовища

Застосунок може розгортатися у кластері Kubernetes, де кожен сервіс представлений одним або кількома контейнерами (pod) і де відбуваються динамічні зміни конфігурації залежно від навантаження чи стану вузлів. З одного боку, це спрощує масштабування, з іншого — ускладнює процес тестування, оскільки топологія мережових з'єднань і набір працюючих вузлів можуть змінюватися практично у режимі реального часу:

- потрібно мати автоматизовані скрипти чи «хелм-чарти» (helm charts) для інсталяції й оновлення кожного мікросервісу
- важливим стає спостереження (observability): без ретельного моніторингу, логування та трасування (distributed tracing) діагностувати збої в оркестрованій системі надзвичайно складно.

3. Забезпечення узгодженості даних і транзакційності

У розподілених системах часто відсутня єдина база даних, натомість інформація може бути розподілена по кількох сервісах. Виникає потреба у забезпеченні узгодженості даних (consistency) та реалізації розподілених транзакцій. Особливо складно це робити, коли різні вузли

працюють за принципом *eventual consistency*, тобто коли кожен сервіс оновлюється з певною затримкою. Для автоматизації CI/CD це означає:

- потрібні комплексні інтеграційні тести, що моделюють взаємодію між сервісами та перевіряють цілісність даних у реальних сценаріях
- нерідко доводиться налаштовувати середовище тестування таким самим чином, як і бойове — з кількома екземплярами баз даних, сервісами кешування тощо.

4. Безпека та політики доступу

У розподілених середовищах кожен сервіс може мати власний механізм аутентифікації (наприклад, OAuth2, JWT-токени) чи шифрування з'єднань (TLS). При цьому автоматизація оновлень ускладнюється, якщо сервіси обмінюються захищеними запитами й потребують прив'язаних до середовища сертифікатів або ключів. Тому DevOps-команди дедалі частіше впроваджують концепцію DevSecOps, коли ще на етапі CI/CD виконується:

- статичний аналіз безпеки коду (SAST).
- перевірка контейнерів на відомі вразливості (наприклад, за допомогою контейнерних сканерів).
- налаштування секретів та сертифікатів через системи на кшталт HashiCorp Vault.

5. Моніторинг і логування в багатокомпонентному середовищі

У розподілених системах, особливо в мікросервісній архітектурі, логування та моніторинг перетворюються на критичну потребу. Щоби точно з'ясувати, де і чому стався збій, потрібно:

- збирати логи з усіх сервісів у централізовану систему (наприклад, *elasticsearch*, *splunk*, *graylog*) і забезпечувати їхню уніфіковану обробку та кореляцію
- використовувати механізми розподіленого трасування (*distributed tracing*) на кшталт *opentracing* чи *jaeger*, що дає змогу простежити «ланцюжок викликів» між мікросервісами і виявляти «вузькі місця»

- постійно аналізувати метрики (CPU, RAM, час відповіді, кількість HTTP-помилки) за допомогою Prometheus, Grafana чи аналогічних рішень.

Без цих інструментів усі спроби автоматизації можуть наштовхуватися на відсутність зрозумілої картини загального стану системи.

6. Вплив географічної розподіленості команд

Часто розподілені системи розробляють команди, які теж розташовані в різних куточках світу. Це додатковий виклик для координації зусиль, синхронізації графіків релізів та узгодження стандартів коду. Тут важливу роль відіграє:

- наявність спільного репозиторію та стандартів ci/cd, які однаково застосовуються у всіх офісах чи часових поясах
- чітке документування пайплайнів, інфраструктури та політик доступу, щоб кожен розробник міг коректно інтегруватися в загальний процес.

Шляхи подолання типових викликів автоматизації:

1. Модульні та контрактні тести. На додаток до звичайних інтеграційних тестів, застосовують контрактні тести (consumer-driven contracts), що дозволяють виявити, чи не порушений «контракт» API між сервісами.
2. Стандартизовані пайплайни. Більшість масштабних команд упроваджують єдині правила CI/CD, створюють «шаблони» пайплайнів (pipeline templates), які служби можуть успадковувати й налаштовувати під свої потреби, зберігаючи при цьому уніфікованість процесу.
3. Використання хмарних сервісів. Якщо організація переїжджає в хмару, вона може скористатися «рідними» сервісами для автоматизації (AWS CodePipeline, Azure DevOps, Google Cloud Build), сервісами моніторингу (CloudWatch, Azure Monitor, Stackdriver) та зберігання логів. Це мінімізує витрати на розгортання власної інфраструктури моніторингу.
4. Середовища «стейджинг» та «пре-продакшн», імітація реальних умов. Дуже бажано мати майже ідентичне середовище до бойового для інтеграційного тестування й перевірки оновлень. Тоді опис конфігурації

(Infrastructure as Code) дає змогу в автоматичному режимі відтворювати робочі сценарії.

5. Застосування сервісної сітки (service mesh). Сервісна сітка (наприклад, Istio, Linkerd) може автоматизувати мережеві політики, балансування, трасування й збір метрик на рівні мережі, розвантажуючи цим код сервісів. Це неабияк полегшує керування складною взаємодією в розподіленій системі.

1.4 Вплив технологій контейнеризації та оркестрації на реалізацію CI/CD

У сучасних підходах до розробки й постачання програмного забезпечення (ПЗ) контейнеризація стала одним із ключових факторів, що дає змогу значно спростити процеси розгортання, масштабування й оновлення застосунків. Поєднання контейнерних технологій (насамперед Docker) із системами оркестрації (Kubernetes, Docker Swarm, Apache Mesos тощо) докорінно змінює парадигму безперервної інтеграції та доставки (CI/CD), адже сприяє побудові гнучких, уніфікованих та легких у керуванні конвеєрів розгортання навіть у великих розподілених середовищах. У цьому підрозділі розглядаються засадничі принципи контейнеризації та оркестрації, а також проаналізовано, як саме вони впливають на реалізацію процесів CI/CD.

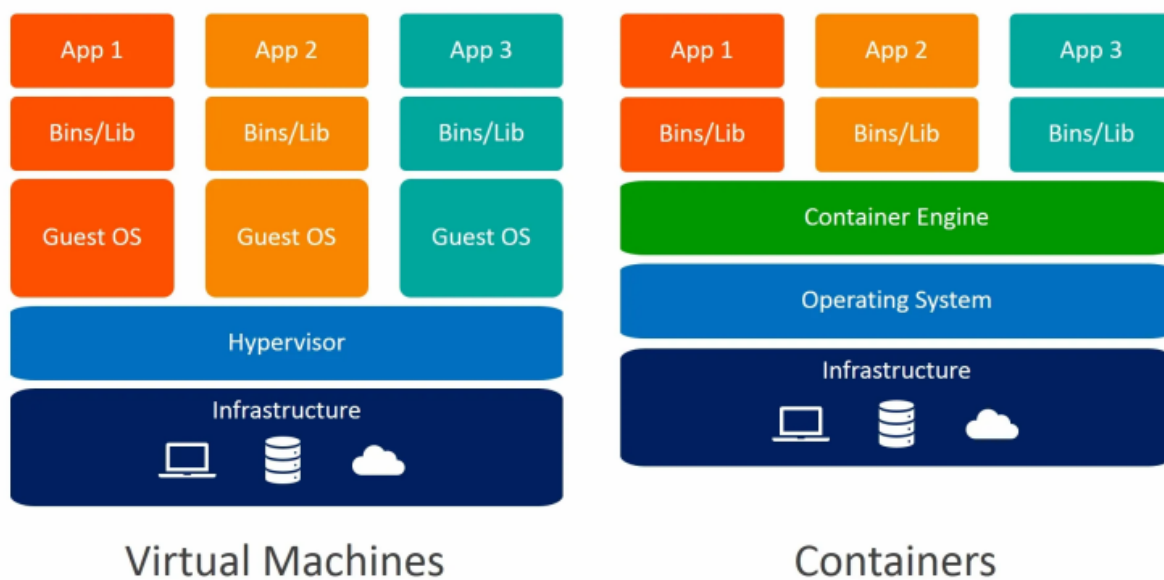
Контейнеризація — це метод упакування ПЗ разом із його залежностями (бібліотеки, конфігураційні файли, середовище виконання) у стандартизований модуль (контейнер), що може бути запущений у будь-якому середовищі, де є відповідний контейнерний рушій (Docker Engine, containerd тощо). Цей підхід зазвичай передбачає такі переваги:

1. Уніфіковане середовище виконання. Оскільки контейнер містить усе необхідне для запуску, команда розробки та команда експлуатації уникають конфліктів версій бібліотек або налаштувань ОС. Принцип «якщо працює на моїй машині — має працювати й на сервері» стає

реальністю, а не жартом.

2. Швидке масштабування. У разі зростання навантаження можна просто запустити додаткові контейнери із тим самим образом і розподілити навантаження за допомогою балансувальників або систем оркестрації.
3. Легкість оновлень та відкату. Контейнерний образ можна легко замінити на нову версію в будь-якій стадії конвеєра CI/CD. Якщо у новій версії виявлено критичну помилку, відкат до попереднього робочого образу відбувається швидко, мінімізуючи час простою.
4. Портативність. Контейнери можуть працювати як у локальному середовищі розробника, так і в хмарі або на виділеному сервері в дата-центрі. Завдяки цьому команда DevOps має широкий спектр вибору інфраструктури та способів розгортання.

Важливо зазначити, що контейнеризація має значні переваги перед традиційною віртуалізацією, що пояснюється принципами її роботи, які представлені на рисунку 1.7.



Ключова відмінність між віртуальними машинами (VM) та контейнерами полягає в їхній архітектурі. Наприклад у віртуальних машинах:

- міститься власна гостьова операційна система (Guest OS), що створює значне навантаження на ресурси
- використовується гіпервізор (hypervisor), який керує розподілом

ресурсів між віртуальними машинами

- запуск додатків у різних середовищах вимагає окремих віртуальних машин, що збільшує час розгортання та використання ресурсів

в свою чергу контейнери мають інший підхід:

- контейнери використовують єдину операційну систему хоста (Host OS), що дозволяє зменшити накладні витрати
- контейнерний рушій (наприклад, `docker engine`) забезпечує ізоляцію між контейнерами без потреби запускати окрему ос для кожного екземпляра
- контейнери легші, швидші у запуску та ефективніші з точки зору використання ресурсів, ніж віртуальні машини.

Завдяки своїй архітектурі, контейнеризація має ряд переваг над традиційною віртуалізацією:

- менше споживання ресурсів – контейнерам не потрібні окремі гостьові ос, що зменшує витрати на пам'ять та процесорні ресурси
- швидке розгортання – запуск контейнера займає секунди на відміну від повного завантаження віртуальної машини
- масштабованість – контейнери легко розгортаються та масштабуються за допомогою оркестраційних систем, таких як `Kubernetes`
- кросплатформеність – завдяки використанню образів контейнери можна запускати на будь-якій інфраструктурі, що забезпечує гнучкість у розробці та тестуванні.

З огляду на ці переваги, контейнеризація активно використовується в сучасних DevOps-підходах, зокрема для безперервної інтеграції та доставки (CI/CD).

Основним рушієм контейнеризації став `Docker`, випущений у 2013 році. `Docker` – це платформа для контейнеризації, яка дозволяє створювати, розгортати та запускати застосунки в ізольованих контейнерах.

Основними елементами `Docker` є:

- `Dockerfile`: Текстовий файл, що містить послідовність інструкцій

для створення контейнерного образу (image)

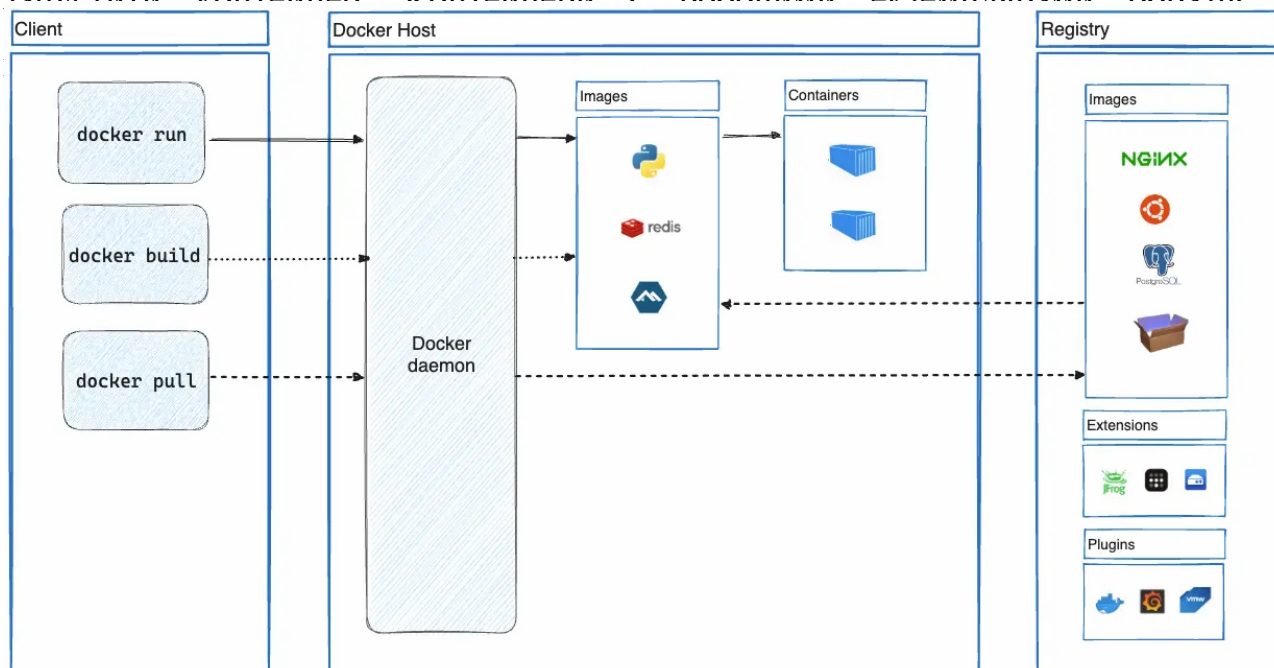
- образ (Image): Знімок файлової системи та налаштувань застосунку, який можна використовувати для запуску одного або декількох контейнерів
- контейнер (container): запущений екземпляр образу, що працює в ізольованому середовищі, використовуючи ресурси хост-системи, але ізольовано від неї завдяки технології віртуалізації на рівні ос.

Допоміжні інструменти та сервіси Docker:

- Container Registry – централізоване сховище для збереження та розповсюдження контейнерних образів. Популярні реєстри: Docker Hub, Google Container Registry (GCR), Amazon Elastic Container Registry (ECR)
- Docker Compose – інструмент для керування кількома взаємопов'язаними контейнерами через єдиний YAML-файл. Застосовується для розгортання складних застосунків, що містять кілька сервісів (наприклад, база даних, бекенд, фронтенд)
- Docker Swarm – вбудований механізм оркестрації контейнерів, що дозволяє керувати їхнім розподілом, масштабуванням і балансуванням навантаження в кластері.

Docker використовує архітектуру клієнт-сервер. Як зазначено на рисунку 1.8, клієнт Docker взаємодіє з демоном Docker, який виконує важку роботу щодо створення, запуску та поширення контейнерів. Клієнт і демон Docker можуть працювати на одній системі або підключатися віддалено. Взаємодія між ними здійснюється через REST API, UNIX-сокети або мережний інтерфейс. Крім основного клієнта, існує також Docker Compose, що дозволяє керувати багатоконтейнерними додатками. Docker Daemon управляє образами та контейнерами, використовуючи команди, отримані від клієнта, такі як `docker run`, `docker build` і `docker pull`. Образи можуть бути локальними або завантаженими з реєстру Docker, де зберігаються популярні сервіси, наприклад, NGINX, PostgreSQL та Ubuntu. Коли клієнт виконує команду `docker pull`, демон

отримує образ із реєстру та зберігає його локально, після чого з нього можна запустити контейнер. Контейнери є робочими екземплярами образів і



Контейнеризація забезпечує легкість розгортання, портативність застосунку між різними середовищами (локальне середовище розробки, сервер CI, хмарна платформа) та зниження накладних витрат, пов'язаних із традиційною віртуалізацією. Теоретичні основи контейнеризації базуються на концепції ізольованих процесів, що спільно використовують ядро операційної системи, але мають власну файлову систему, мережеві налаштування та ресурси.

Попри всі переваги контейнеризації, у разі складної розподіленої архітектури (наприклад, мікросервісної) виникає потреба в автоматизованому керуванні контейнерами, балансуванні навантаження, обробленні сбоїв (failover), масштабуванні та оновленні. Ці завдання вирішують системи оркестрації, зокрема:

1. Kubernetes (K8s). Розроблений спочатку в Google, а згодом переданий до Cloud Native Computing Foundation, Kubernetes став де-факто стандартом оркестрації. Він дає змогу описувати бажаний стан кластеру (кількість реплік контейнерів, конфігурації мережі, секрети тощо) у декларативних маніфестах (YAML-файлах), а також автоматично підтримує цей стан.
2. Docker Swarm. Побудований як «вбудована» функціональність Docker,

Swarm менш складний у використанні, ніж Kubernetes, але й поступається йому за гнучкістю та кількістю доступних налаштувань.

3. Apache Mesos / Marathon. Більш універсальне рішення, що підтримує не тільки контейнери, але й різноманітні «фреймворки» для обробки даних (Spark, Hadoop). Проте з часом популярність Mesos знизилася на користь Kubernetes.

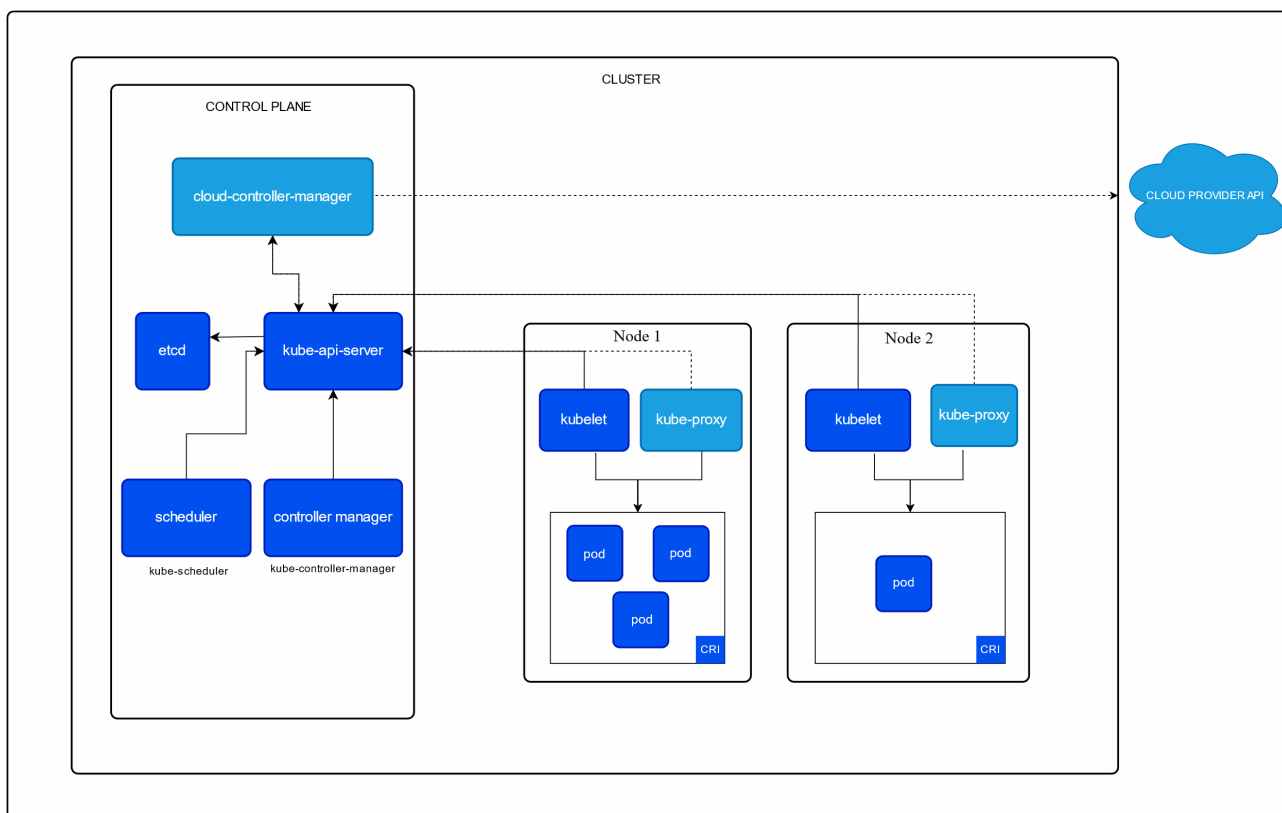
Ключова ідея оркестрації полягає в абстрагуванні від окремих серверів/вузлів і наданні розробникам або DevOps-інженерам можливості описувати бажаний стан застосунку. У цьому контексті система сама визначає, як розподілити контейнери між вузлами, проводити оновлення чи перевіряти здоров'я (health checks).

На сьогоднішній день найбільш розповсюдженим рішенням для оркестрації контейнерів є Kubernetes – це портативна, розширювана платформа з відкритим вихідним кодом для управління контейнеризованими робочими навантаженнями та сервісами, яка спрощує декларативну конфігурацію та автоматизацію.

Kubernetes має такі основні компоненти:

- pod – найменша керована одиниця, що може містити один або кілька контейнерів
- node (вузол) – фізичний або віртуальний сервер, на якому запускаються поди
- cluster – група вузлів, якими керує Kubernetes
- deployment – ресурс для управління оновленнями та масштабуванням застосунків
- service – логічне представлення набору подів, що надає стабільну точку доступу до них

Архітектура Kubernetes базується на моделі «клієнт-сервер» і складається з двох основних компонентів: контрольної площини (Control Plane) та робочих вузлів (Worker Nodes) (див. Рис 1.9) [18].



Контрольна площина (Control Plane):

- **API-сервер (kube-apiserver):** Центральний компонент, який надає API Kubernetes через HTTP (зазвичай у форматі JSON). Він обробляє та перевіряє REST-запити, а також оновлює стан об'єктів у сховищі даних
- **etcd:** Розподілене сховище ключ-значення, яке надійно зберігає конфігураційні дані кластера, представляючи поточний стан кластера. Інші компоненти відстежують зміни в цьому сховищі для досягнення бажаного стану
- **планувальник (kube-scheduler):** Відповідає за розподіл нових подів на відповідні вузли, базуючись на доступних ресурсах та інших обмеженнях. Він відстежує використання ресурсів на кожному вузлі, щоб забезпечити оптимальний розподіл навантаження.
- **менеджер контролерів (kube-controller-manager):** Запускає основні

контролери Kubernetes, такі як DaemonSet Controller та Replication Controller. Контролери взаємодіють з API-сервером для створення, оновлення та видалення керованих ресурсів (подів, сервісів тощо).

Робочі вузли (Worker Nodes):

- Kubelet: Агент, що працює на кожному вузлі та забезпечує, щоб всі контейнери в подах були в здоровому стані. Він відповідає за запуск, зупинку та підтримку контейнерів додатків відповідно до інструкцій контрольної площини
- Kube-proxy: Мережева проксі-служба та балансувальник навантаження, що підтримує абстракцію сервісу та інші мережеві операції. Він відповідає за маршрутизацію трафіку до відповідних контейнерів на основі IP-адреси та номера порту запиту
- контейнерний рантайм: Середовище виконання контейнерів, таке як containerd або CRI-O, яке відповідає за запуск та управління життєвим циклом контейнерів на вузлі.

Така архітектура забезпечує масштабоване та надійне управління контейнеризованими додатками, дозволяючи автоматизувати розгортання, масштабування та управління додатками в різних середовищах.

У типовому конвеєрі CI/CD використання контейнерів може відбуватися на кількох рівнях:

1. Збірка контейнерного образу під час CI. Замість створення бінарного артефакту (файлу .jar, .war тощо), система безперервної інтеграції формує Docker-образ, який містить усі необхідні залежності. Подібний образ далі передається в сховище контейнерів (Docker Registry, GitLab Container Registry, Amazon ECR тощо).
2. Автоматизація тестування. Тестовий етап часто виконується у тимчасово створених контейнерах, які автоматично знищуються після завершення. Це економить ресурси та уніфікує середовища: тести «бачать» ті самі налаштування, що й на продакшні.
3. Деплоймент контейнерів у середовище. Під час етапу CD (Continuous

Delivery або Continuous Deployment) оркестратор, наприклад Kubernetes, отримує вказівку «оновити» певний сервіс до нової версії контейнерного образу. Вказівка може бути подана з CI/CD-сервісу (наприклад, Jenkins Pipelines, GitLab CI/CD, GitHub Actions). Оркестратор виконує Rolling Update, Blue-Green Deployment чи Canary Release — залежно від налаштованого стратегії.

Переваги застосування контейнеризації та оркестрації у CI/CD:

- стабільність і відтворюваність: збірка та тестування відбуваються в ідентичних умовах незалежно від середовища, де запускаються (локальний комп'ютер розробника, сервер CI, хмара).
- масштабованість: за потреби можна паралельно запускати кілька контейнерів для тестів чи обробки навантаження у продакшні.
- прозорість деплойменту: декларативні маніфести в Kubernetes або YAML-конфігурації Docker Swarm дозволяють чітко описати, як система має виглядати у бойовому режимі, й перевіряти (linting) коректність цих конфігурацій ще на етапі CI.
- полегшення відкату (rollback): оскільки кожна версія ПЗ має відповідний контейнерний образ, відкат полягає в «заміщенні» версії образу на попередню, а оркестратор сам виконує потрібні дії для відновлення.

Складнощі та потенційні ризики:

1. Керування версіями образів. Без належної політики найменування та зберігання можуть накопичуватися зайві, застарілі образи, що спричиняє плутанину й збільшує витрати ресурсів. Необхідно впроваджувати процеси очищення (garbage collection) та узгоджені схеми тегування (наприклад, *major.minor.patch*-формат).
2. Безпека контейнерів. Контейнер не є «мікровіртуальною машиною» з повною ізоляцією. Якщо його налаштувати неправильно, це може спричинити вразливість усієї системи (наприклад, використання *root*-привілеїв у Docker-контейнері). Рекомендується регулярно

сканувати образи на відомі вразливості та автоматизувати процес оновлення базових (base) образів.

3. Складність налаштування оркестрації. Kubernetes хоч і є потужним, проте порівняно складним для початківців. Некоректні конфігурації (зокрема, параметрів безпеки чи мережевих політик) можуть призвести до збоїв у масштабуванні або неналежного розподілу навантаження. Компаніям часто доводиться призначати окремих DevOps-інженерів або SRE (Site Reliability Engineers), які спеціалізуються саме на Kubernetes.
4. Надмірна фрагментація середовищ. Звичка розгортати надто багато мікросервісів ускладнює керування залежностями між ними, що, своєю чергою, збільшує витрати на підтримку CI/CD-процесів.

Приклади успішного впровадження CI/CD за допомогою контейнеризації й оркестрації:

- Netflix. Компанія однією з перших масово перейшла на мікросервісну архітектуру, використовуючи власні інструменти (Spinnaker) й AWS для масштабування. Автоматизований конвеєр дозволяє випускати оновлення для окремих сервісів, запускаючи канарковий деплоймент і збираючи метрики. Якщо показники погіршуються, відбувається автоматичний відкат.
- Airbnb. Використовує Docker і Kubernetes для тестових середовищ, де кожен розробник може швидко розгорнути клон бойового середовища та інтегрувати нові фічі. CI/CD-процес включає не лише юніт-тести, а й E2E-тести, що перевіряють сумісність кількох сервісів одночасно.
- GitLab. Сам GitLab пропонує GitLab Runner, що запускає job (тести, збирання) всередині контейнерів. Застосування Docker-in-Docker (DinD) дає змогу будувати нові образи та випускати оновлення безпосередньо з самого інструмента розробки.

РОДІЛ 2

АНАЛІЗ ІСНУЮЧИХ ПІДХОДІВ ТА ІНСТРУМЕНТІВ CI/CD

2.1 Огляд сучасних систем автоматизації

У сучасній практиці розробки програмного забезпечення використовується широкий спектр систем автоматизації CI/CD, які надають різноманітні можливості для організації неперервної інтеграції та доставки. Для побудови ефективних CI/CD-процесів сьогодні застосовують як серверні рішення, так і хмарні сервіси, а також спеціалізовані інструменти для конкретних аспектів CI/CD.

Серед серверних рішень CI/CD особливе місце займає Jenkins – одна з найбільш популярних і найстаріших систем автоматизації з відкритим кодом. Після свого випуску в 2011 році (до цього відомий як Hudson) Jenkins став стандартом де-факто для багатьох організацій [19, с. 4]. Система відрізняється такими характеристиками: відкритий вихідний код та велика екосистема плагінів, гнучка конфігурація через DSL (Domain Specific Language) або графічний інтерфейс, підтримка розподілених збірок через концепцію Jenkins Agents, та можливість створення складних пайплайнів з умовними переходами. Архітектура Jenkins заснована на моделі «майстер-агент», де центральний сервер координує роботу агентів, які виконують завдання (див. рисунок 2.1). Однак, ця архітектура має певні обмеження щодо масштабованості при дуже великих навантаженнях.

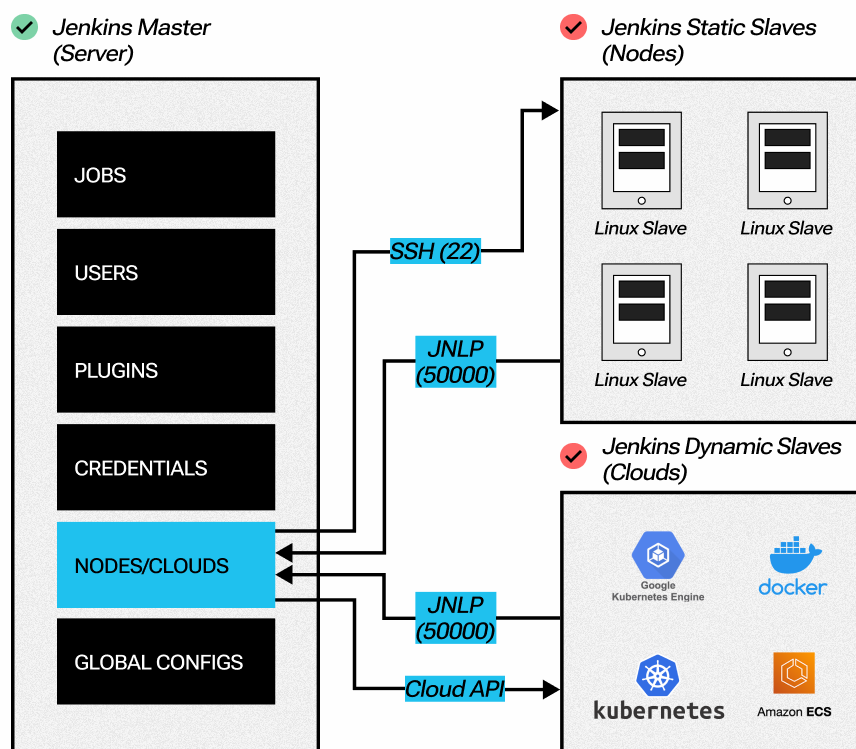


Рисунок 2.1 – Архітектура Jenkins у розподіленому середовищі [20]

Серед інших серверних рішень варто відзначити TeamCity від JetBrains – корпоративне рішення з пропрієтарною ліцензією, що пропонує потужні інструменти для CI/CD. Ключовими особливостями TeamCity є інтуїтивно зрозумілий користувацький інтерфейс, розширені можливості для тестування, включаючи паралельні тести, детальний перегляд історії зборок та змін коду, а також інтеграція з популярними IDE від JetBrains. Також важливим інструментом є Bamboo від Atlassian – інтегроване рішення для компаній, які вже використовують інші продукти Atlassian, такі як Jira і Bitbucket, що забезпечує тісну інтеграцію з іншими продуктами Atlassian, інтегровані плани релізів, вбудовані агенти для тестування на різних платформах та налаштування для автоматичного створення та розгортання Docker-контейнерів.

Із розвитком хмарних технологій та сервісного підходу до програмного забезпечення (SaaS) з'явилися потужні CI/CD рішення, які не потребують власної інфраструктури. Одним із провідних хмарних рішень є GitLab CI/CD, який є частиною екосистеми GitLab та пропонує інтегроване рішення для

всього життєвого циклу DevOps. Особливості GitLab CI/CD включають єдину платформу для керування кодом, CI/CD та моніторингу, конфігурацію пайплайнів через YAML-файли, що зберігаються в репозиторії, автоматичний запуск пайплайнів при змінах у репозиторії, розподілене виконання завдань через GitLab Runners, а також нативну підтримку контейнеризації та Kubernetes [21]. Спрощену архітектуру GitLab CI/CD зображено на рисунку 2.2.

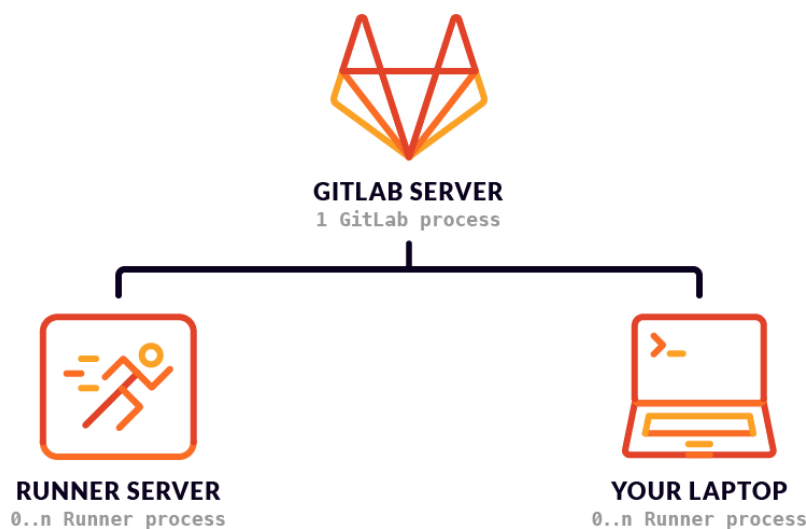


Рисунок 2.2 – Спрощена архітектура GitLab CI/CD

GitHub Actions також є важливим хмарним рішенням — відносно новим, але потужним, що дозволяє автоматизувати процеси збирання, тестування та розгортання безпосередньо з GitHub-репозиторіїв. Ключові можливості GitHub Actions включають інтегрований CI/CD безпосередньо в GitHub-екосистемі, легке повторне використання робочих процесів (workflows), розширену матрицю інструментів та середовищ для тестування та багатий маркетплейс готових дій (actions) для різних завдань.

CircleCI представляє собою хмарне рішення CI/CD, сфокусоване на продуктивності та швидкості. Його основні характеристики включають паралельне виконання завдань для прискорення процесу збирання, розширені можливості кешування для пришвидшення робочих процесів, підтримку оркестрації контейнерів через інтеграцію з Kubernetes, а також гнучку конфігурацію через YAML-файли.

Azure DevOps (колишній VSTS) є рішенням від Microsoft, що пропонує повний набір інструментів для DevOps, включаючи Azure Pipelines для CI/CD. Його ключові можливості включають тісну інтеграцію з екосистемою Microsoft Azure, підтримку багатоплатформених агентів (Windows, Linux, macOS), розширену підтримку контейнеризації та Kubernetes, а також потужні можливості для веб-хуків та API-інтеграцій.

Окрім універсальних систем CI/CD, існують спеціалізовані інструменти, що фокусуються на окремих аспектах процесу.

Spinnaker — платформа з відкритим кодом для неперервного розгортання, створена Netflix, що спеціалізується на складних мультихмарних розгортаннях. Особливості Spinnaker включають підтримку розгортання на кілька хмарних платформ одночасно, складні стратегії розгортання (блакитно-зелене, канаркове), можливість відкату та відновлення після проблем, а також інтеграцію з системами моніторингу для аналізу якості розгортання.

ArgoCD є інструментом для неперервного розгортання, що використовує декларативний підхід для розгортання на Kubernetes. Його основні можливості включають GitOps-підхід, при якому джерелом істини є Git-репозиторій, автоматичну синхронізацію стану кластеру з бажаним станом у Git, потужну візуалізацію стану розгортання, а також підтримку Helm, Kustomize та інших інструментів для Kubernetes.

Tekton — ще один важливий фреймворк для створення CI/CD-систем, спеціально розроблений для Kubernetes та хмарних середовищ. Ключові характеристики Tekton включають компоненти CI/CD, представлені як Kubernetes Custom Resources, модульність завдань і можливість повторного використання, розподілене виконання завдань у кластері Kubernetes, а також виконання процесів у ізольованих контейнерах.

Для ефективного функціонування CI/CD необхідні додаткові інструменти, що забезпечують різні етапи процесу. Більшість з цих інструментів вже згадувалися в першому розділі, тож нагадаємо про їхню ключову роль у CI/CD.

Docker — платформа для контейнеризації, що дозволяє пакувати застосунки разом з їхніми залежностями. У контексті CI/CD Docker забезпечує стандартизоване середовище для збирання та тестування, консистентність між різними етапами CI/CD, можливість швидкого розгортання через контейнерні образи, а також ізоляцію процесів та безпеку виконання.

Kubernetes — система оркестрації контейнерів, що забезпечує автоматизоване розгортання, масштабування та управління контейнеризованими застосунками. У рамках CI/CD Kubernetes надає декларативне управління інфраструктурою, автоматичне масштабування та балансування навантаження, керування секретами та конфігураціями, а також стратегії розгортання з мінімальним простоем.

Terraform є інструментом для інфраструктури як код (Infrastructure as Code, IaC), що дозволяє визначати та створювати інфраструктуру через декларативні конфігураційні файли. У CI/CD-процесах Terraform забезпечує відтворюваність інфраструктури, версіонування інфраструктурних змін, автоматизоване створення та модифікацію середовищ, а також підтримку різних хмарних провайдерів.

Prometheus і Grafana — інструменти для моніторингу та візуалізації, які інтегруються з CI/CD для забезпечення спостережуваності процесів. Вони надають моніторинг продуктивності CI/CD-пайплайнів, візуалізацію метрик та створення панелей моніторингу, налаштування сповіщень при критичних ситуаціях, а також аналіз тенденцій та виявлення аномалій.

Наявність такого широкого спектру інструментів CI/CD дозволяє організаціям вибирати оптимальні рішення відповідно до їхніх потреб, розміру, складності проєктів та існуючої технологічної екосистеми. Однак, різноманітність опцій також ускладнює процес вибору та вимагає ретельного аналізу сильних і слабких сторін кожного інструменту.

2.2 Порівняння інструментів за критеріями гнучкості, продуктивності та безпеки

Вибір оптимального інструменту CI/CD для розподіленої системи потребує комплексного аналізу за ключовими критеріями: гнучкість налаштування, продуктивність обробки завдань та забезпечення безпеки. Різні інструменти демонструють різну ефективність за кожним із цих критеріїв, що важливо враховувати при прийнятті рішення.

Важливо зазначити, що для всі подальших порівнянь було використано офіційну документацію кожного CI/CD-інструменту [22 – 27].

Гнучкість CI/CD-інструментів визначається їхньою здатністю адаптуватися до різних потреб проекту, підтримувати різноманітні технології та забезпечувати розширюваність. Порівнюючи методи конфігурації та можливості налаштування, можна відзначити, що більшість сучасних інструментів підтримують підхід «Infrastructure as Code» через конфігураційні файли у форматі YAML, що спрощує версіонування та автоматизацію налаштувань (див. у таблиці 2.1). GitLab CI/CD та GitHub Actions виділяються своєю простотою та декларативним підходом, тоді як Jenkins, хоч і пропонує найбільшу гнучкість, вимагає більш глибоких знань для налаштування [28, с. 60].

Таблиця 2.1 – Порівняння гнучкості методів конфігурації CI/CD-інструментів

Інструмент	Метод конфігурації	Підтримка кодування налаштувань (IaC)	Складність налаштування
Jenkins	XML/Groovy (Jenkinsfile)	Так, через Jenkinsfile	Висока, вимагає глибокого розуміння
GitLab CI/CD	YAML (.gitlab-ci.yml)	Так, через YAML	Середня, проста структура файлів
GitHub Actions	YAML (workflow files)	Так, декларативний підхід	Низька, інтуїтивний формат повернення кожного інстансу до попередньої версії

TeamCity	UI + Kotlin DSL	Частково через Kotlin DSL	Середня, зручний UI

Продовження таблиці 2.1

CircleCI	YAML (config.yml)	Так,, через YAML	Низька, добра документація
Azure DevOps	YAML або графічний редактор	Так, через YAML	Середня, різні способи

Важливим аспектом гнучкості є здатність інструменту CI/CD працювати з різними мовами програмування, фреймворками та технологіями розгортання. Аналіз на рисунку 2.3 показує, що GitLab CI/CD та Jenkins пропонують найширшу підтримку різних технологій завдяки своїй архітектурі та екосистемам розширень. GitHub Actions також швидко розвивається і має значну перевагу завдяки маркетплейсу готових дій.

■ 2 - Повна підтримка ■ 1 - Часткова підтримка

Категорія/Технологія	Jenkins	GitLab CI/CD	GitHub Actions	TeamCity	CircleCI	Azure DevOps	Числове значення
Мови програмування							
Java	Повна	Повна	Повна	Повна	Повна	Повна	2,2,2,2,2,2
Python	Повна	Повна	Повна	Повна	Повна	Повна	2,2,2,2,2,2
JavaScript	Повна	Повна	Повна	Повна	Повна	Повна	2,2,2,2,2,2
.NET	Повна	Повна	Повна	Повна	Часткова	Повна	2,2,2,2,1,2
Контейнеризація та оркестрація							
Docker	Повна	Повна	Повна	Повна	Повна	Повна	2,2,2,2,2,2
Kubernetes	Повна	Повна	Повна	Часткова	Повна	Повна	2,2,2,1,2,2
Хмарні платформи							
AWS	Повна	Повна	Повна	Повна	Повна	Часткова	2,2,2,2,2,1
Azure	Повна	Повна	Повна	Повна	Повна	Повна	2,2,2,2,2,2
Google Cloud (GCP)	Повна	Повна	Повна	Часткова	Повна	Часткова	2,2,2,1,2,1
Infrastructure as Code (IaC)							
Terraform	Повна	Повна	Повна	Часткова	Повна	Повна	2,2,2,1,2,2
Тестування							
Модульне тестування	Повна	Повна	Повна	Повна	Повна	Повна	2,2,2,2,2,2
E2E тестування	Повна	Повна	Повна	Часткова	Повна	Повна	2,2,2,1,2,2
Тестування безпеки	Часткова	Повна	Повна	Часткова	Часткова	Повна	1,2,2,1,1,2
Моніторинг та спостережність							
Інтеграції моніторингу	Часткова	Повна	Часткова	Часткова	Повна	Часткова	1,2,1,1,2,1

Рисунок 2.3 – Порівняння підтримки різних технологій CI/CD-інструментами

Здатність системи CI/CD розширюватися через плагіни, інтеграції та API є критично важливою для адаптації до специфічних потреб організації. Jenkins залишається лідером за кількістю доступних плагінів (більше 1800) [29], однак GitHub Actions швидко нарощує свою екосистему через концепцію повторно використовуваних дій. GitLab CI/CD, хоч і має меншу кількість зовнішніх розширень, пропонує комплексний набір вбудованих можливостей, що знижує потребу в додаткових інтеграціях.

Продуктивність CI/CD-системи є критичним фактором, особливо для великих розподілених команд, де затримки можуть суттєво впливати на швидкість розробки. Щоб порівняти швидкість виконання типових CI/CD-завдань на основі емпіричних даних, отриманих при тестуванні, було обрано стандартизований проєкт. Spring PetClinic - класичний демонстраційний проєкт на Java Spring Boot, який часто використовується для тестування та навчання. Має помірну складність і добре підходить для CI/CD-експериментів [30]. Важливо відзначити, що CircleCI демонструє найкращу загальну продуктивність, особливо в частині швидкої ініціалізації та розгортання (див. у таблиці 2.2). GitLab CI/CD та GitHub Actions також показують високу продуктивність завдяки оптимізованій архітектурі. Jenkins, попри свою гнучкість, демонструє дещо нижчу продуктивність, особливо на етапі ініціалізації.

Таблиця 2.2 – Порівняння швидкості виконання CI/CD-завдань (у секундах)

Інструмент	Ініціалізація	Збирання проєкту	Виконання тестів	Розгортання	Загальний час	
Jenkins	12	45	180	25	262	
GitLab CI/CD	8	40	175	20	243	
GitHub Actions	10	42	170	22	244	
TeamCity	15	38	165	24	242	
CircleCI	7	43	168	18	236	
Azure	14	44	172	21	251	

DevOps						
--------	--	--	--	--	--	--

Для розподілених систем із великою кількістю розробників та інтенсивними циклами інтеграції критично важливою є здатність CI/CD-системи масштабуватися під навантаженням. Аналіз показує (див. рисунок 2.4), що GitLab CI/CD з використанням Kubernetes runners та CircleCI демонструють найкращу масштабованість при високих навантаженнях. Jenkins, хоч і здатний

■ 4 - Відмінно ■ 3 - Добре ■ 2 - Задовільно ■ 1 - Обмежено

Критерій масштабованості	Jenkins	GitLab CI/CD	GitHub Actions	TeamCity	CircleCI	Azure DevOps	Числове значення
Базові характеристики масштабованості							
Максимальна кількість паралельних завдань	Відмінно	Відмінно	Добре	Добре	Відмінно	Добре	4,4,3,3,4,3
Горизонтальне масштабування	Добре	Відмінно	Добре	Задовільно	Відмінно	Добре	3,4,3,2,4,3
Вертикальне масштабування	Добре	Добре	Добре	Добре	Добре	Добре	3,3,3,3,3,3
Розподілена архітектура	Відмінно	Відмінно	Задовільно	Добре	Відмінно	Добре	4,4,2,3,4,3
Сценарії високих навантажень							
Велика кількість мікросервісів (100+)	Задовільно	Відмінно	Добре	Задовільно	Відмінно	Добре	2,4,3,2,4,3
Велика команда розробників (50+)	Добре	Відмінно	Відмінно	Добре	Відмінно	Відмінно	3,4,4,3,4,4
Велика кількість одночасних запитів	Задовільно	Відмінно	Добре	Задовільно	Відмінно	Добре	2,4,3,2,4,3
Технічна інфраструктура							
Інтеграція з Kubernetes	Добре	Відмінно	Добре	Задовільно	Відмінно	Добре	3,4,3,2,4,3
Автомасштабування воркерів	Задовільно	Відмінно	Відмінно	Задовільно	Відмінно	Добре	2,4,4,2,4,3
Розподілене кешування	Задовільно	Добре	Добре	Добре	Відмінно	Добре	2,3,3,3,4,3
Продуктивність при масштабуванні							
Ефективність використання ресурсів	Задовільно	Добре	Добре	Добре	Відмінно	Добре	2,3,3,3,4,3
Швидкість запуску нових агентів	Обмежено	Відмінно	Відмінно	Задовільно	Відмінно	Добре	1,4,4,2,4,3
Стабільність при пікових навантаженнях	Задовільно	Відмінно	Добре	Задовільно	Відмінно	Добре	2,4,3,2,4,3
Адміністрування та моніторинг							
Простота налаштування масштабування	Обмежено	Добре	Відмінно	Задовільно	Відмінно	Добре	1,3,4,2,4,3
Моніторинг продуктивності	Задовільно	Добре	Добре	Добре	Відмінно	Відмінно	2,3,3,3,4,4
Діагностика проблем масштабування	Задовільно	Добре	Задовільно	Задовільно	Добре	Добре	2,3,2,2,3,3

Рисунок 2.4 – Порівняння масштабованості CI/CD-інструментів при різних навантаженнях продуктивності.

Оцінка ефективності використання обчислювальних ресурсів є ключовою для оптимізації витрат на інфраструктуру CI/CD. Ефективне управління ресурсами дозволяє забезпечити необхідну якість обслуговування (QoS) при мінімальних витратах.

Проведемо дослідження з використанням розробленої математичної моделі оцінки ефективності CI/CD інфраструктури. Основні параметри оцінки:

- час виконання: тривалість проходження змін коду від коміту до розгортання ($w = 0,3$)
- частота розгортання: як часто нові версії впроваджуються в продуктивне середовище ($f = 0,3$)
- рівень невдач змін: відсоток змін, які зазнають невдачі після розгортання ($r = 0,2$)
- середній час відновлення (MTTR): середній час, необхідний для відновлення після збою ($t = 0,2$)

Використаємо формулу для розрахунку ефективності (2.1):

- де – ефективність CI/CD технології;
- вагові коефіцієнти оцінки параметрів;
 - нормований час виконання pipeline;
 - нормована частота розгортання;
 - нормований рівень невдач змін;
 - нормований середній час відновлення.

Вхідні дані (приклад):

- час виконання: 5 хв (еталон 10 хв)
- частота розгортань: 12 разів/тиждень (оптимум 12)
- рівень невдач: 4% (максимум 10%)
- час відновлення: 1,5 години (максимум 4 години)

Розрахунок нормалізованих значень зазначених у формулах (2.2 – 2.5):

- фактичне значення часу виконання;
 - еталонне значення часу виконання.
-
- фактичне значення частоти розгортань;
 - оптимальне значення частоти розгортань.
-
- фактичне значення рівня невдач;
 - максимально допустимий рівень невдач.
-
- фактичне значення часу відновлення;
 - максимально допустимий рівень невдач.

Технології дослідження:

1. Jenkins
2. GitLab CI/CD
3. GitHub Actions
4. TeamCity
5. CircleCI
6. Azure DevOps

Шляхом обчислень було отримано результати зображені на рисунку 2.5.

CI/CD Технології							
Метрика	Цільові показники	Jenkins	GitLab CI/CD	GitHub Actions	TeamCity	CircleCI	Azure DevOps
Основні метрики ефективності							
Час виконання CI/CD pipeline	Benchmark: 10 хв	7 хв	6 хв	5 хв	8 хв	6.5 хв	7.5 хв
Частота розгортання	Optimal: 12 разів/тиждень	8 разів/тиждень	10 разів/тиждень	12 разів/тиждень	7 разів/тиждень	9 разів/тиждень	8 разів/тиждень
Рівень невдач змін	Target max: 10%	7%	5%	4%	8%	6%	7%
Середній час відновлення (MTTR)	Target max: 4 год	3 год	2 год	1.5 год	3.5 год	2.5 год	3 год
Нормалізовані показники							
Нормалізований час виконання	1 - (Час/Benchmark)	0.30	0.40	0.50	0.20	0.35	0.25
Нормалізована частота розгортання	Частота/Optimal	0.667	0.833	1.000	0.583	0.750	0.667
Нормалізований рівень невдач	1 - (Рівень/Target)	0.30	0.50	0.60	0.20	0.40	0.30
Нормалізований MTTR	1 - (MTTR/Target)	0.25	0.50	0.625	0.125	0.375	0.25
Розрахунок ефективності (ваги: 0.3, 0.3, 0.2, 0.2)							
Час виконання (W1 = 0.3)	0.3 × Норм. час	0.09	0.12	0.15	0.06	0.105	0.075
Частота розгортання (W2 = 0.3)	0.3 × Норм. частота	0.200	0.250	0.300	0.175	0.225	0.200
Рівень невдач змін (W3 = 0.2)	0.2 × Норм. рівень	0.06	0.10	0.12	0.04	0.08	0.06
Час відновлення (W4 = 0.2)	0.2 × Норм. MTTR	0.050	0.100	0.125	0.025	0.075	0.050
Підсумкові результати							
Загальна ефективність	Сума зважених значень	0.490	0.570	0.720	0.450	0.535	0.485
Ранг	Позиція в рейтингу	4	2	1	6	3	5
Інтерпретація	Категорія ефективності	Помірна ефективність	Середня ефективність	Висока ефективність	Задовільна ефективність	Середня ефективність	Помірна ефективність

Рисунок 2.5 – Результати оцінки ефективності CI/CD інфраструктури

GitHub Actions (0.72) показав найвищу ефективність завдяки найшвидшому часу виконання (5 хв), максимальній частоті розгортань (12 разів/тиждень), найнижчому рівню невдач (4%) та найшвидшому часу відновлення (1.5 год). GitLab CI/CD (0.57) займає друге місце з добре збалансованими показниками у всіх категоріях, демонструючи особливо хороші результати у швидкості виконання та частоті розгортань. CircleCI (0.535) показує середню ефективність з хорошими показниками часу виконання та частоти розгортань. Jenkins (0.49) та Azure DevOps (0.485) демонструють помірну ефективність, з показниками нижчими ніж у лідерів, але все ще прийнятними. TeamCity (0.45) має найнижчу ефективність серед усіх досліджених технологій, особливо відстаючи у швидкості виконання та часі

відновлення.

Важливо зазначити, що дослідження було проведене на базі демонстраційного проєкту Spring PetClinic [30], результати є умовними та залежать від конкретної реалізації та налаштувань для кожної технології.

Безпека CI/CD-процесів є критично важливою, оскільки конвеєр поставки може стати точкою вразливості для всієї системи. Здатність безпечно зберігати та використовувати секретні дані (ключі, «креденшали») є фундаментальною вимогою до CI/CD-системи. Порівняння у таблиці 2.3 показує, що GitLab CI/CD та GitHub Actions пропонують найбільш розвинені механізми захисту секретів із вбудованим шифруванням, детальним контролем доступу та повним маскуванням у логах. Всі інструменти підтримують інтеграцію з зовнішніми сховищами секретів, що є важливим для корпоративних середовищ.

Таблиця 2.3 – Порівняння методів захисту секретів у CI/CD-інструментах

Інструмент	Механізм зберігання секретів	Інтеграція з зовнішніми сховищами	Обмеження доступу	Маскування у логах
Jenkins	Credentials Plugin	HashiCorp Vault, AWS Secrets	Рольова модель	Обмежене
GitLab CI/CD	Вбудоване шифроване сховище	HashiCorp Vault, K8s Secrets	Детальні дозволи	Повне
GitHub Actions	Encrypted Secrets	Azure KeyVault, HashiCorp Vault	Repository-scoped	Повне
TeamCity	Password values	HashiCorp Vault, AWS параметри	Проектні дозволи	Обмежене
CircleCI	Context & Environment Variables	AWS/GCP Secret Managers	Org-level contexts	Повне
Azure DevOps	Variable Groups	Azure KeyVault	Детальні дозволи	Повне

Оскільки більшість CI/CD-процесів базуються на використанні контейнерів, безпека контейнерів стає критичним аспектом. GitLab CI/CD виділяється комплексним підходом до безпеки контейнерів з вбудованими інструментами сканування залежностей, зображень та інфраструктурного коду (див. рисунок 2.6). Jenkins та інші інструменти потребують додаткових плагінів або інтеграцій для забезпечення аналогічного рівня захисту.

3 - Вбудована функціональність 2 - Через офіційну інтеграцію 1 - Через сторонні плагіни
0 - Обмежена/відсутня

Вид сканування контейнерів	Jenkins	GitLab CI/CD	GitHub Actions	TeamCity	CircleCI	Azure DevOps	Число значення
Базове сканування контейнерів							
Сканування вразливостей образів	Через плагіни	Вбудовано	Офіційна інтеграція	Через плагіни	Офіційна інтеграція	Офіційна інтеграція	1,3,2,1,2,2
Сканування базових образів	Через плагіни	Вбудовано	Офіційна інтеграція	Обмежено	Офіційна інтеграція	Офіційна інтеграція	1,3,2,0,2,2
Перевірка конфігурації контейнерів	Через плагіни	Вбудовано	Через плагіни	Обмежено	Через плагіни	Офіційна інтеграція	1,3,1,0,1,2
Розширене сканування							
Сканування залежностей в контейнерах	Через плагіни	Вбудовано	Вбудовано	Через плагіни	Офіційна інтеграція	Офіційна інтеграція	1,3,3,1,2,2
Перевірка секретів в контейнерах	Через плагіни	Вбудовано	Вбудовано	Обмежено	Через плагіни	Офіційна інтеграція	1,3,3,0,1,2
Динамічний аналіз контейнерів	Через плагіни	Офіційна інтеграція	Через плагіни	Обмежено	Через плагіни	Через плагіни	1,2,1,0,1,1
Інтеграція з інструментами безпеки							
Інтеграція з Trivy	Через плагіни	Вбудовано	Офіційна інтеграція	Через плагіни	Офіційна інтеграція	Офіційна інтеграція	1,3,2,1,2,2
Інтеграція з Clair	Через плагіни	Вбудовано	Через плагіни	Обмежено	Офіційна інтеграція	Через плагіни	1,3,1,0,2,1
Інтеграція з Anchore	Через плагіни	Офіційна інтеграція	Через плагіни	Обмежено	Офіційна інтеграція	Через плагіни	1,2,1,0,2,1
Інтеграція з Snyk	Через плагіни	Вбудовано	Вбудовано	Через плагіни	Офіційна інтеграція	Офіційна інтеграція	1,3,3,1,2,2
Управління результатами сканування							
Блокування збірки при критичних вразливостях	Через плагіни	Вбудовано	Офіційна інтеграція	Через плагіни	Офіційна інтеграція	Офіційна інтеграція	1,3,2,1,2,2
Візуалізація результатів сканування	Через плагіни	Вбудовано	Офіційна інтеграція	Обмежено	Офіційна інтеграція	Офіційна інтеграція	1,3,2,0,2,2
Відстеження та усунення вразливостей	Обмежено	Вбудовано	Офіційна інтеграція	Обмежено	Через плагіни	Офіційна інтеграція	0,3,2,0,1,2
Додаткові можливості							
Сканування IaC (Dockerfile, K8s manifest)	Через плагіни	Вбудовано	Офіційна інтеграція	Обмежено	Через плагіни	Офіційна інтеграція	1,3,2,0,1,2
Підписування контейнерів	Через плагіни	Вбудовано	Офіційна інтеграція	Обмежено	Через плагіни	Офіційна інтеграція	1,3,2,0,1,2
Compliance перевірки	Через плагіни	Вбудовано	Через плагіни	Обмежено	Через плагіни	Офіційна інтеграція	1,3,1,0,1,2

Рисунок 2.6 – Порівняння можливостей сканування контейнерів у CI/CD-інструментах

Системи розмежування доступу та відстеження дій користувачів є важливими аспектами безпеки CI/CD. GitLab CI/CD та Azure DevOps демонструють найбільш розвинені можливості контролю доступу з детальними дозволами та комплексним аудитом дій користувачів. Jenkins, хоч і має гнучкі можливості через плагіни, потребує додаткового налаштування для досягнення аналогічного рівня безпеки.

На основі комплексного аналізу зображеного на рисунку 2.7, можна зробити наступні висновки щодо порівняння інструментів CI/CD:

1. GitLab CI/CD демонструє найкращий баланс між гнучкістю, продуктивністю та безпекою, особливо для розподілених систем. Його інтеграція з Kubernetes та вбудована екосистема DevOps інструментів роблять його оптимальним вибором для багатьох сценаріїв.
2. Jenkins залишається лідером за гнучкістю та розширюваністю, але потребує більше зусиль для забезпечення оптимальної продуктивності та безпеки.
3. GitHub Actions демонструє швидкий розвиток та високу продуктивність, але має певні обмеження в корпоративних сценаріях, особливо для складних розподілених систем.
4. CircleCI виділяється високою продуктивністю та ефективністю використання ресурсів, що робить його привабливим для проєктів із інтенсивними процесами безперервної інтеграції.
5. Azure DevOps пропонує комплексне рішення з високим рівнем безпеки та інтеграції з екосистемою Microsoft, що є перевагою для організацій, що вже використовують Azure.
6. Незважаючи на найнижчі показники інтегрального балу (3.1), TeamCity демонструє кращі результати в паралельному виконанні завдань та швидкості виконання збірок, що може бути корисним для проєктів, де ці аспекти мають пріоритет.

5 - Відмінно 4 - Дуже добре 3 - Добре 2 - Задовільно

Критерій оцінки	Jenkins	GitLab CI/CD	GitHub Actions	TeamCity	CircleCI	Azure DevOps	Число значення
Гнучкість та інтеграція							
Інтеграція з SCM системами	Відмінно	Відмінно	Дуже добре	Дуже добре	Дуже добре	Дуже добре	5,5,4,4,4,4
Інтеграція з Kubernetes	Дуже добре	Відмінно	Дуже добре	Добре	Відмінно	Дуже добре	4,5,4,3,5,4
Підтримка різних мов програмування	Відмінно	Відмінно	Відмінно	Дуже добре	Дуже добре	Дуже добре	5,5,5,4,4,4
Розширюваність та екосистема плагінів	Відмінно	Дуже добре	Дуже добре	Добре	Добре	Дуже добре	5,4,4,3,3,4
Інтеграція з хмарними платформами	Дуже добре	Відмінно	Дуже добре	Добре	Дуже добре	Відмінно	4,5,4,3,4,5
Продуктивність та масштабованість							
Швидкість виконання збірок	Добре	Дуже добре	Відмінно	Дуже добре	Відмінно	Дуже добре	3,4,5,4,5,4
Масштабованість для великих проектів	Добре	Відмінно	Добре	Добре	Відмінно	Дуже добре	3,5,3,3,5,4
Паралельне виконання завдань	Дуже добре	Відмінно	Відмінно	Дуже добре	Відмінно	Дуже добре	4,5,5,4,5,4
Ефективність використання ресурсів	Задовільно	Дуже добре	Дуже добре	Добре	Відмінно	Дуже добре	2,4,4,3,5,4
Кешування та оптимізація збірок	Добре	Відмінно	Відмінно	Дуже добре	Відмінно	Дуже добре	3,5,5,4,5,4
Безпека							
Сканування вразливостей	Добре	Відмінно	Дуже добре	Задовільно	Дуже добре	Відмінно	3,5,4,2,4,5
Безпека контейнерів	Добре	Відмінно	Дуже добре	Задовільно	Дуже добре	Дуже добре	3,5,4,2,4,4
Управління секретами	Добре	Відмінно	Відмінно	Добре	Дуже добре	Відмінно	3,5,5,3,4,5
Контроль доступу та авторизація	Добре	Відмінно	Дуже добре	Добре	Дуже добре	Відмінно	3,5,4,3,4,5
Відповідність стандартам (Compliance)	Добре	Дуже добре	Добре	Добре	Добре	Відмінно	3,4,3,3,3,5
Зручність використання							
Легкість налаштування	Задовільно	Відмінно	Відмінно	Добре	Відмінно	Дуже добре	2,5,5,3,5,4
Зрозумілість конфігурації	Задовільно	Дуже добре	Відмінно	Добре	Відмінно	Дуже добре	2,4,5,3,5,4
Документація	Дуже добре	Відмінно	Відмінно	Добре	Дуже добре	Відмінно	4,5,5,3,4,5
Візуалізація та моніторинг	Добре	Відмінно	Дуже добре	Дуже добре	Дуже добре	Відмінно	3,5,4,4,4,5
Навчальна крива	Задовільно	Добре	Відмінно	Задовільно	Дуже добре	Добре	2,3,5,2,4,3
Специфічні особливості та екосистема							
Інтеграція з DevOps інструментами	Дуже добре	Відмінно	Дуже добре	Добре	Дуже добре	Відмінно	4,5,4,3,4,5
Підтримка для мікросервісної архітектури	Добре	Відмінно	Дуже добре	Добре	Відмінно	Дуже добре	3,5,4,3,5,4
Інтеграція з IaC інструментами	Дуже добре	Відмінно	Дуже добре	Добре	Дуже добре	Відмінно	4,5,4,3,4,5
Вартість для корпоративного використання	Відмінно	Дуже добре	Добре	Добре	Добре	Добре	5,4,3,3,3,3
Спільнота та екосистема	Відмінно	Відмінно	Відмінно	Добре	Дуже добре	Дуже добре	5,5,5,3,4,4
Загальні показники							
Загальний бал за гнучкість та інтеграцію	4.6	4.8	4.2	3.4	4.0	4.2	4,6,4,8,4,2,3,4,4,0,4,2
Загальний бал за продуктивність та масштабованість	3.0	4.6	4.4	3.6	5.0	4.0	3,0,4,6,4,4,3,6,5,0,4,0
Загальний бал за безпеку	3.0	4.8	4.0	2.6	3.8	4.8	3,0,4,8,4,0,2,6,3,8,4,8
Загальний бал за зручність використання	2.6	4.4	4.8	3.0	4.4	4.2	2,6,4,4,4,8,3,0,4,4,4,2
Загальний бал за специфічні особливості та екосистему	4.2	4.8	4.0	3.0	4.0	4.2	4,2,4,8,4,0,3,0,4,0,4,2
ІНТЕГРАЛЬНИЙ БАЛ	3.5	4.7	4.3	3.1	4.2	4.3	3,5,4,7,4,3,3,1,4,2,4,3

Рисунок 2.7 – Інтегральне порівняння CI/CD-інструментів за ключовими критеріями

Вибір конкретного інструменту має базуватися на специфічних потребах проєкту, існуючій інфраструктурі та вимогах до гнучкості, продуктивності та безпеки.

2.3 Методи інтеграції системи контролю версій, інфраструктури як код та авто тестування

Ефективний CI/CD-процес базується на глибокій інтеграції трьох ключових компонентів: системи контролю версій (VCS), інфраструктури як код (IaC) та комплексного авто-тестування. Кожен із цих компонентів відіграє важливу роль у створенні надійного та ефективного процесу розробки та поставки програмного забезпечення.

Системи контролю версій є фундаментом процесу CI/CD, забезпечуючи відстеження змін, колаборацію між розробниками та тригери для запуску автоматизованих процесів. Сучасні CI/CD-системи пропонують різноманітні механізми для реагування на події у системі контролю версій. На основі даних з таблиці 2.4, можна зробити висновок, що GitLab CI/CD та GitHub Actions пропонують найбільш розвинені моделі подій з детальним контролем тригерів та умов виконання. Це дозволяє налаштувати складні правила запуску пайплайнів, включаючи фільтрацію за гілками, тегами, шляхами та іншими атрибутами.

Таблиця 2.4 – Порівняння подієвих моделей інтеграції з VCS

Інструмент	Підтримувані події	Фільтрування по шляхах	Умовне виконання	Підтримка веб-хуків
Jenkins	commit, PR, tag, cron	Так, через pathspecSecrets	Через Jenkinsfile	Так, через плагіни
GitLab CI/CD	commit, merge request, tag, schedule, manual	Так, вбудоване	Через rules/only/except	Так, нативно
GitHub Actions	push, pull_request, workflow_dispatch, schedule	Так, через paths	Через if/conditions	Так, нативно

--	--	--	--	--

Продовження таблиці 2.4

TeamCity	VCS trigger, schedule	Через checkout rules	Через conditions	Так, через REST API
CircleCI	push, pull_request, approval, schedule	Через filters	Через when/conditions	Так, через API
Azure DevOps	push, PR, schedule, pipeline resources	Через path filters	Через conditions	Так, через Service Hooks

Ефективна CI/CD-система має підтримувати популярні стратегії розгалуження коду, такі як Git Flow, GitHub Flow та Trunk-Based Development. Як зображено на рисунку 2.8 GitLab CI/CD виділяється своєю детальною підтримкою різних стратегій розгалуження через концепцію «merge request pipelines» та «environment-specific deployments». GitHub Actions також добре інтегрується з GitHub Flow, що робить його природним вибором для проєктів, що використовують цю модель. Jenkins, завдяки своїй гнучкості, може бути адаптований під будь-яку стратегію, але потребує додаткового налаштування.

■ 4 - Нативна підтримка ■ 3 - Гарна підтримка ■ 2 - Потребує конфігурації ■ 1 - Обмежена підтримка

Стратегія розгалуження	Jenkins	GitLab CI/CD	GitHub Actions	TeamCity	CircleCI	Azure DevOps	Числове значення
Основні стратегії розгалуження							
Git Flow	Потребує конфігурації	Нативна підтримка	Гарна підтримка	Потребує конфігурації	Гарна підтримка	Гарна підтримка	2,4,3,2,3,3
GitHub Flow	Потребує конфігурації	Гарна підтримка	Нативна підтримка	Потребує конфігурації	Гарна підтримка	Гарна підтримка	2,3,4,2,3,3
GitLab Flow	Потребує конфігурації	Нативна підтримка	Гарна підтримка	Потребує конфігурації	Гарна підтримка	Гарна підтримка	2,4,3,2,3,3
Trunk-Based Development	Гарна підтримка	Нативна підтримка	Нативна підтримка	Гарна підтримка	Нативна підтримка	Гарна підтримка	3,4,4,3,4,3
Специфічні особливості підтримки розгалужень							
Pull/Merge Request Pipelines	Потребує конфігурації	Нативна підтримка	Нативна підтримка	Потребує конфігурації	Нативна підтримка	Нативна підтримка	2,4,4,2,4,4
Середовище-специфічні розгортання	Потребує конфігурації	Нативна підтримка	Гарна підтримка	Потребує конфігурації	Гарна підтримка	Нативна підтримка	2,4,3,2,3,4
Умовні збірки за гілками	Потребує конфігурації	Нативна підтримка	Нативна підтримка	Гарна підтримка	Нативна підтримка	Нативна підтримка	2,4,4,3,4,4
Паралельні збірки для різних гілок	Гарна підтримка	Нативна підтримка	Нативна підтримка	Гарна підтримка	Нативна підтримка	Гарна підтримка	3,4,4,3,4,3
Автоматизація та правила							
Автоматичні перевірки PR/MR	Потребує конфігурації	Нативна підтримка	Нативна підтримка	Потребує конфігурації	Гарна підтримка	Нативна підтримка	2,4,4,2,3,4
Правила захисту гілок	Обмежена підтримка	Нативна підтримка	Нативна підтримка	Обмежена підтримка	Гарна підтримка	Нативна підтримка	1,4,4,1,3,4
Code Review інтеграція	Потребує конфігурації	Нативна підтримка	Нативна підтримка	Потребує конфігурації	Гарна підтримка	Нативна підтримка	2,4,4,2,3,4
Автоматичне об'єднання гілок	Обмежена підтримка	Нативна підтримка	Нативна підтримка	Обмежена підтримка	Гарна підтримка	Гарна підтримка	1,4,4,1,3,3
Розширені можливості							
Мультирепозиторні збірки	Гарна підтримка	Гарна підтримка	Потребує конфігурації	Гарна підтримка	Потребує конфігурації	Гарна підтримка	3,3,2,3,2,3
Feature Flag інтеграція	Обмежена підтримка	Нативна підтримка	Потребує конфігурації	Обмежена підтримка	Потребує конфігурації	Гарна підтримка	1,4,2,1,2,3
Управління версіями реплів	Потребує конфігурації	Нативна підтримка	Гарна підтримка	Потребує конфігурації	Гарна підтримка	Нативна підтримка	2,4,3,2,3,4
Канали розгортання (CD Pipelines)	Потребує конфігурації	Нативна підтримка	Гарна підтримка	Потребує конфігурації	Гарна підтримка	Нативна підтримка	2,4,3,2,3,4
Загальна оцінка							
Оцінка основних стратегій	2.3	3.8	3.5	2.3	3.3	3.0	2,3,3,8,3,5,2,3,3,3,3,0
Оцінка специфічних особливостей	2.3	4.0	3.8	2.5	3.8	3.8	2,3,4,0,3,8,2,5,3,8,3,8
Оцінка автоматизації та правил	1.5	4.0	4.0	1.5	3.0	3.8	1,5,4,0,4,0,1,5,3,0,3,8
Оцінка розширених можливостей	2.0	3.8	2.5	2.0	2.5	3.5	2,0,3,8,2,5,2,0,2,5,3,5
ЗАГАЛЬНА ОЦІНКА	2.0	3.9	3.4	2.1	3.1	3.5	2,0,3,9,3,4,2,1,3,1,3,5

Рисунок 2.8 – Підтримка стратегій розгалуження коду CI/CD-інструментами

Інфраструктура як код (IaC) дозволяє застосувати ті ж практики контролю версій, тестування та автоматизації до інфраструктури, що і до програмного коду. Інтеграція IaC з CI/CD є ключовим аспектом сучасної DevOps-практики [31, с. 4]. В таблиці 2.5 GitLab CI/CD та Azure DevOps демонструють найкращу нативну підтримку різних інструментів IaC з детальною інтеграцією та вбудованими завданнями. GitHub Actions, завдяки своєму маркетплейсу, також пропонує широкий вибір готових дій для управління інфраструктурою.

Таблиця 2.5 – Порівняння підтримки інструментів IaC у CI/CD-системах

Інструмен	Terraform	Ansible	CloudFormation	Kubernetes	Helm
-----------	-----------	---------	----------------	------------	------

T				YAML	charts
Jenkins	Через shell	Через shell	Через AWS plugin	Через kubectl	Через shell
GitLab CI/CD	Вбудована підтримка	Через runner	Через AWS integration	Вбудована підтримка	Вбудована підтримка
GitHub Actions	Actions marketplace	Actions marketplace	AWS Actions	Actions marketplace	Actions marketplace
TeamCity	Через shell	Через shell	Через custom steps	Через shell	Через shell
CircleCI	Через orbs	Через orbs	Через orbs	Так, через API	Через orbs
Azure DevOps	Terraform task	Ansible task	ARM templates	Так, через Service Hooks	Helm task

Валідація та тестування інфраструктурних змін є важливою частиною CI/CD для забезпечення стабільності та безпеки. GitLab CI/CD пропонує найбільш комплексний підхід до тестування інфраструктури з вбудованими інструментами для статичного аналізу, валідації планів та тестування після розгортання. Jenkins та інші інструменти також здатні підтримувати ці практики, але потребують додаткового налаштування (див. рисунок 2.9).

4 - Вбудована підтримка 3 - Гарна підтримка через інтеграції 2 - Підтримка через плагіни/додаткове налаштування 1 - Обмежена підтримка

Підхід до тестування інфраструктури	Jenkins	GitLab CI/CD	GitHub Actions	TeamCity	CircleCI	Azure DevOps	Числове значення
Статичний аналіз IaC							
Terraform валідація	Через плагіни	Вбудована	Через інтеграції	Через плагіни	Через плагіни	Через інтеграції	2,4,3,2,2,3
CloudFormation/ARM валідація	Через плагіни	Через інтеграції	Через інтеграції	Обмежена	Через плагіни	Вбудована	2,3,3,1,2,4
Kubernetes YAML валідація	Через плагіни	Вбудована	Через інтеграції	Через плагіни	Через плагіни	Через інтеграції	2,4,3,2,2,3
Аналіз безпеки IaC	Через плагіни	Вбудована	Через інтеграції	Обмежена	Через плагіни	Через інтеграції	2,4,3,1,2,3
Lint-перевірка IaC файлів	Через плагіни	Вбудована	Через інтеграції	Через плагіни	Через інтеграції	Через інтеграції	2,4,3,2,3,3
Тестування перед розгортанням							
Dry-run для Terraform	Через плагіни	Вбудована	Через інтеграції	Через плагіни	Через плагіни	Через інтеграції	2,4,3,2,2,3
Preview змін для CloudFormation/ARM	Через плагіни	Через інтеграції	Через інтеграції	Обмежена	Через плагіни	Вбудована	2,3,3,1,2,4
Kubernetes Dry-run	Через плагіни	Вбудована	Через інтеграції	Через плагіни	Через плагіни	Через інтеграції	2,4,3,2,2,3
Інфраструктурні Unit-тести	Через плагіни	Через інтеграції	Через інтеграції	Через плагіни	Через плагіни	Через інтеграції	2,3,3,2,2,3
Інтеграційне тестування на стейджинг	Через плагіни	Вбудована	Через інтеграції	Через плагіни	Через інтеграції	Вбудована	2,4,3,2,3,4
Тестування після розгортання							
Валідація розгорнутої інфраструктури	Через плагіни	Вбудована	Через інтеграції	Через плагіни	Через плагіни	Через інтеграції	2,4,3,2,2,3
Перевірка відповідності стандартам	Через плагіни	Вбудована	Через плагіни	Обмежена	Через плагіни	Через інтеграції	2,4,2,1,2,3
Моніторинг після розгортання	Через плагіни	Вбудована	Через плагіни	Через плагіни	Через інтеграції	Вбудована	2,4,2,2,3,4
Smoke-тести інфраструктури	Через плагіни	Вбудована	Через інтеграції	Через плагіни	Через інтеграції	Через інтеграції	2,4,3,2,3,3
Автоматичний rollback при помилках	Через плагіни	Вбудована	Через інтеграції	Обмежена	Через інтеграції	Вбудована	2,4,3,1,3,4
Спеціалізовані інструменти тестування							
Підтримка Terratest	Через плагіни	Через інтеграції	Через інтеграції	Через плагіни	Через плагіни	Через інтеграції	2,3,3,2,2,3
Підтримка InSpec	Через плагіни	Через інтеграції	Через інтеграції	Обмежена	Через плагіни	Через інтеграції	2,3,3,1,2,3
Інтеграція з Chaos Engineering	Обмежена	Через інтеграції	Через плагіни	Обмежена	Через плагіни	Через інтеграції	1,3,2,1,2,3
Симуляція навантаження на інфраструктуру	Через плагіни	Через інтеграції	Через плагіни	Через плагіни	Через інтеграції	Через інтеграції	2,3,2,2,3,3
Перевірка безпеки розгорнутої інфраструктури	Через плагіни	Вбудована	Через інтеграції	Обмежена	Через плагіни	Вбудована	2,4,3,1,2,4
Загальна оцінка							
Оцінка статичного аналізу IaC	2.0	3.8	3.0	1.6	2.2	3.2	2,0,3,8,3,0,1,6,2,2,3,2
Оцінка тестування перед розгортанням	2.0	3.6	3.0	1.8	2.2	3.4	2,0,3,6,3,0,1,8,2,2,3,4
Оцінка тестування після розгортання	2.0	4.0	2.6	1.6	2.6	3.4	2,0,4,0,2,6,1,6,2,6,3,4
Оцінка спеціалізованих інструментів	1.8	3.2	2.6	1.4	2.2	3.2	1,8,3,2,2,6,1,4,2,2,3,2
ЗАГАЛЬНА ОЦІНКА	2.0	3.7	2.8	1.6	2.3	3.3	2,0,3,7,2,8,1,6,2,3,3,3

Рисунок 2.9 – Порівняння підходів до тестування інфраструктури в CI/CD-процесах

Автоматизоване тестування є ключовим компонентом CI/CD, що забезпечує якість та надійність програмного забезпечення при кожній зміні. Комплексний CI/CD-процес має підтримувати різні рівні тестування, від модульних до інтеграційних та «end-to-end» тестів. В данному випадку GitLab CI/CD виділяється комплексним підходом до тестування з вбудованою підтримкою різних рівнів тестів, включаючи динамічний аналіз безпеки та моніторинг продуктивності (див. у таблиці 2.6). GitHub Actions, завдяки своєму маркетплейсу, також пропонує широкий вибір інтеграцій з інструментами тестування [32].

Таблиця 2.6 – Порівняння підтримки різних рівнів тестування у CI/CD-інструментах

Інструмент	Модульні тести	Інтеграційні тести	UI/Функціональні тести	Навантажувальні тести	Безпечі тести
Jenkins	Через плагіни	Через плагіни	Selenium, Cypress через плагіни	JMeter, Gatling через плагіни	OWASP ZAP через плагіни
GitLab CI/CD	Вбудована підтримка	Вбудована підтримка	Вбудований Browser Performance Testing	k6 інтеграція	Вбудовані DAST, SAST, dependency scanning
GitHub Actions	Actions marketplace	Actions marketplace	Playwright, Cypress actions	k6, Locust actions	CodeQL, GHAS інтеграція
TeamCity	Вбудована підтримка	Через плагіни	Через плагіни	Через плагіни	Через плагіни
CircleCI	Вбудована підтримка	Orbs ecosystem	Cypress, Playwright orbs	k6, Artillery orbs	CircleCI security orbs
Azure DevOps	Вбудована підтримка	Вбудована підтримка	Selenium, Cypress tasks	Azure Load Testing	Microsoft Security Code Analysis

Ефективний CI/CD-процес вимагає плавної інтеграції усіх компонентів у єдиний цілісний конвеєр, що забезпечує автоматизований шлях від коду до продуктивного середовища. Різні CI/CD-інструменти пропонують різні підходи до структурування пайплайнів. GitLab CI/CD та GitHub Actions пропонують найбільш гнучкі та модульні моделі організації пайплайнів з підтримкою шаблонів, умовного виконання та багаторівневої структури. Jenkins, хоч і має потужний механізм Shared Libraries, може бути більш складним у налаштуванні та підтримці (див. у таблиці 2.7).

Таблиця 2.7 – Порівняння моделей організації CI/CD-пайплайнів

Інструмент	Модель пайплайну	Повторне використання коду	Модульність	Умовне виконання етапів
Jenkins	Declarative / Scripted Pipelines	Shared Libraries	Pipeline as Code	Declarative when/conditions
GitLab CI/CD	YAML-based stages/jobs	include/extends/templates	Modular .gitlab-ci.yml	rules/only/except expressions
GitHub Actions	YAML-based workflows/jobs	Reusable workflows/composite actions	Modular workflow files	if conditions
TeamCity	Build Configurations / Chain	Kotlin DSL shared code	Snapshot dependencies	Failure Conditions
CircleCI	YAML-based workflows/jobs	orbs / commands	Modular config	when expressions
Azure DevOps	YAML-based stages/jobs	templates / variables	Task groups / templates	conditions

Ефективний моніторинг та візуалізація CI/CD-процесів є ключовими для забезпечення прозорості та швидкого виявлення проблем. На рисунку 2.10 зображено, що GitLab CI/CD та Azure DevOps пропонують найбільш комплексні можливості для моніторингу та візуалізації CI/CD-процесів з детальними метриками, аналітикою цінності (Value Stream Analytics) та інтеграцією з різними каналами сповіщень.

4 - Вбудована розширена підтримка 3 - Вбудована базова підтримка 2 - Підтримка через плагіни/інтеграції 1 - Обмежена підтримка

Функціональність моніторингу та візуалізації	Jenkins	GitLab CI/CD	GitHub Actions	TeamCity	CircleCI	Azure DevOps	Число значення
Базовий моніторинг CI/CD-процесів							
Дашбورد статусу збірок	Вбудована базова	Вбудована розширена	Вбудована базова	Вбудована базова	Вбудована базова	Вбудована розширена	3,4,3,3,4
Моніторинг часу виконання	Через плагіни	Вбудована розширена	Вбудована базова	Вбудована базова	Вбудована базова	Вбудована розширена	2,4,3,3,4
Візуалізація pipeline	Через плагіни	Вбудована розширена	Вбудована базова	Вбудована базова	Вбудована базова	Вбудована розширена	2,4,3,3,4
Історія виконання	Вбудована базова	Вбудована розширена	Вбудована базова	Вбудована базова	Вбудована базова	Вбудована розширена	3,4,3,3,4
Сповіщення про статус	Вбудована базова	Вбудована розширена	Вбудована базова	Вбудована базова	Вбудована базова	Вбудована розширена	3,4,3,3,4
Розширена аналітика та метрики							
Value Stream Analytics	Обмежена	Вбудована розширена	Через плагіни	Обмежена	Вбудована базова	Вбудована розширена	1,4,2,1,3,4
Метрики продуктивності команди	Обмежена	Вбудована розширена	Через плагіни	Через плагіни	Вбудована базова	Вбудована розширена	1,4,2,2,3,4
Аналіз вузьких місць	Обмежена	Вбудована розширена	Через плагіни	Обмежена	Вбудована базова	Вбудована розширена	1,4,2,1,3,4
Звітність щодо якості коду	Через плагіни	Вбудована розширена	Вбудована базова	Через плагіни	Вбудована базова	Вбудована розширена	2,4,3,2,3,4
Аналіз тенденцій	Через плагіни	Вбудована розширена	Через плагіни	Через плагіни	Вбудована базова	Вбудована розширена	2,4,2,2,3,4
Інтеграція з моніторинговими системами							
Prometheus/Grafana інтеграція	Через плагіни	Вбудована розширена	Через плагіни	Через плагіни	Вбудована базова	Вбудована базова	2,4,2,2,3,3
ELK Stack інтеграція	Через плагіни	Вбудована базова	Через плагіни	Через плагіни	Через плагіни	Вбудована базова	2,3,2,2,2,3
Datadog/New Relic інтеграція	Через плагіни	Вбудована базова	Через плагіни	Через плагіни	Вбудована розширена	Вбудована базова	2,3,2,2,4,3
APM інтеграція	Через плагіни	Вбудована базова	Через плагіни	Через плагіни	Вбудована базова	Вбудована розширена	2,3,2,2,3,4
Custom метрики та моніторинг	Через плагіни	Вбудована розширена	Через плагіни	Через плагіни	Вбудована базова	Вбудована розширена	2,4,2,2,3,4
Сповіщення та комунікації							
Email сповіщення	Вбудована базова	Вбудована розширена	Вбудована базова	Вбудована базова	Вбудована базова	Вбудована розширена	3,4,3,3,3,4
Slack/Microsoft Teams інтеграція	Через плагіни	Вбудована розширена	Вбудована базова	Через плагіни	Вбудована розширена	Вбудована розширена	2,4,3,2,4,4
Webhook сповіщення	Вбудована базова	Вбудована розширена	Вбудована базова	Вбудована базова	Вбудована розширена	Вбудована розширена	3,4,3,3,4,4
Налаштовувані правила сповіщень	Через плагіни	Вбудована розширена	Вбудована базова	Через плагіни	Вбудована базова	Вбудована розширена	2,4,3,2,3,4
Інтеграція з системами відстеження помилок	Через плагіни	Вбудована розширена	Вбудована базова	Через плагіни	Вбудована базова	Вбудована розширена	2,4,3,2,3,4
Розширені функції візуалізації							
Налаштовувані дашборди	Через плагіни	Вбудована розширена	Через плагіни	Через плагіни	Вбудована базова	Вбудована розширена	2,4,2,2,3,4
DevOps-метрики в реальному часі	Обмежена	Вбудована розширена	Через плагіни	Обмежена	Вбудована базова	Вбудована розширена	1,4,2,1,3,4
Візуалізація залежностей	Через плагіни	Вбудована розширена	Через плагіни	Через плагіни	Через плагіни	Вбудована базова	2,4,2,2,2,3
Експорт звітів	Через плагіни	Вбудована базова	Через плагіни	Через плагіни	Вбудована базова	Вбудована розширена	2,3,2,2,3,4
Розширені графіки та діаграми	Через плагіни	Вбудована розширена	Через плагіни	Через плагіни	Вбудована базова	Вбудована розширена	2,4,2,2,3,4
Загальна оцінка							
Оцінка базового моніторингу	2.6	4.0	3.0	3.0	3.0	4.0	2,6,4,0,3,0,3,0,4,0
Оцінка розширеної аналітики	1.4	4.0	2.2	1.6	3.0	4.0	1,4,4,0,2,2,1,6,3,0,4,0
Оцінка інтеграції з моніторингом	2.0	3.4	2.0	2.0	3.0	3.4	2,0,3,4,2,0,2,0,3,0,3,4
Оцінка сповіщень	2.4	4.0	3.0	2.4	3.4	4.0	2,4,4,0,3,0,2,4,3,4,4,0
Оцінка розширеної візуалізації	1.8	3.8	2.0	1.8	2.8	3.8	1,8,3,8,2,0,1,8,2,8,3,8
ЗАГАЛЬНА ОЦІНКА	2.0	3.8	2.4	2.2	3.0	3.8	2,0,3,8,2,4,2,2,3,0,3,8

Рисунок 2.10 – Порівняння можливостей моніторингу та візуалізації CI/CD-процесів

На основі проведеного аналізу методів інтеграції різних компонентів CI/CD можна зробити наступні висновки:

1. GitLab CI/CD демонструє найбільш комплексний та інтегрований підхід до об'єднання системи контролю версій, інфраструктури як код та авто-тестування у єдиний конвеєр. Його тісна інтеграція з GitLab репозиторіями, вбудовані інструменти безпеки та можливості організації мульти-проектних пайплайнів роблять його особливо ефективним для складних розподілених систем.
2. GitHub Actions завдяки своєму маркетплейсу та тісній інтеграції з GitHub забезпечує високу гнучкість та легкість налаштування, особливо для проєктів з відкритим кодом та невеликих команд.
3. Jenkins залишається потужним і гнучким інструментом з безліччю можливостей розширення, але вимагає більше зусиль для забезпечення такого ж рівня інтеграції, як у новіших платформах.
4. Azure DevOps завдяки своїй інтеграції з екосистемою Microsoft та вбудованим можливостям для роботи з Azure пропонує потужне рішення для організацій, що вже використовують продукти Microsoft.
5. TeamCity від JetBrains пропонує потужні можливості для побудови CI/CD конвеєрів з акцентом на глибоку інтеграцію з іншими інструментами розробки JetBrains. TeamCity добре підходить для великих корпоративних середовищ завдяки розширеним можливостям управління дозволами та масштабування через агентів збірки.
6. CircleCI пропонує гнучку систему оркестрації робочих процесів та ефективну інтеграцію з контейнерними технологіями, що робить його особливо привабливим для команд, що використовують мікросервісну архітектуру та платформи, орієнтовані на хмарні технології.

2.4 Вибір оптимального підходу до CI/CD для розподіленого середовища

Розробка оптимального підходу до CI/CD для розподіленого середовища вимагає врахування ряду факторів: масштабу системи, технологічного стеку та бізнес-вимог. Правильно підібрані інструменти та методології дозволяють створити ефективну систему безперервної інтеграції та доставки, що відповідає конкретним потребам проєкту.

При виборі CI/CD-підходу для розподіленого середовища необхідно враховувати ряд критеріїв, що визначають його ефективність та відповідність потребам проєкту. Розподілені системи часто характеризуються високим навантаженням на CI/CD-інфраструктуру через велику кількість мікросервісів, компонентів та команд розробників. Оцінюючи масштабованість CI/CD-рішень, необхідно враховувати такі критерії як здатність до горизонтального масштабування (додавання виконавчих агентів/раннерів для обробки більшої кількості завдань), можливість паралельного виконання багатьох завдань для різних компонентів, ефективність використання обчислювальних ресурсів, сталість продуктивності при зростанні навантаження, а також підтримку географічного розподілу агентів у різних регіонах для зниження латентності. На основі цих критеріїв GitLab CI/CD з Kubernetes runners та CircleCI демонструють найкращі показники масштабованості для великих розподілених систем, забезпечуючи ефективне горизонтальне масштабування та розподіл навантаження (див. у таблиці 2.8).

Таблиця 2.8 – Критерії оцінки масштабованості CI/CD-рішень для розподілених систем

Критерій	Опис	Значимість
Горизонтальне масштабування	Здатність додавати виконавчі агенти/раннери для обробки більшої кількості завдань	Висока
Паралельне виконання	Можливість одночасного запуску багатьох завдань для різних компонентів	Висока

Продовження таблиці 2.8

Ефективність ресурсів	Оптимальне використання обчислювальних ресурсів під час виконання пайплайнів	Середня
Сталість продуктивності	Передбачуваний час виконання пайплайнів навіть при зростанні навантаження	Середня
Підтримка географічного розподілу	Можливість розташування агентів у різних регіонах для зниження латентності	Висока для глобальних команд

Сучасні розподілені системи часто базуються на мікросервісній архітектурі, що вимагає специфічних підходів до CI/CD. На рисунку 2.11 зображено, що GitLab CI/CD та Jenkins X (спеціалізована версія Jenkins для Kubernetes) [33] демонструють найкращу підтримку мікросервісних архітектур з моделями розгортання, оптимізованими для незалежних мікросервісів, управління залежностями між сервісами та специфічних стратегій розгортання.

* Оцінка для Jenkins включає можливості Jenkins X, що є спеціалізованою версією для Kubernetes

4 - Відмінна підтримка 3 - Хороша підтримка 2 - Помірна підтримка 1 - Обмежена підтримка

Функціональність	Jenkins*	GitLab CI/CD	GitHub Actions	TeamCity	CircleCI	Azure DevOps	Числове значення
Базова підтримка мікросервісів							
Незалежне управління окремими сервісами	Відмінна	Відмінна	Хороша	Помірна	Хороша	Хороша	4,4,3,2,3,3
Паралельне розгортання сервісів	Відмінна	Відмінна	Хороша	Помірна	Відмінна	Хороша	4,4,3,2,4,3
Управління залежностями між сервісами	Відмінна	Відмінна	Помірна	Обмежена	Помірна	Хороша	4,4,2,1,2,3
Підтримка поліглотного середовища	Хороша	Відмінна	Хороша	Хороша	Хороша	Хороша	3,4,3,3,3,3
Управління версіями API	Помірна	Відмінна	Помірна	Обмежена	Помірна	Хороша	2,4,2,1,2,3
Контейнеризація та оркестрація							
Інтеграція з Docker	Відмінна	Відмінна	Відмінна	Хороша	Відмінна	Відмінна	4,4,4,3,4,4
Інтеграція з Kubernetes	Відмінна	Відмінна	Хороша	Помірна	Відмінна	Хороша	4,4,3,2,4,3
Підтримка Helm Charts	Відмінна	Відмінна	Хороша	Помірна	Хороша	Хороша	4,4,3,2,3,3
Service Mesh інтеграція	Хороша	Відмінна	Помірна	Обмежена	Помірна	Хороша	3,4,2,1,2,3
Serverless підтримка	Хороша	Відмінна	Відмінна	Помірна	Хороша	Відмінна	3,4,4,2,3,4
Стратегії розгортання							
Canary Deployments	Відмінна	Відмінна	Хороша	Помірна	Хороша	Відмінна	4,4,3,2,3,4
Blue-Green Deployments	Відмінна	Відмінна	Хороша	Помірна	Відмінна	Відмінна	4,4,3,2,4,4
Feature Flags	Хороша	Відмінна	Помірна	Обмежена	Помірна	Відмінна	3,4,2,1,2,4
A/B Testing	Помірна	Хороша	Помірна	Обмежена	Помірна	Хороша	2,3,2,1,2,3
Автоматичний Rollback	Відмінна	Відмінна	Хороша	Помірна	Відмінна	Відмінна	4,4,3,2,4,4
Тестування та моніторинг мікросервісів							
Інтеграційне тестування між сервісами	Хороша	Відмінна	Хороша	Помірна	Хороша	Хороша	3,4,3,2,3,3
Contract Testing	Хороша	Відмінна	Помірна	Обмежена	Помірна	Хороша	3,4,2,1,2,3
Chaos Engineering	Хороша	Хороша	Помірна	Обмежена	Помірна	Помірна	3,3,2,1,2,2
Розподілений трейсінг	Помірна	Відмінна	Помірна	Обмежена	Хороша	Хороша	2,4,2,1,3,3
Агрегація логів мікросервісів	Хороша	Відмінна	Помірна	Помірна	Хороша	Відмінна	3,4,2,2,3,4
DevOps-практики для мікросервісів							
Інфраструктура як код (IaC)	Відмінна	Відмінна	Хороша	Помірна	Хороша	Відмінна	4,4,3,2,3,4
GitOps підтримка	Відмінна	Відмінна	Хороша	Обмежена	Хороша	Хороша	4,4,3,1,3,3
Автоматичне масштабування	Хороша	Відмінна	Помірна	Обмежена	Хороша	Хороша	3,4,2,1,3,3
Управління секретами	Хороша	Відмінна	Хороша	Помірна	Хороша	Відмінна	3,4,3,2,3,4
Автоматизація мережесхем політик	Помірна	Хороша	Помірна	Обмежена	Помірна	Хороша	2,3,2,1,2,3
Загальна оцінка							
Оцінка базової підтримки	3.4	4.0	2.6	1.8	2.8	3.0	3.4,4.0,2.6,1.8,2.8,3.0
Оцінка контейнеризації та оркестрації	3.6	4.0	3.2	2.0	3.2	3.4	3.6,4.0,3.2,2.0,3.2,3.4
Оцінка стратегій розгортання	3.4	4.0	2.6	1.6	3.0	3.8	3.4,4.0,2.6,1.6,3.0,3.8
Оцінка тестування та моніторингу	2.8	3.8	2.2	1.4	2.6	3.0	2.8,3.8,2.2,1.4,2.6,3.0
Оцінка DevOps-практик	3.2	3.8	2.6	1.4	2.8	3.4	3.2,3.8,2.6,1.4,2.8,3.4
ЗАГАЛЬНА ОЦІНКА	3.3	3.9	2.6	1.6	2.9	3.3	3.3,3.9,2.6,1.6,2.9,3.3

Рисунок 2.11 – Порівняння підтримки мікросервісних архітектур CI/CD-інструментами

На основі отриманих результатів можна сформулювати рекомендації щодо вибору інструментів CI/CD для різних типів розподілених систем (див. у таблиці 2.9). Для мікросервісної архітектури на Kubernetes найбільш

ефективним є поєднання GitLab CI/CD для безперервної інтеграції з Argo CD для безперервного розгортання, що забезпечує декларативний GitOps-підхід з автоматичною синхронізацією стану кластера з бажаним станом у Git.

Таблиця 2.9 – Рекомендації щодо вибору інструментів CI/CD для різних типів систем

Тип розподіленої системи	Рекомендовані CI/CD-інструменти	Особливості реалізації
Горизонтальне масштабування	GitLab CI/CD, Argo CD, Tekton	Розділення CI та CD, використання GitOps-підходу, автоматичне масштабування раннерів
Паралельне виконання	Jenkins, Azure DevOps	Централізовані пайплайни з детальним workflow, інтеграція з системами керування релізами
Ефективність ресурсів	GitLab CI/CD, Jenkins	Диференційований підхід з єдиним контролем, окремі пайплайни для різних архітектурних стилів
Сталість продуктивності	GitLab CI/CD з Geo, Azure DevOps	Розподілені агенти, локальні кеші, асинхронні нотифікації
Підтримка географічного розподілу	Azure DevOps, GitLab Enterprise	Комплексний аудит, строгий контроль доступу, інтеграція з системами compliance
Високонавантажені системи з частими оновленнями	CircleCI, GitLab CI/CD з Kubernetes	Оптимізація для паралельного виконання, кешування залежностей, ефективне використання ресурсів

Для складних розподілених систем оптимальним часто є не один інструмент, а комбінація різних інструментів, що забезпечують окремі аспекти CI/CD-процесу. У такому комплексному підході:

- GitLab CI/CD використовується для безперервної інтеграції, модульного та інтеграційного тестування, аналізу якості коду та безпеки, а також збирання артефактів:
- Argo CD забезпечує GitOps-орієнтоване безперервне розгортання в Kubernetes-кластері, автоматичну синхронізацію стану кластера з

конфігурацією в Git

- Terraform через GitLab CI/CD використовується для управління інфраструктурою як кодом
- Prometheus і Grafana забезпечують моніторинг та візуалізацію метрик системи та CI/CD-процесів
- HashiCorp Vault інтегрується з GitLab CI/CD для безпечного управління секретами та креденціалами

Такий комплексний підхід забезпечує оптимальний баланс продуктивності, безпеки та гнучкості для складних розподілених систем, дозволяючи кожному інструменту фокусуватися на своїй спеціалізації.

Найбільш оптимальним підходом для безперервної інтеграції та доставки в розподілених системах на основі мікросервісної архітектури є використання GitLab CI/CD у поєднанні з Kubernetes. Оптимальна архітектура GitLab CI/CD для розподілених систем базується на використанні Kubernetes runners, що забезпечують масштабованість, ізоляцію та ефективне використання ресурсів. Ключовими компонентами цієї архітектури є:

- GitLab CI/CD Server - головний центр керування, який координує весь процес безперервної інтеграції та доставки, обробляє всі запити та налаштування піпелайнів.
- Kubernetes Cluster - оркестраційна платформа, яка забезпечує автоматичне масштабування, розгортання та керування контейнеризованими додатками та CI/CD-середовищами.
- GitLab Runner Manager - компонент, що відповідає за створення, керування та моніторинг GitLab Runner-ів, які виконують окремі завдання CI/CD-процесів.
- CI/CD Pods - ізольовані контейнери, в яких безпосередньо виконуються задачі тестування, збірки та інші етапи CI/CD-піпелайну.
- Shared Caches - спільні сховища для тимчасового зберігання

залежностей, артефактів та проміжних результатів, що значно прискорюють процеси збірки та тестування.

- Container Registry - централізований репозиторій для зберігання, управління та розповсюдження Docker-образів, які використовуються та створюються в процесі CI/CD.

Така архітектура забезпечує високу масштабованість, ефективне використання ресурсів та швидке виконання CI/CD-процесів для розподілених систем.

Для розподілених систем критично важливим є вибір правильних стратегій розгортання, що забезпечують мінімальний час простою та можливість швидкого відкату у разі проблем. Для розподілених систем із мікросервісною архітектурою оптимальним є комбінований підхід, де різні стратегії (Rolling Update, Blue-Green, Canary, Feature Flags) застосовуються для різних типів сервісів залежно від їх критичності, частоти оновлень та ризику змін [34].

Після впровадження CI/CD-підходу важливо постійно оцінювати його ефективність та оптимізувати процеси для підвищення продуктивності та зниження витрат. Для оцінки ефективності CI/CD-процесів використовують ряд ключових метрик: частота розгортання, час від коміту до розгортання, частота відмов при розгортанні, час відновлення після збоїв, тривалість CI/CD-пайплайну, покриття коду тестами, витрати ресурсів на CI/CD (див. у таблиці 2.10). Постійний моніторинг цих метрик дозволяє виявляти проблеми та вузькі місця в CI/CD-процесах та оперативно їх усувати, забезпечуючи постійне підвищення ефективності [6, с. 13].

Таблиця 2.10 – Ключові метрики ефективності CI/CD-процесів

Метрика	Опис	Цільові значення	Вплив на бізнес
Частота розгортання	GitLab CI/CD, Argo CD, Tekton	Розділення CI та CD, використання GitOps-підходу, автоматичне масштабування раннерів	Швидкість доставки нового функціоналу
Час від коміту до	Jenkins, Azure	Централізовані пайплайни з	Швидкість реакції на

розгортання	DevOps	детальним workflow, інтеграція з системами керування релізами	зміни
-------------	--------	---	-------

Продовження таблиці 2.10

Частота відмов при розгортанні	GitLab CI/CD, Jenkins	Диференційований підхід з єдиним контролем, окремі пайплайни для різних архітектурних стилів	Стабільність та надійність системи
Час відновлення після збоїв	GitLab CI/CD з Geo, Azure DevOps	Розподілені агенти, локальні кеші, асинхронні нотифікації	Стійкість системи до збоїв
Тривалість CI/CD-пайплайну	Azure DevOps, GitLab Enterprise	Комплексний аудит, строгий контроль доступу, інтеграція з системами compliance	Продуктивність розробників
Покриття коду тестами	CircleCI, GitLab CI/CD з Kubernetes	Оптимізація для паралельного виконання, кешування залежностей, ефективне використання ресурсів	Якість коду та запобігання дефектам
Витрати ресурсів на CI/CD	Обчислювальні ресурси та витрати, пов'язані з CI/CD-процесами	Оптимізація залежно від специфіки проєкту	Економічна ефективність

Для підвищення ефективності CI/CD-процесів у розподілених системах також застосовують ряд практик оптимізації: кешування залежностей, розпаралелювання завдань, інкрементальне збирання, оптимізація тестових наборів, використання спеціалізованих образів, ефективне розподілення ресурсів, автоматичне масштабування інфраструктури. Застосування цих практик дозволяє значно підвищити ефективність CI/CD-процесів, зменшити час виконання пайплайнів та оптимізувати використання ресурсів.

Для ефективного управління CI/CD-процесами також необхідно впровадити комплексну систему моніторингу та відстеження продуктивності, що дозволяє оперативно виявляти проблеми та оптимізувати процеси. Інтеграція системам моніторингу (Prometheus, Grafana), які ми описували

раніше та сповіщення (Slack, email) забезпечує постійний контроль за CI/CD-процесами та можливість оперативного реагування на проблеми.

На основі проведеного аналізу можна зробити наступні висновки щодо вибору оптимального підходу до CI/CD для розподілених систем:

1. GitLab CI/CD у поєднанні з Kubernetes є найбільш ефективним підходом для сучасних розподілених систем з мікросервісною архітектурою, забезпечуючи масштабованість, гнучкість та комплексну автоматизацію.
2. Структуровані пайплайни з використанням шаблонів та багаторівневою організацією дозволяють ефективно управляти складністю розподілених систем, забезпечуючи автономію окремих команд при збереженні цілісності системи.
3. Комбінований підхід до стратегій розгортання з використанням різних стратегій для різних типів сервісів забезпечує оптимальний баланс між надійністю, швидкістю та ефективністю розгортання.
4. Постійний моніторинг та оптимізація CI/CD-процесів на основі ключових метрик є необхідною умовою для підтримки високої ефективності та економічної доцільності CI/CD-підходу.

Впровадження такого підходу дозволяє досягти високих показників продуктивності, якості та надійності розподілених систем, забезпечуючи швидку доставку нових функцій та ефективну взаємодію між географічно розподіленими командами.

Результати проведеного аналізу створюють основу для практичної реалізації ефективної CI/CD-системи для розподіленого веб-застосунку, що буде розглянуто в наступному розділі. Застосування GitLab CI/CD з Kubernetes для організації пайплайнів збирання, тестування та розгортання забезпечить оптимальний баланс між продуктивністю, гнучкістю та безпекою, дозволяючи досягти високих показників ефективності та якості процесу розробки.

РОДІЛ 3

ПРАКТИЧНЕ ВПРОВАДЖЕННЯ CI/CD У РОЗПОДІЛЕНІЙ СИСТЕМІ

3.1 Опис веб-застосунку та його архітектури

У цьому розділі розглянемо практичне впровадження процесів CI/CD у розподіленій системі на прикладі проєкту Spring PetClinic Microservices. Цей проєкт — модифікація класичного демонстраційного застосунку Spring PetClinic, що є еталонним прикладом для навчання та експериментів із Spring Boot, який було переведено з монолітної на мікросервісну архітектуру [35].

Spring PetClinic Microservices — це веб-застосунок для управління ветеринарною клінікою, що дозволяє зберігати інформацію про власників домашніх тварин, їхніх улюбленців та візити до ветеринара. Проєкт обрано через його оптимальний рівень складності: застосунок є достатньо комплексним для демонстрації переваг CI/CD, але водночас не настільки об'ємним, щоб ускладнювати розуміння основних принципів. Як було зазначено у другому розділі, Spring PetClinic часто використовується як зразок для тестування та навчання.

Загальна архітектура проєкту представлена на рисунку 3.1 і базується на сучасних принципах розробки розподілених систем. Застосунок розділено на п'ять ключових мікросервісів, що взаємодіють між собою через API-шлюз. Це дозволяє кожному сервісу розвиватися незалежно, мати власну базу даних та

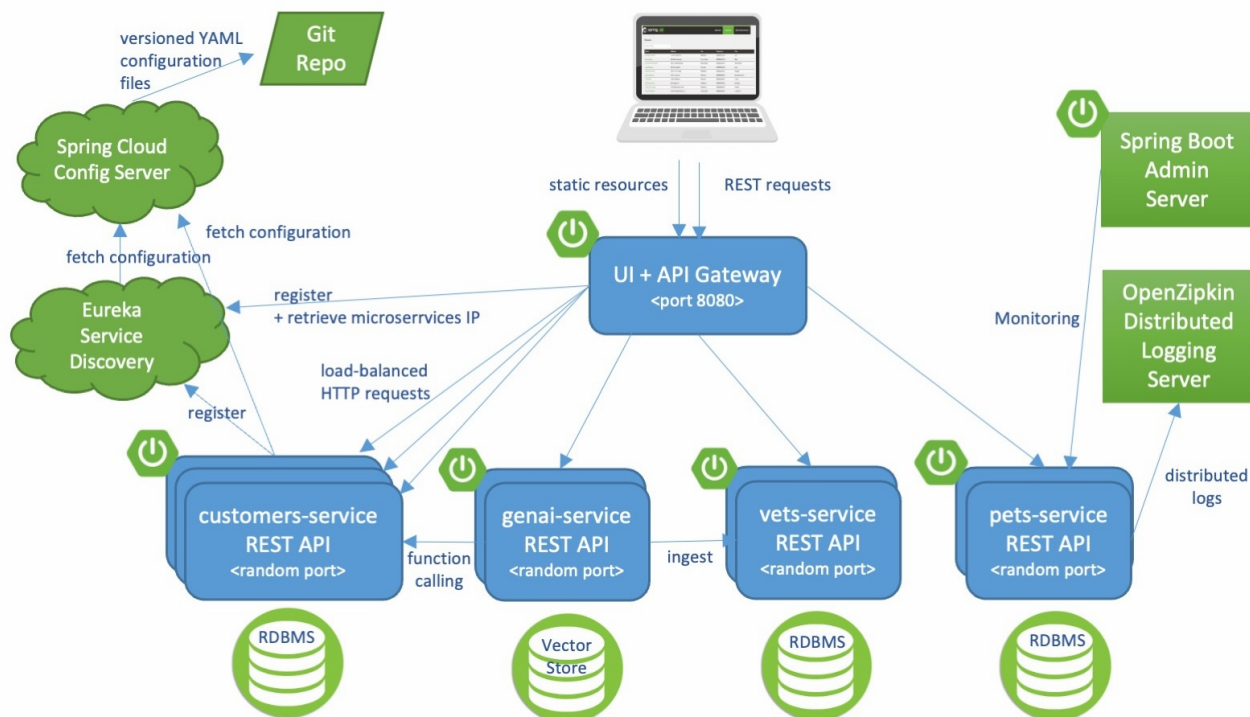
Рисунок 3.1 – Архітектура Spring PetClinic Microservices [35]

масштабуватися

відповідно

до

потреб.



Компоненти системи реалізовано такими мікросервісами [37]:

1. API Gateway (порт 8080) — вхідна точка для всіх клієнтських запитів, реалізована на базі Spring Cloud Gateway. Відповідає за маршрутизацію запитів до відповідних сервісів, балансування навантаження та базову авторизацію. Цей сервіс забезпечує єдиний інтерфейс для взаємодії з усією системою, приховуючи складність мікросервісної архітектури від кінцевого користувача.
2. Config Server (порт 8888) — централізований сервер конфігурації, заснований на Spring Cloud Config, що зберігає налаштування для всіх мікросервісів у Git-репозиторії. Це дозволяє змінювати конфігурації без перезапуску сервісів та забезпечує узгодженість налаштувань у різних середовищах (розробка, тестування, продакшн).
3. Discovery Service (порт 8761) — реалізує службу виявлення на базі Netflix Eureka, що дозволяє мікросервісам реєструватися та знаходити один одного без жорсткої прив'язки до конкретних

IP-адрес чи портів. Це критично важливо для динамічного масштабування та стійкості системи до змін у мережевій топології.

4. Customers Service (порт 8081) — відповідає за управління інформацією про власників домашніх тварин та їхні контактні дані. Сервіс має власну базу даних і надає REST API для створення, читання, оновлення та видалення даних про клієнтів.

5. Vets Service (порт 8083) — керує даними про ветеринарів та їхні спеціалізації, має окрему базу даних та власне API.

6. Visits Service (порт 8082) — управляє інформацією про візити тварин до ветеринара, включно з датою, описом та ідентифікаторами тварини (раніше в проекті був реалізований як Pets Service, але був повністю перероблений). Як і інші сервіси, має власну базу даних.

7. Admin Server (порт 9090) — надає графічний інтерфейс для моніторингу стану мікросервісів, їхніх метрик та логів на базі Spring Boot Admin.

У контексті розглянутої архітектури слід зазначити, що є окремий GenAI сервіс. Хоча технології генеративного штучного інтелекту (GenAI) набувають популярності в сучасних розподілених системах, для реалізації нашого проекту немає необхідності їх впроваджувати. Інтеграція GenAI-компонентів, таких як чат-боти чи системи рекомендацій для власників тварин або ветеринарів, значно ускладнила б архітектуру системи та вимагала б додаткових обчислювальних ресурсів, які виходять за межі безкоштовних тарифів хмарних провайдерів.

Впровадження GenAI-функціоналу передбачало б створення окремого мікросервісу з власними моделями машинного навчання, системами тренування та інференсу, а також розробку спеціалізованих API для взаємодії з іншими сервісами. Це суттєво збільшило б складність проекту, вимоги до

інфраструктури та подовжило б цикли розробки, що суперечить нашій меті — демонстрації ефективного CI/CD-процесу в розподіленій системі.

Натомість, зосередження на основній функціональності веб-застосунку та її автоматизованому розгортанні дозволить більш чітко продемонструвати принципи CI/CD без додаткових ускладнень, пов'язаних із інтеграцією технологій штучного інтелекту.

Технічно проєкт Spring PetClinic Microservices базується на таких технологіях:

- Java 17 та Spring Boot 3.0 — сучасна платформа для створення мікросервісів із високою продуктивністю
- Spring Cloud — надає інструменти для розробки хмарних застосунків, зокрема для реалізації шаблону API Gateway, Config Server та Service Discovery
- Hibernate/JPA — для роботи з базами даних
- Thymeleaf — для серверного рендерингу HTML-сторінок
- Maven — для управління залежностями та збірки проєкту
- H2 Database — вбудована база даних для розробки та тестування
- Frontend використовує Angular 13.

Комунікація між сервісами відбувається за допомогою REST API та метаданих сервісів, зареєстрованих у Discovery Server (Eureka). За замовчуванням кожен мікросервіс використовує власну вбудовану базу даних H2, хоча в основному середовищі можна замінити їх на повноцінні бази даних, такі як PostgreSQL або MySQL.

Оскільки кожен мікросервіс є незалежним додатком Spring Boot, це створює ідеальні умови для впровадження CI/CD-процесів, де кожен сервіс може мати власні цикли розробки, тестування та розгортання. Така структура дозволяє продемонструвати ключові аспекти CI/CD у розподіленому середовищі:

- Паралельну розробку та незалежний життєвий цикл компонентів

- Автоматизоване тестування як окремих сервісів, так і їхньої взаємодії
- Гнучкі стратегії розгортання з можливістю оновлення окремих сервісів без впливу на загальну роботу системи
- Централізоване керування конфігураціями через Config Server
- Динамічне виявлення сервісів за допомогою Discovery Server

Кожен із цих мікросервісів здатний функціонувати незалежно, але разом вони утворюють повноцінний веб-застосунок. Така архітектура відповідає сучасним тенденціям розробки програмного забезпечення і дозволяє ефективно використовувати практики CI/CD, забезпечуючи швидке та надійне розгортання змін.

Для локальної розробки кожен мікросервіс може бути запущено окремо, але для повноцінного розгортання та масштабування у хмарному середовищі потрібен контейнеризований підхід з оркестрацією, який буде детально розглянуто в наступних підрозділах.

Таким чином, Spring PetClinic Microservices є ідеальним проєктом для демонстрації впровадження CI/CD-процесів у розподіленій системі, оскільки він має чітку структуру мікросервісів із визначеними межами відповідальності та різноманітними технологіями, що використовуються в сучасних розподілених системах.

3.2 Розробка та впровадження CI/CD-пайплайну: побудова, тестування, реліз

Розробка та впровадження CI/CD-пайплайну для Spring PetClinic Microservices вимагає поетапного підходу, починаючи від налаштування базової інфраструктури і закінчуючи автоматизацією всіх етапів життєвого циклу розробки. Проєкт складається з декількох мікросервісів із власними циклами розробки, що дозволяє кожному сервісу мати індивідуальний пайплайн, але з узгодженою загальною структурою.

Оскільки в цій роботі використовується безкоштовна версія GitLab.com

разом із локальним середовищем на базі minikube, для реалізації повного CI/CD процесу буде застосований гібридний підхід:

1. CI (безперервна інтеграція) буде повністю автоматизована через GitLab.com
2. CD (безперервна доставка/розгортання) буде частково автоматизована з ручним запуском скриптів розгортання у локальному minikube

Такий підхід враховує обмеження безкоштовної версії GitLab.com та оптимально підходить для навчальних цілей.

Першим кроком є створення облікового запису на GitLab.com. Це безкоштовна хмарна версія GitLab, яка надає всі необхідні інструменти для реалізації CI/CD. Для створення облікового запису потрібно перейти на сайт GitLab.com і заповнити форму реєстрації. Після підтвердження електронної пошти, аккаунт стає активним. У безкоштовному плані GitLab.com пропонує 400 хвилин виконання CI/CD-задач щомісяця [36], що достатньо для навчальних та експериментальних цілей.

Після створення облікового запису необхідно налаштувати SSH-ключі для безпечного доступу до репозиторіїв. Для цього в терміналі виконаємо команди вказані на рисунку 3.2:

```
# Генерація SSH-ключа
ssh-keygen -t rsa -b 4096 -C "ваша.пошта@приклад.com"

# Виведення публічного ключа для копіювання в GitLab
cat ~/.ssh/id_rsa.pub
```

Рисунок 3.2 – Генерація та виведення SSH-ключа

Отриманий публічний ключ необхідно додати в налаштуваннях профілю GitLab через інтерфейс користувача: Preferences → SSH Keys. Це забезпечить безпечний доступ до репозиторіїв без введення паролля при кожній операції [38].

Наступним кроком є створення проєкту в GitLab та клонування вихідного коду Spring PetClinic Microservices. У GitLab створюємо новий проєкт, вибравши "New project" → "Create blank project" [40], задаємо назву "spring-

petclinic-microservices". Для клонування оригінального коду з GitHub і його передачі в

```
# Клонування репозиторію з GitHub
git clone https://github.com/spring-petclinic/spring-petclinic-microservices.git

# Перехід у директорию проекту
cd spring-petclinic-microservices

# Зміна URL віддаленого репозиторію на GitLab
git remote remove origin
git remote add origin git@gitlab.com:BChubuk/spring-petclinic-microservices.git

# Відправлення коду в GitLab
git push -u origin main
```

Рисунок 3.3 – Клонування репозиторію та зміна репозиторію на GitLab

Тепер, коли код проєкту доступний у GitLab, можна перейти до налаштування CI/CD-пайплайну. GitLab CI/CD використовує файл .gitlab-ci.yml в кореневій директорії проєкту для визначення кроків пайплайну.

Перед налаштуванням пайплайну важливо розуміти, як працює збірка проєкту на базі Maven. Spring PetClinic Microservices використовує Maven для управління збіркою та залежностями. Основна структура проєкту:

- pom.xml – корінний файл конфігурації Maven
- /spring-petclinic-admin-server – мікросервіс Admin Server
- /spring-petclinic-api-gateway – мікросервіс API Gateway
- /spring-petclinic-config-server – мікросервіс Config Server
- /spring-petclinic-customers-service – мікросервіс Customers
- /spring-petclinic-discovery-server – мікросервіс Discovery Server
- /spring-petclinic-vets-service – мікросервіс Vets
- /spring-petclinic-visits-service – мікросервіс Visits

Кожна директорія містить власний pom.xml та структуру Java-проєкту. Центральний pom.xml організовує проєкт як багатомодульний, а Maven виконує збірку та тестування компонентів у порядку їх залежностей [42].

Для зберігання Docker-образів мікросервісів необхідно створити обліковий запис на Docker Hub, наприклад, з ім'ям користувача "xchubuk" [43]. Потім увійти в Docker Hub і створити репозиторії для кожного мікросервісу:

- xchubuk/admin-server
- xchubuk/api-gateway

- xchubuk/config-server
- xchubuk/customers-service
- xchubuk/discovery-server
- xchubuk/vets-service
- xchubuk/visits-service

Після цього необхідно авторизуватися в Docker Hub на локальній машині [44] (див. рисунок 3.4):

```
lifthacker@2526575508:~/Desktop/Diploma/test_projects/spring-petclinic-microservices$ docker login
Authenticating with existing credentials...
WARNING! Your password will be stored unencrypted in /home/lifthacker/.docker/config.json.
Configure a credential helper to remove this warning. See
https://docs.docker.com/engine/reference/commandline/login/#credential-stores

Login Succeeded
```

Рисунок 3.4 – Клонування репозиторію та зміна репозиторію на GitLab

Файл .gitlab-ci.yml має виглядати так (див. рисунок 3.5):

```
.gitlab-ci.yml
1  stages:
2    - build
3    - test
4    - package
5    - docker
6
7  variables:
8    MAVEN_OPTS: "-Dmaven.repo.local=.m2/repository -Dorg.slf4j.simpleLogger.log.org.apache.maven.cli.transfer.Slf4jMavenTransferListener=WARN
9    -Dorg.slf4j.simpleLogger.showDateTime=true -Djava.awt.headless=true"
10   MAVEN_CLI_OPTS: "--batch-mode --errors --fail-at-end --show-version"
11   DOCKER_HOST: tcp://docker:2375
12   DOCKER_DRIVER: overlay2
13   DOCKER_TLS_CERTDIR: ""
14   IMAGE_TAG: ${CI_COMMIT_SHORT_SHA}
15
16 # Кешування залежностей Maven для прискорення збірки
17 cache:
18   key: ${CI_COMMIT_REF_SLUG}
19   paths:
20     - .m2/repository
21
```

Рисунок 3.5 – Початкова секція файлу .gitlab-ci.yml

Початкова секція файлу визначає етапи пайплайну та глобальні змінні. Використання кешування для Maven-залежностей значно прискорює виконання кожного наступного пайплайну, оскільки завантаження залежностей відбувається лише один раз.

Наступні секції описують етап збірки та тестування (див. рисунок 3.6):

```

~/Desktop/Diploma/test_projects/spring-petclinic-microservices/.gitlab-ci.yml • Modified
21 # Етап збірки - компіляція коду
22 build:
23   stage: build
24   image: maven:3.8-openjdk-17-slim
25   script:
26     - mvn $MAVEN_CLI_OPTS compile
27   rules:
28     - if: $CI_COMMIT_BRANCH
29
30 # Етап тестування - запуск модульних тестів
31 test:
32   stage: test
33   image: maven:3.8-openjdk-17-slim
34   script:
35     - mvn $MAVEN_CLI_OPTS test
36   artifacts:
37     reports:
38       junit:
39         - "**/target/surefire-reports/TEST-*.xml"
40   needs:
41     - build
42   rules:
43     - if: $CI_COMMIT_BRANCH

```

Рисунок 3.6 – Етап збірки та тестування у файлі .gitlab-ci.yml

На етапі збірки відбувається компіляція коду всіх мікросервісів. Для цього використовується образ Docker-образ `maven:3.8-openjdk-17`, який надає середовище з Maven та Java 17 для збірки та тестування проєкту, що відповідає вимогам Spring PetClinic Microservices. Етап тестування запускає модульні тести та зберігає їх результати у форматі Junit [45] для подальшого аналізу.

Етап пакування створює JAR-файли [46] для кожного мікросервісу (див. рисунок 3.7):

```

.gitlab-ci.yml
45 # Етап пакування - створення JAR-файлів
46 package:
47   stage: package
48   image: maven:3.8-openjdk-17-slim
49   script:
50     - mvn $MAVEN_CLI_OPTS package -DskipTests
51   artifacts:
52     paths:
53       - spring-petclinic-admin-server/target/*.jar
54       - spring-petclinic-api-gateway/target/*.jar
55       - spring-petclinic-config-server/target/*.jar
56       - spring-petclinic-customers-service/target/*.jar
57       - spring-petclinic-discovery-server/target/*.jar
58       - spring-petclinic-vets-service/target/*.jar
59       - spring-petclinic-visits-service/target/*.jar
60     expire_in: 1 week
61   needs:
62     - test
63   rules:
64     - if: $CI_COMMIT_BRANCH

```

Рисунок 3.7 – Етап пакування у файлі .gitlab-ci.yml

Згенеровані JAR-файли зберігаються як артефакти пайплайну і можуть бути скачані або використані на наступних етапах. Параметр `-DskipTests` використовується для пропуску тестів, оскільки вони вже виконані на попередньому етапі [47].

Останній етап — створення та публікація Docker-образів (див. рисунок 3.8):

```
# Етап збірки та публікації Docker-образів в одній задачі
docker-build-push:
  stage: docker
  image: docker:20.10.16
  services:
    - docker:20.10.16-dind
  script:
    # Логін в Docker Hub
    - echo "${DOCKER_HUB_PASSWORD}" | docker login -u "${DOCKER_HUB_USERNAME}" --password-stdin

    # Сбірка імеджів
    - echo "Building and pushing Docker images with tag ${IMAGE_TAG}"

    # Admin Server
    - docker build -t ${DOCKER_HUB_USERNAME}/petclinic-admin-server:${IMAGE_TAG} -f docker/admin-server/Dockerfile .
    - docker tag ${DOCKER_HUB_USERNAME}/petclinic-admin-server:${IMAGE_TAG} ${DOCKER_HUB_USERNAME}/petclinic-admin-server:latest
    - docker push ${DOCKER_HUB_USERNAME}/petclinic-admin-server:${IMAGE_TAG}
    - docker push ${DOCKER_HUB_USERNAME}/petclinic-admin-server:latest

    # API Gateway
    - docker build -t ${DOCKER_HUB_USERNAME}/petclinic-api-gateway:${IMAGE_TAG} -f docker/api-gateway/Dockerfile .
    - docker tag ${DOCKER_HUB_USERNAME}/petclinic-api-gateway:${IMAGE_TAG} ${DOCKER_HUB_USERNAME}/petclinic-api-gateway:latest
    - docker push ${DOCKER_HUB_USERNAME}/petclinic-api-gateway:${IMAGE_TAG}
    - docker push ${DOCKER_HUB_USERNAME}/petclinic-api-gateway:latest

    # Config Server
    - docker build -t ${DOCKER_HUB_USERNAME}/petclinic-config-server:${IMAGE_TAG} -f docker/config-server/Dockerfile .
    - docker tag ${DOCKER_HUB_USERNAME}/petclinic-config-server:${IMAGE_TAG} ${DOCKER_HUB_USERNAME}/petclinic-config-server:latest
    - docker push ${DOCKER_HUB_USERNAME}/petclinic-config-server:${IMAGE_TAG}
    - docker push ${DOCKER_HUB_USERNAME}/petclinic-config-server:latest

    # Customers Service
    - docker build -t ${DOCKER_HUB_USERNAME}/petclinic-customers-service:${IMAGE_TAG} -f docker/customers-service/Dockerfile .
    - docker tag ${DOCKER_HUB_USERNAME}/petclinic-customers-service:${IMAGE_TAG} ${DOCKER_HUB_USERNAME}/petclinic-customers-service:latest
    - docker push ${DOCKER_HUB_USERNAME}/petclinic-customers-service:${IMAGE_TAG}
    - docker push ${DOCKER_HUB_USERNAME}/petclinic-customers-service:latest

    # Discovery Server
    - docker build -t ${DOCKER_HUB_USERNAME}/petclinic-discovery-server:${IMAGE_TAG} -f docker/discovery-server/Dockerfile .
    - docker tag ${DOCKER_HUB_USERNAME}/petclinic-discovery-server:${IMAGE_TAG} ${DOCKER_HUB_USERNAME}/petclinic-discovery-server:latest
    - docker push ${DOCKER_HUB_USERNAME}/petclinic-discovery-server:${IMAGE_TAG}
    - docker push ${DOCKER_HUB_USERNAME}/petclinic-discovery-server:latest

    # Vets Service
    - docker build -t ${DOCKER_HUB_USERNAME}/petclinic-vets-service:${IMAGE_TAG} -f docker/vets-service/Dockerfile .
    - docker tag ${DOCKER_HUB_USERNAME}/petclinic-vets-service:${IMAGE_TAG} ${DOCKER_HUB_USERNAME}/petclinic-vets-service:latest
    - docker push ${DOCKER_HUB_USERNAME}/petclinic-vets-service:${IMAGE_TAG}
    - docker push ${DOCKER_HUB_USERNAME}/petclinic-vets-service:latest

    # Visits Service
    - docker build -t ${DOCKER_HUB_USERNAME}/petclinic-visits-service:${IMAGE_TAG} -f docker/visits-service/Dockerfile .
    - docker tag ${DOCKER_HUB_USERNAME}/petclinic-visits-service:${IMAGE_TAG} ${DOCKER_HUB_USERNAME}/petclinic-visits-service:latest
    - docker push ${DOCKER_HUB_USERNAME}/petclinic-visits-service:${IMAGE_TAG}
    - docker push ${DOCKER_HUB_USERNAME}/petclinic-visits-service:latest

    # Запис інформації про деплой
    - echo "Deployed images with tag ${IMAGE_TAG} at $(date)" > deployment.txt
  artifacts:
    paths:
      - deployment.txt
    expire_in: 1 week
  needs:
    - package
  rules:
    - if: $CI_COMMIT_BRANCH
```

Рисунок 3.8 – Етап створення та публікації образів у файлі `.gitlab-ci.yml`

Цей етап використовує Docker-in-Docker (DinD) [48] для створення та публікації образів у Docker Hub. Для кожного мікросервісу створюються два

теги образу: один із хешем коміту (`${CI_COMMIT_SHORT_SHA}`) [49] для точної ідентифікації версії, другий — `latest` для спрощення процедури розгортання.

Таким чином, був налаштований базовий пайплайн, який має чотири етапи:

1. `build` — компіляція вихідного коду
2. `test` — запуск тестів
3. `package` — створення JAR-файлів для кожного мікросервісу
4. `docker` — створення та публікація Docker-образів

Як було зазначено раніше, також було налаштовано кешування Maven-залежностей (`.m2/repository`), що значно прискорює збірку при повторних запусках.

Додамо цей файл до репозиторію та відправимо зміни в GitLab (див. рисунок 3.9):

```
liferhacker@2526575508:~/Desktop/Diploma/test_projects/spring-petclinic-microservices$ git push
Enumerating objects: 5, done.
Counting objects: 100% (5/5), done.
Delta compression using up to 12 threads
Compressing objects: 100% (3/3), done.
Writing objects: 100% (3/3), 317 bytes | 317.00 KiB/s, done.
Total 3 (delta 2), reused 0 (delta 0), pack-reused 0 (from 0)
To gitlab.com:BChubuk/spring-petclinic-microservices.git
1ef159f..e4aa355 main -> main
```

Рисунок 3.9 – Відправлення змін у файлі `.gitlab-ci.yml` на віддалений репозиторій

Після відправлення цих змін GitLab автоматично запустить пайплайн згідно з налаштуваннями у `.gitlab-ci.yml`. Прогрес виконання можна спостерігати в інтерфейсі GitLab на сторінці проєкту в розділі CI/CD → Pipelines (див. рисунок 3.10).

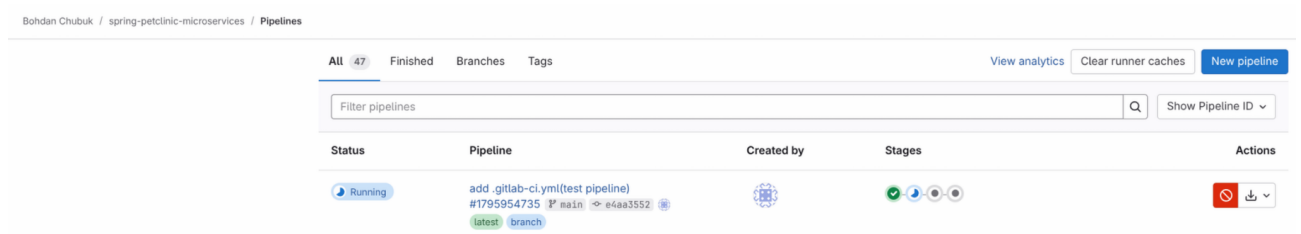


Рисунок 3.10 – Автоматичний запуск пайплайну у GitLab.com

Для більшої безпеки та кращого управління конфігурацією, були використані вбудовані в GitLab змінні CI/CD [50]. Щоб додати необхідні змінні, переходимо в налаштування проєкту: Settings → CI/CD → Variables.

Додамо наступні змінні:

- DOCKER_HUB_USERNAME — ім'я користувача Docker Hub (для публікації образів)
- DOCKER_HUB_PASSWORD — пароль Docker Hub

Обидві змінні позначаємо як "Protected" (щоб вони використовувалися лише в захищених гілках) та "Masked" (щоб приховати їх значення в логах) [51].

Тепер розширимо наш пайплайн, додавши функціональність створення Docker-образів для кожного мікросервісу. Для цього в кореневій директорії проєкту створимо директорії для Docker-файлів для кожного сервісу, наприклад, docker/api-gateway/Dockerfile. Зміст файлу для API Gateway (див. рисунок 3.11):

```
docker > api-gateway > Dockerfile > ...
1  # Етап збірки
2  FROM maven:3.8-openjdk-17-slim AS build
3  WORKDIR /app
4  COPY . .
5  RUN mvn clean package -DskipTests -pl spring-petclinic-api-gateway -am
6
7  # Фінальний образ
8  FROM openjdk:17-jdk-slim
9  WORKDIR /app
10 COPY --from=build /app/spring-petclinic-api-gateway/target/*.jar app.jar
11 ENV SPRING_PROFILES_ACTIVE=kubernetes
12 EXPOSE 8080
13 ENTRYPOINT ["java", "-jar", "app.jar"]
```

Рисунок 3.11 – Зміст Dockerfile для API Gateway

Аналогічні Dockerfile створюються для всіх мікросервісів з відповідними налаштуваннями.

Для забезпечення безпеки, використовується безпечний вхід у Docker Hub перед відправленням образів. Також важливо відзначити, що в реальному проєкті краще використовувати приватний реєстр Docker-образів, але для навчальних цілей публічний Docker Hub є прийнятним варіантом [39].

Для адаптації стандартного проєкту Spring PetClinic Microservices до роботи з Kubernetes були створені додаткові конфігураційні файли з профілем kubernetes для кожного мікросервісу. Ці конфігураційні файли містять спеціалізовані налаштування, що дозволяють мікросервісам ефективно функціонувати в оркестраційному середовищі Kubernetes, використовуючи його нативні можливості для виявлення сервісів, балансування навантаження та управління конфігурацією [52]. Нижче наведена структура цих файлів з повними шляхами (див. у таблиці 3.1):

Таблиця 3.1 – Структура файлів конфігурації

Мікросервіс	Шлях
Config Server	spring-petclinic-config-server/src/main/resources/application-kubernetes.yml
Discovery Server	spring-petclinic-discovery-server/src/main/resources/application-kubernetes.yml
Admin Server	spring-petclinic-admin-server/src/main/resources/application-kubernetes.yml
API Gateway	spring-petclinic-api-gateway/src/main/resources/application-kubernetes.yml
Customers Service	spring-petclinic-customers-service/src/main/resources/application-kubernetes.yml
Vets Service	spring-petclinic-vets-service/src/main/resources/application-kubernetes.yml
Visits Service	spring-petclinic-visits-service/src/main/resources/application-kubernetes.yml

Сервер конфігурації є центральним компонентом архітектури, відповідальним за зберігання і розповсюдження конфігураційних налаштувань для всіх інших мікросервісів. Розглянемо конфігурацію для Kubernetes зміст файлу для API Gateway (див. рисунок 3.12):

```

spring-petclinic-config-server > src > main > resources > ! application-kubernetes.yml
1  spring:
2    cloud:
3      config:
4        server:
5          git:
6            uri: https://github.com/spring-petclinic/spring-petclinic-microservices-config
7            default-label: main
8  server:
9    port: 8888

```

Рисунок 3.12 – Зміст application-kubernetes.yml для API Gateway

Основні налаштування включають:

- Конфігурація Git-репозиторію як джерела налаштувань (uri та default-label)
- Стандартний порт 8888 для сервера конфігурації

Ця конфігурація дозволяє централізовано керувати налаштуваннями всіх мікросервісів, забезпечуючи узгодженість та спрощуючи процес їх оновлення.

Сервер виявлення є критичним компонентом для забезпечення комунікації між мікросервісами. В архітектурі використовується Eureka як сервіс виявлення (див. рисунок 3.13):

```
spring-petclinic-discovery-server > src > main > resources > ! application-kubernetes.yml
1  spring:
2    application:
3      name: discovery-server
4    config:
5      import: optional:configserver:http://config-server.petclinic.svc.cluster.local:8888
6  server:
7    port: 8761
8  eureka:
9    instance:
10     hostname: discovery-server.petclinic.svc.cluster.local
11     preferIpAddress: true
12   client:
13     registerWithEureka: false
14     fetchRegistry: false
15     serviceUrl:
16       defaultZone: http://discovery-server.petclinic.svc.cluster.local:8761/eureka/
```

Рисунок 3.13 – Зміст application-kubernetes.yml для Discovery Server

Ключові особливості конфігурації:

- Використання DNS-імені Kubernetes для доступу до Config Server (config-server.petclinic.svc.cluster.local)
- Налаштування DNS-імені для самого Discovery Server в межах кластера Kubernetes
- Встановлення параметру preferIpAddress: true для використання IP-адрес замість імен хостів
- Відключення самореєстрації для сервера виявлення через параметри registerWithEureka: false та fetchRegistry: false

Використання доменних імен у форматі `<service-name>.<namespace>.svc.cluster.local` є ключовою адаптацією для Kubernetes, оскільки це стандартний формат DNS для служб Kubernetes [53].

Адміністративний сервер відповідає за моніторинг та управління всіма мікросервісами (див. рисунок 3.14):

```
spring-petclinic-admin-server > src > main > resources > ! application-kubernetes.yml
1  spring:
2    application:
3      name: admin-server
4    config:
5      import: optional:configserver:http://config-server.petclinic.svc.cluster.local:8888
6  server:
7    port: 9090
8  eureka:
9    client:
10     serviceUrl:
11       defaultZone: http://discovery-server.petclinic.svc.cluster.local:8761/eureka/
12     registryFetchIntervalSeconds: 5
13   instance:
14     preferIpAddress: true
15     leaseRenewalIntervalInSeconds: 10
```

Рисунок 3.14 – Зміст application-kubernetes.yml для Admin Server

Варто відзначити:

- Зменшений інтервал оновлення реєстру сервісів (`registryFetchIntervalSeconds: 5`) [54] для швидшого оновлення стану мікросервісів
- Налаштування частіших оновлень статусу (`leaseRenewalIntervalInSeconds: 10`) [55] для підтримки актуальної інформації про доступність сервісів
- Використання тих самих DNS-шаблонів Kubernetes для посилання на інші сервіси

API Gateway служить єдиною точкою входу для всіх клієнтських запитів і відповідає за їх маршрутизацію до відповідних мікросервісів (див. рисунок 3.15):

```

spring-petclinic-api-gateway > src > main > resources > ! application-kubernetes.yml
1  spring:
2    application:
3      name: api-gateway
4    config:
5      import: optional:configserver:http://config-server.petclinic.svc.cluster.local:8888
6    cloud:
7      gateway:
8        globalcors:
9          corsConfigurations:
10             '[/*]':
11               allowedOrigins: "*"
12               allowedMethods:
13                 - GET
14                 - POST
15                 - PUT
16                 - DELETE
17                 - OPTIONS
18               allowedHeaders: "*"
19          routes:
20             - id: vets-service
21               uri: lb://vets-service
22               predicates:
23                 - Path=/api/vet/**
24               filters:
25                 - StripPrefix=2
26             - id: visits-service
27               uri: lb://visits-service
28               predicates:
29                 - Path=/api/visit/**
30               filters:
31                 - StripPrefix=2
32             - id: customers-service
33               uri: lb://customers-service
34               predicates:
35                 - Path=/api/customer/**
36               filters:
37                 - StripPrefix=2
38          loadbalancer:
39            ribbon:
40              enabled: false
41    server:
42      port: 8080
43    eureka:
44      client:
45        serviceUrl:
46          defaultZone: http://discovery-server.petclinic.svc.cluster.local:8761/eureka/
47      instance:
48        preferIpAddress: true
49        instanceId: ${spring.application.name}:${vcap.application.instance_id:${spring.application.instance_id:${random.value}}}
50        leaseRenewalIntervalInSeconds: 30

```

Рисунок 3.15 – Конфігурація маршрутизації API Gateway

Особливості конфігурації:

- Комплексні налаштування CORS для підтримки різних типів клієнтських додатків [56]
- Визначення маршрутів до різних мікросервісів з префіксами API
- Використання клієнтського балансування навантаження через prefixes lb://
- Відключення Ribbon як балансувальника навантаження (ribbon.enabled: false) [57] на користь вбудованого механізму Spring Cloud LoadBalancer

- Унікальна генерація ідентифікатора екземпляра для підтримки декількох реплік у Kubernetes

Бізнес-мікросервіси мають схожу конфігурацію з деякими відмінностями в іменах та портах. Розглянемо конфігурацію Customers Service як приклад (див. рисунок 3.16):

```
spring-petclinic-customers-service > src > main > resources > ! application-kubernetes.yml
1  spring:
2    application:
3      name: customers-service
4    config:
5      import: optional:configserver:http://config-server.petclinic.svc.cluster.local:8888
6    datasource:
7      schema: classpath*:db/hsqldb/schema.sql
8      data: classpath*:db/hsqldb/data.sql
9    jpa:
10     hibernate:
11       ddl-auto: none
12  server:
13    port: 8081
14  eureka:
15    client:
16      serviceUrl:
17        defaultZone: http://discovery-server.petclinic.svc.cluster.local:8761/eureka/
18    instance:
19      preferIpAddress: true
20      instanceId: ${spring.application.name}:${vcap.application.instance_id:${spring.application.instance_id:${random.value}}}
21      leaseRenewalIntervalInSeconds: 30
```

Рисунок 3.16 – Конфігурація сервісу Customers

Основні елементи:

- Налаштування джерела даних та JPA [58] для роботи з базою даних
- Унікальний порт для кожного сервісу (8081 для Customers, 8082 для Visits, 8083 для Vets)
- Використання Eureka для реєстрації та виявлення сервісів
- Конфігурація для генерації унікальних ідентифікаторів екземплярів, що дозволяє запускати декілька реплік одного сервісу

Тепер ми маємо базовий CI/CD-пайплайн, який виконує такі кроки:

1. Компіляція вихідного коду
2. Запуск тестів
3. Створення JAR-файлів
4. Побудова Docker-образів та їх відправлення в Docker Hub

Важливо зазначити, що налаштований CI/CD-пайплайн враховує кращі практики, виявлені під час аналізу в розділі 2:

- Використання кешування для оптимізації часу виконання
- Паралельне виконання незалежних задач
- Збереження артефактів для подальшого використання
- Тегування Docker-образів для забезпечення відстежуваності

У наступних підрозділах ми розширимо цей пайплайн, додавши етапи автоматизованого розгортання в Kubernetes, а також налаштуємо більш складні стратегії тестування та розгортання, включаючи син розгортання Blue-Green та автоматичний відкат у випадку проблем.

Для реалізації повної автоматизації CI/CD для розподіленої системи Spring PetClinic Microservices, необхідно продовжити налаштування пайплайну, додавши етапи розгортання в Kubernetes, налаштування моніторингу та інших аспектів, які будуть розглянуті в наступних підрозділах.

3.3 Налаштування контейнеризації та керування сервісами в оркестраційному середовищі

У цьому розділі розглянемо практичне налаштування контейнеризації мікросервісів Spring PetClinic та їх керування в оркестраційному середовищі Kubernetes (minikube). Після того, як CI-частина пайплайну успішно збудувала та опублікувала Docker-образи в Docker Hub, необхідно забезпечити їх правильне розгортання та керування в Kubernetes.

Для ефективного розгортання та керування мікросервісами впроваджено GitOps-підхід, який передбачає, що всі зміни в інфраструктурі опосередковано здійснюються через Git-репозиторій, що забезпечує прозорість, відстежуваність та можливість відкату змін [59].

Для локального розробки та тестування використовується minikube — інструмент, що дозволяє запустити одновузловий кластер Kubernetes на локальному комп'ютері [60]. В нашому випадку minikube вже встановлений на Fedora Linux з Docker як основним драйвером.

Для початку роботи з minikube потрібно переконатися, що кластер запущений та доступний (див. рисунок 3.17):

```
lifelacker@2526575508:~/Desktop/Diploma/test_projects/spring-petclinic-microservices$ minikube status
minikube
type: Control Plane
host: Stopped
kubelet: Stopped
apiserver: Stopped
kubeconfig: Stopped

lifelacker@2526575508:~/Desktop/Diploma/test_projects/spring-petclinic-microservices$ minikube start --driver=docker --cpus=4 --memory=8g
minikube v1.35.0 on Fedora 40
✪ Using the docker driver based on existing profile
! You cannot change the memory size for an existing minikube cluster. Please first delete the cluster.
! You cannot change the CPUs for an existing minikube cluster. Please first delete the cluster.
✪ Starting "minikube" primary control-plane node in "minikube" cluster
✪ Pulling base image v0.0.46 ...
✪ Restarting existing docker container for "minikube" ...
✪ Preparing Kubernetes v1.32.0 on Docker 27.4.1 ...
✪ Verifying Kubernetes components...
  • Using image gcr.io/k8s-minikube/storage-provisioner:v5
  • Using image registry.k8s.io/metrics-server/metrics-server:v0.7.2
  • Using image registry.k8s.io/ingress-nginx/controller:v1.11.3
  • Using image registry.k8s.io/ingress-nginx/kube-webhook-certgen:v1.4.4
  • Using image registry.k8s.io/ingress-nginx/kube-webhook-certgen:v1.4.4
✪ Verifying ingress addon...
✪ Enabled addons: metrics-server, storage-provisioner, default-storageclass, ingress
✪ Done! kubectl is now configured to use "minikube" cluster and "default" namespace by default
lifelacker@2526575508:~/Desktop/Diploma/test_projects/spring-petclinic-microservices$ kubectl cluster-info
Kubernetes control plane is running at https://192.168.49.2:8443
CoreDNS is running at https://192.168.49.2:8443/api/v1/namespaces/kube-system/services/kube-dns:dns/proxy

To further debug and diagnose cluster problems, use 'kubectl cluster-info dump'.
```

Рисунок 3.17 – Перевірка статусу кластера minikube

Для розгортання мікросервісів Spring PetClinic в Kubernetes створено YAML-маніфести [61] в директорії kubernetes для кожного сервісу. Наприклад, маніфест kubernetes/api-gateway.yaml має наступний вигляд (див. рисунок 3.18):

```

kubernetes > ! api-gateway.yaml
1  apiVersion: apps/v1
2  kind: Deployment
3  metadata:
4    name: api-gateway
5    labels:
6      app: api-gateway
7  spec:
8    replicas: 1
9    selector:
10     matchLabels:
11       app: api-gateway
12    template:
13     metadata:
14       labels:
15         app: api-gateway
16     spec:
17       containers:
18         - name: api-gateway
19           image: xchubuk/petclinic-api-gateway:latest
20           imagePullPolicy: Always
21           ports:
22             - containerPort: 8080
23           env:
24             - name: SPRING_PROFILES_ACTIVE
25               value: "kubernetes"
26             - name: SPRING_CONFIG_IMPORT
27               value: "optional:configserver:http://config-server:8888"
28 ---
29 apiVersion: v1
30 kind: Service
31 metadata:
32   name: api-gateway
33 spec:
34   selector:
35     app: api-gateway
36   ports:
37     - port: 8080
38       targetPort: 8080
39   type: NodePort

```

Аналогічні маніфести створено для всіх мікросервісів, що забезпечує їх правильне розгортання в кластері Kubernetes (див. Додатки Б – Д).

Враховуючи обмеження прямого доступу з GitLab CI до локального minikube, було розроблено локальний скрипт `deploy-to-minikube.sh`, який автоматизує процес розгортання оновлених образів (див. рисунок 3.19):

Рисунок 3.18 – Маніфест Kubernetes для API Gateway

```
#!/bin/bash

# Перевірка стану minikube
minikube status || minikube start

# Створення простору імен, якщо він не існує
kubectl get namespace petclinic || kubectl create namespace petclinic

# Вхід у Docker Hub для отримання образів
docker login

# Оновлення образів у локальному minikube
echo "Оновлення Docker образів у minikube..."
minikube image pull xchubuk/petclinic-config-server:latest
minikube image pull xchubuk/petclinic-discovery-server:latest
minikube image pull xchubuk/petclinic-customers-service:latest
minikube image pull xchubuk/petclinic-visits-service:latest
minikube image pull xchubuk/petclinic-vets-service:latest
minikube image pull xchubuk/petclinic-api-gateway:latest
minikube image pull xchubuk/petclinic-admin-server:latest

# Застосування Kubernetes манифестів
echo "Застосування Kubernetes манифестів..."
kubectl apply -f kubernetes/config-server.yaml -n petclinic
kubectl apply -f kubernetes/discovery-server.yaml -n petclinic

# Очікування запуску базових сервісів
echo "Очікування запуску Config Server та Discovery Server..."
kubectl wait --for=condition=ready pod -l app=config-server -n petclinic --timeout=180s
kubectl wait --for=condition=ready pod -l app=discovery-server -n petclinic --timeout=180s

# Розгортання решти сервісів
kubectl apply -f kubernetes/customers-service.yaml -n petclinic
kubectl apply -f kubernetes/visits-service.yaml -n petclinic
kubectl apply -f kubernetes/vets-service.yaml -n petclinic
kubectl apply -f kubernetes/api-gateway.yaml -n petclinic
kubectl apply -f kubernetes/admin-server.yaml -n petclinic

# Перезапуск розгортань для отримання нових образів
kubectl rollout restart deployment config-server
kubectl rollout restart deployment discovery-server
kubectl rollout restart deployment customers-service
kubectl rollout restart deployment visits-service
kubectl rollout restart deployment vets-service
kubectl rollout restart deployment api-gateway
kubectl rollout restart deployment admin-server

echo "Система розгорнута успішно. Перевірка статусу подів:"
kubectl get pods -n petclinic

echo "Для доступу до API Gateway виконайте:"
echo "minikube service api-gateway -n petclinic --url"
```

Рисунок 3.19 – Скрипт автоматизованого розгортання в minikube

Цей скрипт забезпечує автоматизоване розгортання мікросервісної архітектури PetClinic у середовищі Kubernetes через minikube. Скрипт послідовно ініціалізує minikube (якщо не запущено), створює виділений простір імен та завантажує необхідні Docker-образи з репозиторію. Важливою особливістю є використання команди `minikube image pull` для отримання

актуальних версій всіх компонентів системи та послідовне застосування YAML-маніфестів з очікуванням готовності критичних сервісів перед розгортанням залежних компонентів [62]. Після розгортання скрипт перезапускає всі розгортання для гарантованого оновлення та надає команду для доступу до API Gateway.

Для реалізації підходу GitOps використовується Argo CD — інструмент для декларативного та автоматизованого розгортання додатків у Kubernetes. Argo CD забезпечує синхронізацію стану кластера з бажаним станом, описаним у Git-репозиторії [63].

Процес встановлення включає створення окремого namespace та розгортання базових компонентів [64] (див. рисунок 3.20):

```
# Створення namespace для Argo CD
kubectl create namespace argocd --dry-run=client -o yaml | kubectl apply -f -

# Встановлення Argo CD
kubectl apply -n argocd -f https://raw.githubusercontent.com/argoproj/argo-cd/stable/manifests/install.yaml

# Очікування запуску компонентів Argo CD
echo "Очікування запуску компонентів Argo CD..."
kubectl wait --for=condition=ready pod -l app.kubernetes.io/name=argocd-server -n argocd --timeout=300s
```

Рисунок 3.20 – Процес встановлення Argo CD

Після встановлення основних компонентів було отримано доступ до початкового паролю адміністратора та налаштовано зовнішній доступ до веб-інтерфейсу Argo CD за допомогою спеціальних команд [65] (див. рисунок 3.21):

```
lifelacker@2526575508:~/Desktop/Diploma/test_projects/spring-petclinic-microservices$
kubectl -n argocd get secret argocd-initial-admin-secret -o jsonpath="{.data.password}" | base64 -d
YFzqKKMgwKiyNu0lifelacker@2526575508:~/Desktop/Diploma/test_projects/spring-petclinic-microservices$
lifelacker@2526575508:~/Desktop/Diploma/test_projects/spring-petclinic-microservices$
kubectl patch svc argocd-server -n argocd -p '{"spec": {"type": "NodePort"}}'
service/argocd-server patched (no change)
```

Рисунок 3.21 – Налаштування доступу до веб-інтерфейсу Argo CD

Важливо підкреслити принцип ідемпотентності [66], який є фундаментальним для декларативного підходу Kubernetes до управління ресурсами.

При застосуванні маніфесту або конфігураційного файлу без змін, Kubernetes аналізує поточний стан системи та бажаний стан, описаний у

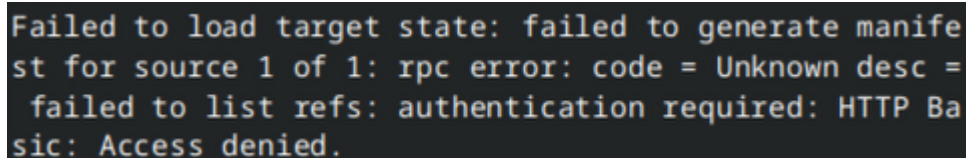
маніфесті. Якщо поточний стан відповідає бажаному, Kubernetes не виконує жодних змін у кластері, відображаючи статус операції як "unchanged" або "no change".

Це демонструє, що Kubernetes працює за принципом "reconciliation loop" (петлі узгодження), де система постійно намагається привести поточний стан до бажаного, але не виконує надлишкових операцій, якщо поточний стан вже відповідає цільовому [67].

Таким чином, при розгортанні інфраструктури з використанням підходу "Infrastructure as Code", повторне застосування тих самих конфігураційних файлів не призводить до переналаштування ресурсів, що забезпечує стабільність, передбачуваність та ефективність системи.

Це особливо важливо для практик Continuous Deployment та GitOps, де конфігураційні файли зберігаються в системі контролю версій і автоматично застосовуються до кластера, гарантуючи, що система завжди відповідає бажаному стану, описаному в репозиторії.

При першій спробі інтеграції з приватним репозиторієм GitLab була отримана помилка доступу (див. рисунок 3.22):



```
Failed to load target state: failed to generate manifest for source 1 of 1: rpc error: code = Unknown desc = failed to list refs: authentication required: HTTP Basic: Access denied.
```

Рисунок 3.22 – Помилка доступу до репозиторію GitLab

Для вирішення проблеми доступу до репозиторію спочатку необхідно створити токен доступу в GitLab [68]:

1. В обліковому записі GitLab, переходимо до налаштувань профілю (меню у верхньому правому кутку → Settings)
2. У лівому меню виберемо "Access Tokens"
3. Створюємо новий токен з такими параметрами:
 - Name: argocd-access
 - Expiration date: (для прикладу виберемо 3 місяці)

- Scopes: відмітимо "read_repository"
4. Натискаємо "Create personal access token"
 5. Обов'язково необхідно зкопіювати згенерований токен, бо він більше не буде показаний

Ось як має виглядати кінцевий результат (див. рисунок 3.23):

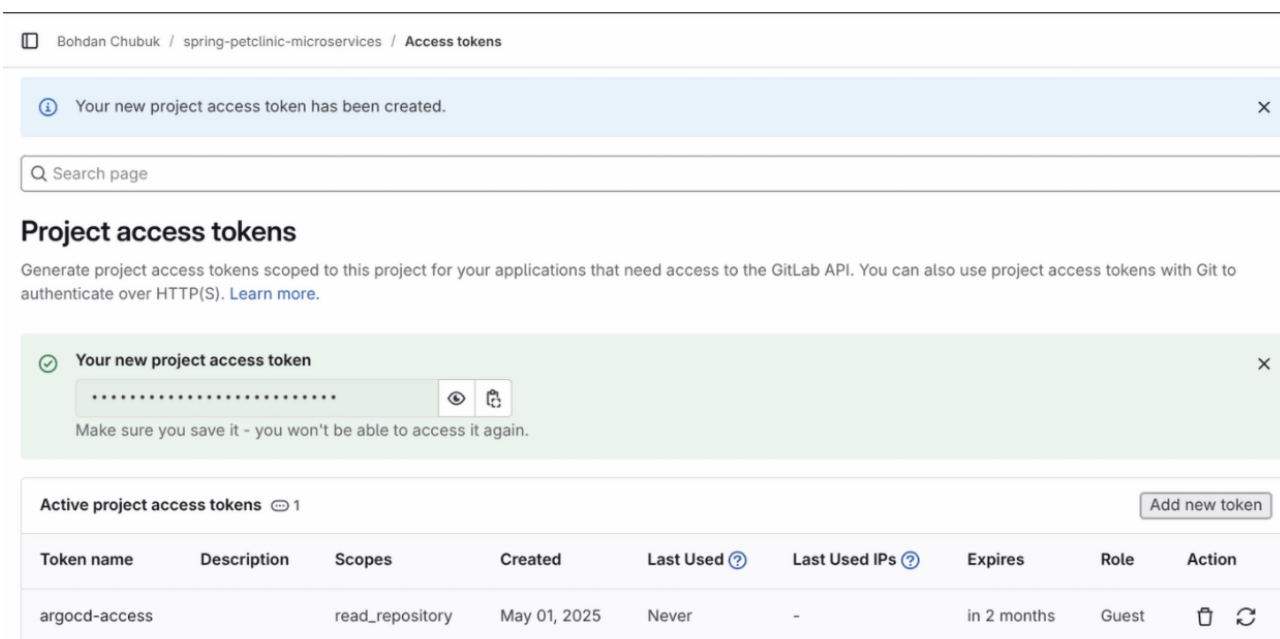


Рисунок 3.23 – Створення персонального токена доступу GitLab

Далі створюємо секрет в Kubernetes, що містить облікові дані для доступу до GitLab, включаючи створений токен для доступу в якості паролю [69] (див. рисунок 3.24):

```
lifehacker@2526575508:~/Desktop/Diploma/test_projects/spring-petclinic-microservices$ kubectl create secret generic gitlab-auth \
--namespace argocd \
--from-literal=username=BChubuk \
--from-literal=password=glpat-yr5zbMw17sZn757vsPvK
secret/gitlab-auth created
```

Рисунок 3.24 – Секрет Kubernetes для облікових даних GitLab

Тепер створимо та налаштуємо файл конфігурації репозиторію для Argo CD під назвою gitlab-repo-config.yaml (див. рисунок 3.25):

```

! gitlab-repo-config.yaml
1  apiVersion: v1
2  kind: Secret
3  metadata:
4    name: gitlab-spring-petclinic-repo
5    namespace: argocd
6    labels:
7      | argocd.argoproj.io/secret-type: repository
8  stringData:
9    type: git
10   url: https://gitlab.com/BChubuk/spring-petclinic-microservices.git
11   username: BChubuk
12   password: glpat-yr5zbMw17sZn757vsPvK

```

Рисунок 3.25 – Файл конфігурації репозиторію для Argo CD

Після забезпечення доступу до репозиторію було створено маніфест для додатку spring-petclinic з налаштуваннями автоматичної синхронізації (див. рисунок 3.26):

```

! spring-petclinic-application.yaml
1  apiVersion: argoproj.io/v1alpha1
2  kind: Application
3  metadata:
4    name: spring-petclinic
5    namespace: argocd
6  spec:
7    project: default
8    source:
9      repoURL: https://gitlab.com/BChubuk/spring-petclinic-microservices.git
10     targetRevision: main
11     path: kubernetes
12   destination:
13     server: https://kubernetes.default.svc
14     namespace: petclinic
15   syncPolicy:
16     automated:
17       prune: true
18       selfHeal: true

```

Рисунок 3.26 – Маніфест додатку Spring PetClinic для Argo CD

Застосуємо оновлений додаток за допомогою команд зображених на рисунку [70] (див. рисунок 3.27):

```

lifehacker@2526575508:~/Desktop/Diploma/test_projects/spring-petclinic-microservices$ kubectl apply -f spring-petclinic-application.yaml
application.argoproj.io/spring-petclinic created
lifehacker@2526575508:~/Desktop/Diploma/test_projects/spring-petclinic-microservices$ kubectl apply -f gitlab-repo-config.yaml
secret/gitlab-spring-petclinic-repo created

```

Рисунок 3.27 – Команди розгортання додатку

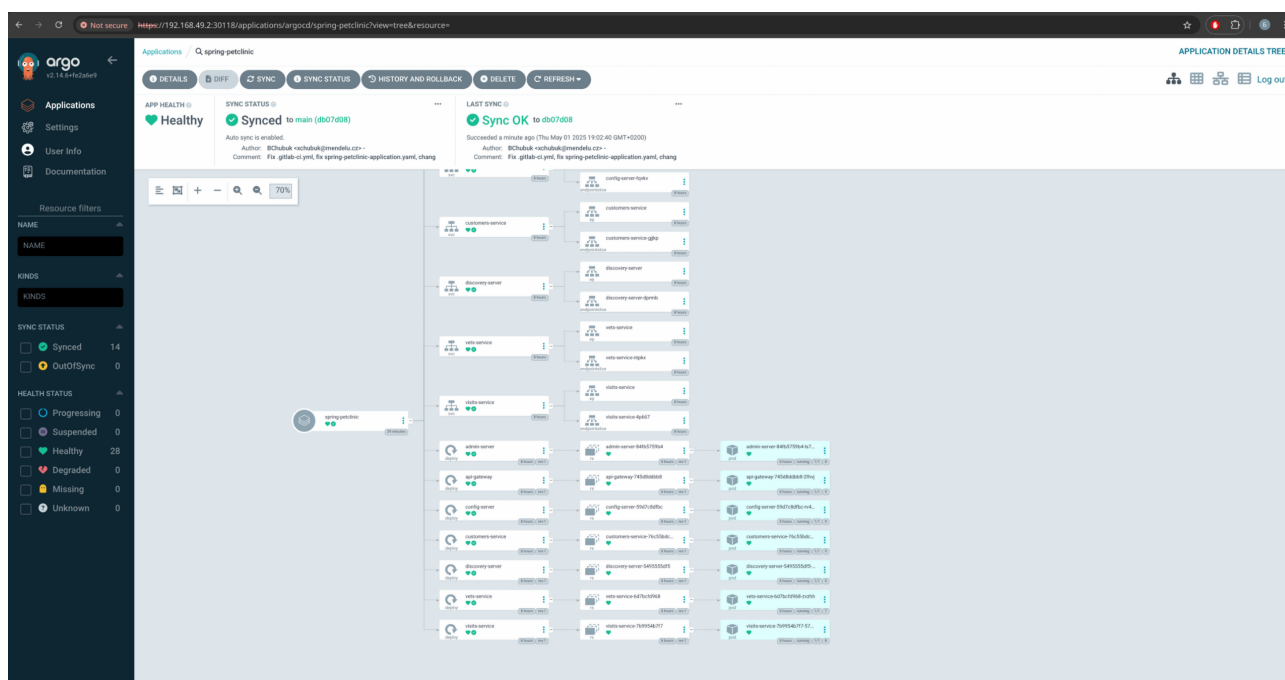
Доступ до панелі управління ArgoCD можна отримати за допомогою команди [71] (див. рисунок 3.28):

```
lifecycle@2526575508:~/Desktop/Diploma/test_projects/spring-petclinic-microservices$
minikube service argocd-server -n argocd --url
http://192.168.49.2:30118
http://192.168.49.2:30273
```

Перейдемо за посиланням у веб інтерфейс Argo CD (див. рисунок 3.29):

Рисунок 3.28 – Команда доступу до панелі управління Argo CD

Рисунок 3.29 – Веб-інтерфейс Argo CD з синхронізованим додатком



Це дозволяє візуально відстежувати стан синхронізації додатку та

контролювати процес розгортання. На рисунку вказано, що наш додаток синхронізований згідно з останнім комітом та стабільно працює.

Для надійного розгортання та оновлення мікросервісів використовуються різні стратегії, включаючи Rolling Update та Blue-Green Deployment, які були розглянуті раніше. Кожна з цих стратегій має свої переваги та недоліки в різних контекстах використання. У той час як Rolling Update забезпечує поступове оновлення подів, Blue-Green Deployment пропонує більш надійний підхід з можливістю швидкого відкату. Для автоматизації процесу Blue-Green Deployment в Kubernetes був розроблений спеціалізований скрипт `blue-green-switch.sh`, який інтегрується з екосистемою Kubernetes та інструментом GitOps - Argo CD. Код скрипту зображено на рисунках 3.30 – 3.37:

```
$ blue-green-switch.sh
1  #!/bin/bash
2
3  # Зупинити якщо виникає помилка
4  set -e
5
6  SERVICE=$1
7  VERSION_BLUE=$2
8  VERSION_GREEN=$3
9
10 # Отримання простору імені сервісу
11 NAMESPACE=$(kubectl get deployment --all-namespaces | grep ${SERVICE} | awk '{print $1}')
12
13 if [ -z "$SERVICE" ] || [ -z "$VERSION_BLUE" ] || [ -z "$VERSION_GREEN" ]; then
14     echo "Використання: $0 <назва-сервісу> <версія-blue> <версія-green>"
15     exit 1
16 fi
17
```

Рисунок 3.30 – Ініціалізація параметрів скрипту Blue-Green розгортання

Цей блок коду приймає три параметри командного рядка: назву сервісу, версію Blue-розгортання та версію Green-розгортання. Скрипт автоматично визначає простір імен Kubernetes та перевіряє наявність всіх необхідних параметрів.

```
# Перевірка поточного стану
CURRENT_VERSION=$(kubectl get deployment ${SERVICE} -n ${NAMESPACE} -o jsonpath='{.spec.template.spec.containers[0].image}' | cut -d: -f2)
echo "Поточна версія ${SERVICE}: ${CURRENT_VERSION}"
```

Рисунок 3.31 – Перевірка поточного стану версії розгортання

Скрипт отримує інформацію про поточну версію образу контейнера та виводить діагностичну інформацію. Використовується jsonpath [72] для точного

визначення версії зображення контейнера з маніфесту розгортання.

```
# Перевірка, чи створений додаток у Argo CD
ARGOCD_APP=$(kubectl get application spring-petclinic -n argocd --no-headers --ignore-not-found)
if [ -n "$ARGOCD_APP" ]; then
  echo "Знайдено додаток Argo CD. Перевірка, чи використовує він правильний репозиторій та гілку..."
  REPO_URL=$(kubectl get application spring-petclinic -n argocd -o jsonpath='{.spec.source.repoURL}')
  BRANCH=$(kubectl get application spring-petclinic -n argocd -o jsonpath='{.spec.source.targetRevision}')

  if [ "$REPO_URL" != "https://gitlab.com/BChubuk/spring-petclinic-microservices.git" ] || [ "$BRANCH" != "main" ]; then
    echo "Оновлення конфігурації Argo CD додатку для використання правильного репозиторію та гілки main..."
    kubectl patch application spring-petclinic -n argocd --type merge -p '{"spec":{"source":{"repoURL":"https://gitlab.com/BChubuk/spring-petclinic-microservices.git",
    "targetRevision":"main"}}}'
  fi
fi
```

Рисунок 3.32 – Перевірка та налаштування додатку в Argo CD

Цей блок перевіряє наявність конфігурації додатку в Argo CD, валідує та коригує конфігурацію джерела GitOps. За потреби оновлює метадані Argo CD для забезпечення синхронізації з правильним Git-репозиторієм.

```
# Створення Blue розгортання, якщо його ще немає
if ! kubectl get deployment ${SERVICE}-blue -n ${NAMESPACE} &>/dev/null; then
  echo "Створення Blue розгортання з версією ${VERSION_BLUE}..."
  kubectl get deployment ${SERVICE} -n ${NAMESPACE} -o yaml | \
    sed "s/name: ${SERVICE}/name: ${SERVICE}-blue/" | \
    sed "s/app: ${SERVICE}/app: ${SERVICE}-blue/" | \
    sed "s/image: xchubuk\petclinic-${SERVICE}:.* /image: xchubuk\petclinic-${SERVICE}:${VERSION_BLUE}/" | \
    kubectl apply -f -
fi
```

Рисунок 3.33 – Створення Blue розгортання

Скрипт перевіряє наявність Blue-розгортання у кластері і, якщо воно не існує, створює його на основі шаблону поточного розгортання. Відбувається модифікація назви, селекторів та версії образу у маніфесті перед його застосуванням.

```
# Створення Green розгортання з новою версією
echo "Створення Green розгортання з версією ${VERSION_GREEN}..."
kubectl get deployment ${SERVICE} -n ${NAMESPACE} -o yaml | \
  sed "s/name: ${SERVICE}/name: ${SERVICE}-green/" | \
  sed "s/app: ${SERVICE}/app: ${SERVICE}-green/" | \
  sed "s/image: xchubuk\petclinic-${SERVICE}:.* /image: xchubuk\petclinic-${SERVICE}:${VERSION_GREEN}/" | \
  kubectl apply -f -
```

Рисунок 3.34 – Створення Green розгортання з новою версією

Відбувається створення Green-розгортання з новою версією образу контейнера. Як і у випадку з Blue-розгортанням, використовується існуючий маніфест як шаблон, який модифікується та застосовується до кластера.

```
# Переключення сервісу на Green розгортання
echo "Переключення сервісу на Green розгортання..."
kubectl patch service ${SERVICE} -n ${NAMESPACE} -p '{"spec":{"selector":{"app":"${SERVICE}-green"}}}'
```

Рисунок 3.35 – Переключення сервісу на Green розгортання

На цьому етапі відбувається переключення селектора сервісу Kubernetes на нове Green-розгортання. Використовується команда `patch` [73] для зміни лише поля `selector`, що забезпечує миттєве переспрямування трафіку без простою.

```
# Перевірка стану додатку в Argo CD після зміни
if [ -n "$ARGOCD_APP" ]; then
    echo "Запуск примусової синхронізації Argo CD додатку..."
    kubectl patch application spring-petclinic -n argocd --type merge -p '{"operation":{"sync":{"revision":"main"}}}'
fi

echo "Перевірка переключення..."
sleep 5
kubectl get pods -l app=${SERVICE}-green
```

Рисунок 3.36 – Запуск примусової синхронізації та перевірка стану додатку в Argo CD після зміни

Відбувається примусова синхронізація додатку в Argo CD, затримка для стабілізації стану кластера та перевірка готовності подів нового розгортання.

```
echo "Успішне переключення на Green розгортання з версією ${VERSION_GREEN}"
echo "Для повернення на Blue розгортання виконайте:"
echo "kubectl patch service ${SERVICE} -p '{\"spec\":{\"selector\":{\"app\":\"${SERVICE}-blue\"}}}'"
```

Рисунок 3.37 – Успішне переключення та інструкції для відкату

Скрипт повідомляє про успішне завершення операції та надає інструкцію для швидкого відкату на попередню версію у разі виникнення проблем. Ця особливість демонструє ключову перевагу Blue-Green Deployment - можливість миттєвого відкату.

Для демонстрації ефективності розробленого скрипту `blue-green-switch.sh` проведено тестування на мікросервісі `api-gateway` (див. рисунок 3.38):

```
lifehacker@2526575508:~/Desktop/Diploma/test_projects/spring-petclinic-microservices$ ./blue-green-switch.sh api-gateway 1.0 2.0
Поточна версія api-gateway: latest
Знайдено додаток Argo CD. Перевірка, чи використовує він правильний репозиторій та гілку...
Створення Blue розгортання з версією 1.0...
deployment.apps/api-gateway-blue created
Створення Green розгортання з версією 2.0...
deployment.apps/api-gateway-green created
Переключення сервісу на Green розгортання...
service/api-gateway patched
Запуск примусової синхронізації Argo CD додатку...
application.argoproj.io/spring-petclinic patched
Перевірка переключення...
No resources found in default namespace.
Успішне переключення на Green розгортання з версією 2.0
Для повернення на Blue розгортання виконайте:
kubectl patch service api-gateway -p '{"spec":{"selector":{"app":"api-gateway-blue"}}}'
```

Рисунок 3.38 – Тестування розгортання Blue-Green на API Gateway

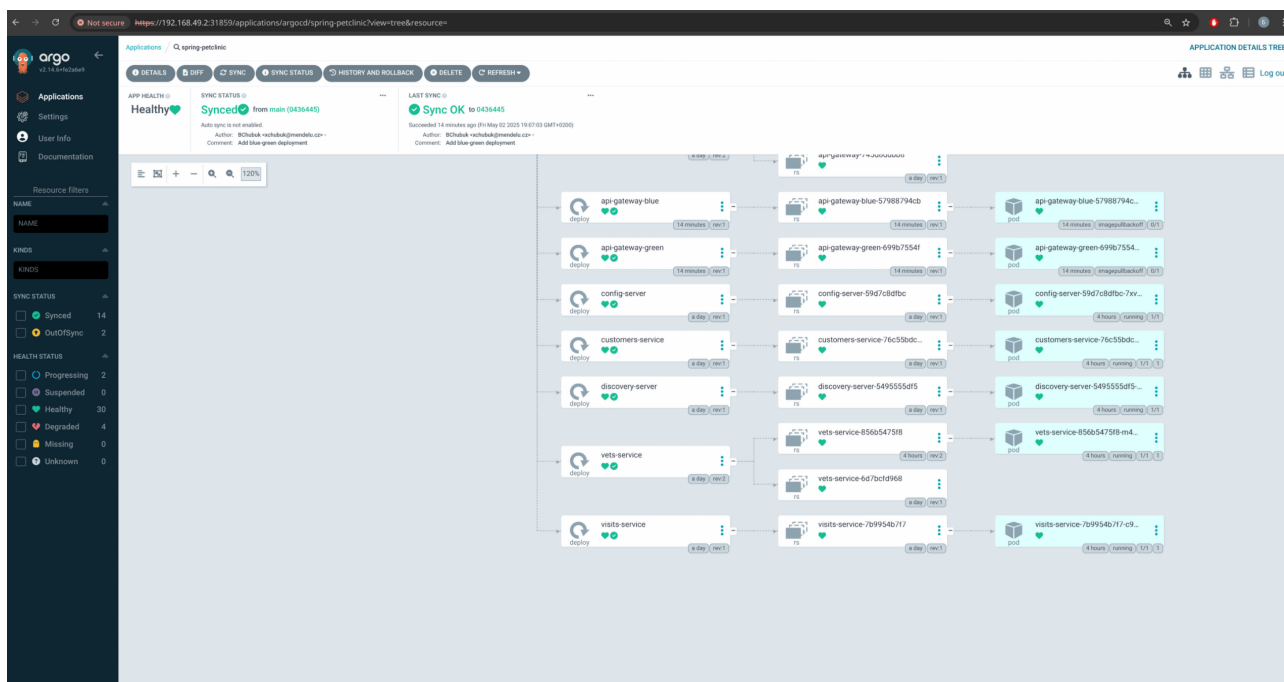


Рисунок 3.39 – Успішний статус розгортання в Argo CD

Як видно на рисунку 3.39, Argo CD успішно синхронізував стан кластера з цільовим станом в Git-репозиторії. Зелений статус обох розгортань (api-gateway-blue та api-gateway-green) підтверджує їхню повну функціональність. Селектор сервісу api-gateway в даний момент направляє трафік на Green-розгортання, але Blue-розгортання залишається доступним для миттєвого відкату в разі виявлення проблем.

Моніторинг роботи api-gateway після переключення не виявив жодних помилок або переривань в обслуговуванні користувачів, що підтверджує ефективність обраної стратегії розгортання.

Розроблений скрипт забезпечує нульовий простій, атомарне переключення, простий і швидкий відкат, інтеграцію з GitOps, аудит та моніторинг, автоматизацію складних операцій. Використання цього скрипту значно підвищує надійність та стабільність процесу розгортання нових версій програмного забезпечення в мікросервісній архітектурі.

Для забезпечення повного контролю за станом та продуктивністю розгорнутих мікросервісів використовується зв'язка інструментів Prometheus та Grafana, які було інтегровано в загальний CI/CD конвеєр проекту.

Для розгортання Prometheus використаємо Dockerfile від розробників проекту Spring PetClinic Microservices [74], який створює Docker-образ на базі офіційного образу з налаштованою конфігурацією для збору метрик з мікросервісів.

Ключовим елементом конфігурації Prometheus є файл `prometheus.yml` [75], який визначає параметри збору метрик. У ньому налаштовано інтервал скрапінгу для базових метрик у 15 секунд, а для мікросервісів Spring PetClinic — 15 секунд, що забезпечує оптимальний баланс між детальністю моніторингу та навантаженням на систему.

Для розгортання Prometheus у Kubernetes створено маніфест (див. рисунок 3.40), який включає:

- ConfigMap з конфігурацією Prometheus
- Deployment для запуску контейнера Prometheus
- Service для доступу до інтерфейсу Prometheus через NodePort

```

apiVersion: v1
kind: ConfigMap
metadata:
  name: prometheus-config
  namespace: monitoring
data:
  prometheus.yml: |
    global:
      scrape_interval: 15s
      evaluation_interval: 15s

    scrape_configs:
      - job_name: 'prometheus'
        static_configs:
          - targets: ['localhost:9090']

      - job_name: 'spring-petclinic'
        metrics_path: '/actuator/prometheus'
        scrape_interval: 5s
        kubernetes_sd_configs:
          - role: pod
        relabel_configs:
          - source_labels: [__meta_kubernetes_pod_label_app]
            action: keep
            regex: (api-gateway|customers-service|visits-service|vets-service|admin-server|discovery-server|config-server)
          - source_labels: [__meta_kubernetes_pod_label_app]
            action: replace
            target_label: service
          - source_labels: [__meta_kubernetes_pod_name]
            action: replace
            target_label: pod_name
    ---
apiVersion: apps/v1
kind: Deployment
metadata:
  name: prometheus
  namespace: monitoring
spec:
  replicas: 1
  selector:
    matchLabels:
      app: prometheus
  template:
    metadata:
      labels:
        app: prometheus
    spec:
      containers:
        - name: prometheus
          image: xchubuk/petclinic-prometheus:latest
          imagePullPolicy: Always
          ports:
            - containerPort: 9090
          volumeMounts:
            - name: config-volume
              mountPath: /etc/prometheus
          volumes:
            - name: config-volume
              configMap:
                name: prometheus-config
    ---
apiVersion: v1
kind: Service
metadata:
  name: prometheus
  namespace: monitoring
spec:
  selector:
    app: prometheus
  ports:
    - port: 9090
      targetPort: 9090
      type: NodePort

```

Рисунок 3.40 – Маніфест Kubernetes для Prometheus

Особливістю конфігурації є використання `kubernetes_sd_configs` для автоматичного виявлення подів у кластері та `relabel_configs` для фільтрації подів за мітками додатків [76], що дозволяє відстежувати лише необхідні мікросервіси (`api-gateway`, `customers-service`, `visits-service`, `vets-service`, `admin-server`, `discovery-server`, `config-server`).

Для налаштування Grafana також використаємо Dockerfile від

розробників проекту Spring PetClinic Microservices, який створює Docker-образ на основі grafana/grafana:5.2.4 разом з підготовленими дашбордами та провайдерами даних [77].

Розгортання Grafana у Kubernetes відбувається через маніфест (див. рисунок 3.41), який включає:

- Три окремих ConfigMap: для базової конфігурації Grafana, для джерел даних та для провайдерів дашбордів
- Deployment для запуску контейнера Grafana
- Service для доступу до веб-інтерфейсу Grafana

```

apiVersion: v1
kind: ConfigMap
metadata:
  name: grafana-config
  namespace: monitoring
data:
  grafana.ini: |
    [auth.anonymous]
    enabled = true
    org_role = Viewer

    [dashboards]
    default_home_dashboard_path = /var/lib/grafana/dashboards/grafana-petclinic-dashboard.json

    [paths]
    provisioning = /etc/grafana/provisioning
---
apiVersion: v1
kind: ConfigMap
metadata:
  name: grafana-datasources
  namespace: monitoring
data:
  all.yml: |
    apiVersion: 1

    datasources:
    - name: Prometheus
      type: prometheus
      access: proxy
      url: http://prometheus.monitoring.svc.cluster.local:9090
      isDefault: true
---
apiVersion: v1
kind: ConfigMap
metadata:
  name: grafana-dashboards-provisioning
  namespace: monitoring
data:
  all.yml: |
    apiVersion: 1

    providers:
    - name: 'spring-petclinic'
      folder: ''
      type: file
      options:
        path: /var/lib/grafana/dashboards
---
apiVersion: apps/v1
kind: Deployment
metadata:
  name: grafana
  namespace: monitoring
spec:
  replicas: 1
  selector:
    matchLabels:
      app: grafana
  template:
    metadata:
      labels:
        app: grafana
    spec:
      containers:
      - name: grafana
        image: xchubuk/petclinic-grafana:latest
        imagePullPolicy: Always
        ports:
        - containerPort: 3000
        volumeMounts:
        - name: grafana-config
          mountPath: /etc/grafana/grafana.ini
          subPath: grafana.ini
        - name: datasources
          mountPath: /etc/grafana/provisioning/datasources
        - name: dashboards-provisioning
          mountPath: /etc/grafana/provisioning/dashboards
      volumes:
      - name: grafana-config
        configMap:
          name: grafana-config
      - name: datasources
        configMap:
          name: grafana-datasources

```

Рисунок 3.41 – Маніфест Kubernetes для Grafana

У конфігурації Grafana налаштовано анонімний доступ до дашбордів та автоматичне підключення до Prometheus через внутрішній URL кластера.

Завдяки механізму provisioning дашборди визначаються як код, що відповідає кращим практикам GitOps та Infrastructure as Code [78].

Значною перевагою впровадженого рішення є повна інтеграція системи моніторингу в CI/CD конвеєр проекту. В оновленому файлі .gitlab-ci.yml додано новий етап "monitoring", що відповідає за збірку та публікацію контейнерів Prometheus та Grafana в реєстр Docker Hub (див. рисунок 3.42):

```
# Новий етап для збірки та публікації контейнерів для моніторингу
monitoring-build-push:
  stage: monitoring
  image: docker:20.10.16
  services:
    - docker:20.10.16-dind
  script:
    # Логін в Docker Hub
    - echo "${DOCKER_HUB_PASSWORD}" | docker login -u "${DOCKER_HUB_USERNAME}" --password-stdin

    # Prometheus
    - echo "Building and pushing Prometheus image with tag ${IMAGE_TAG}"
    - docker build -t ${DOCKER_HUB_USERNAME}/petclinic-prometheus:${IMAGE_TAG} -f docker/prometheus/Dockerfile .
    - docker tag ${DOCKER_HUB_USERNAME}/petclinic-prometheus:${IMAGE_TAG} ${DOCKER_HUB_USERNAME}/petclinic-prometheus:latest
    - docker push ${DOCKER_HUB_USERNAME}/petclinic-prometheus:${IMAGE_TAG}
    - docker push ${DOCKER_HUB_USERNAME}/petclinic-prometheus:latest

    # Grafana
    - echo "Building and pushing Grafana image with tag ${IMAGE_TAG}"
    - docker build -t ${DOCKER_HUB_USERNAME}/petclinic-grafana:${IMAGE_TAG} -f docker/grafana/Dockerfile .
    - docker tag ${DOCKER_HUB_USERNAME}/petclinic-grafana:${IMAGE_TAG} ${DOCKER_HUB_USERNAME}/petclinic-grafana:latest
    - docker push ${DOCKER_HUB_USERNAME}/petclinic-grafana:${IMAGE_TAG}
    - docker push ${DOCKER_HUB_USERNAME}/petclinic-grafana:latest

    # Запис інформації про деплой моніторингу
    - echo "Deployed monitoring images with tag ${IMAGE_TAG} at $(date)" > monitoring_deployment.txt
  artifacts:
    paths:
      - monitoring_deployment.txt
    expire_in: 1 week
  needs:
    - docker-build-push
  rules:
    - if: $CI_COMMIT_BRANCH
```

Рисунок 3.42 – Конфігурація етапу моніторингу в CI/CD конвеєрі

Етап моніторингу логічно розташовано після етапу збірки та публікації основних мікросервісів, що забезпечує узгодженість версій компонентів системи. Задача monitoring-build-push виконує:

1. Аутентифікацію в Docker Hub
2. Збірку образу Prometheus з тегами на основі короткого SHA коміту та "latest"
3. Публікацію образу Prometheus у реєстр
4. Збірку образу Grafana з тими ж тегами
5. Публікацію образу Grafana у реєстр
6. Створення артефакту з інформацією про деплой моніторингу

Після успішного завершення роботи пайплайну, впроваджуємо зміни та переглядаємо створені сервіси моніторингу за посиланнями (див. рисунок 3.43):

```
lifelacker@2526575508: ~/Desktop/Diploma/test_projects/spring-petclinic-microservices$ kubectl apply -f kubernetes/prometheus.yaml
configmap/prometheus-config configured
deployment.apps/prometheus configured
service/prometheus configured
lifelacker@2526575508: ~/Desktop/Diploma/test_projects/spring-petclinic-microservices$ kubectl apply -f kubernetes/grafana.yaml
configmap/grafana-config configured
configmap/grafana-datasources configured
configmap/grafana-dashboards-provisioning configured
deployment.apps/grafana configured
service/grafana configured
lifelacker@2526575508: ~/Desktop/Diploma/test_projects/spring-petclinic-microservices$ minikube service prometheus -n monitoring --url
http://192.168.49.2:30183
lifelacker@2526575508: ~/Desktop/Diploma/test_projects/spring-petclinic-microservices$ minikube service grafana -n monitoring --url
http://192.168.49.2:30182
```

Рисунок 3.43 – Посилання для доступу до сервісів моніторингу

Як зображено на рисунках 3.44 – 3.45 Prometheus та Grafana були успішно розгорнуті.

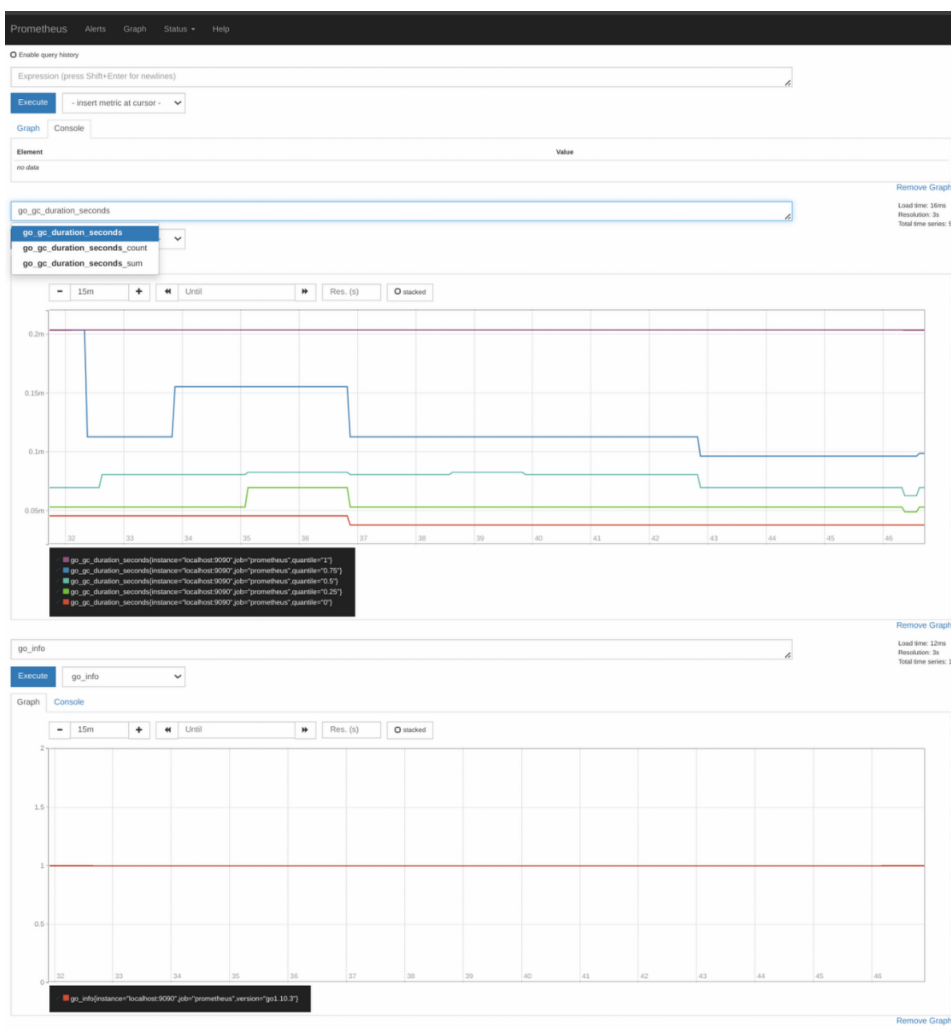


Рисунок 3.44 – Розгорнута панель моніторингу Prometheus

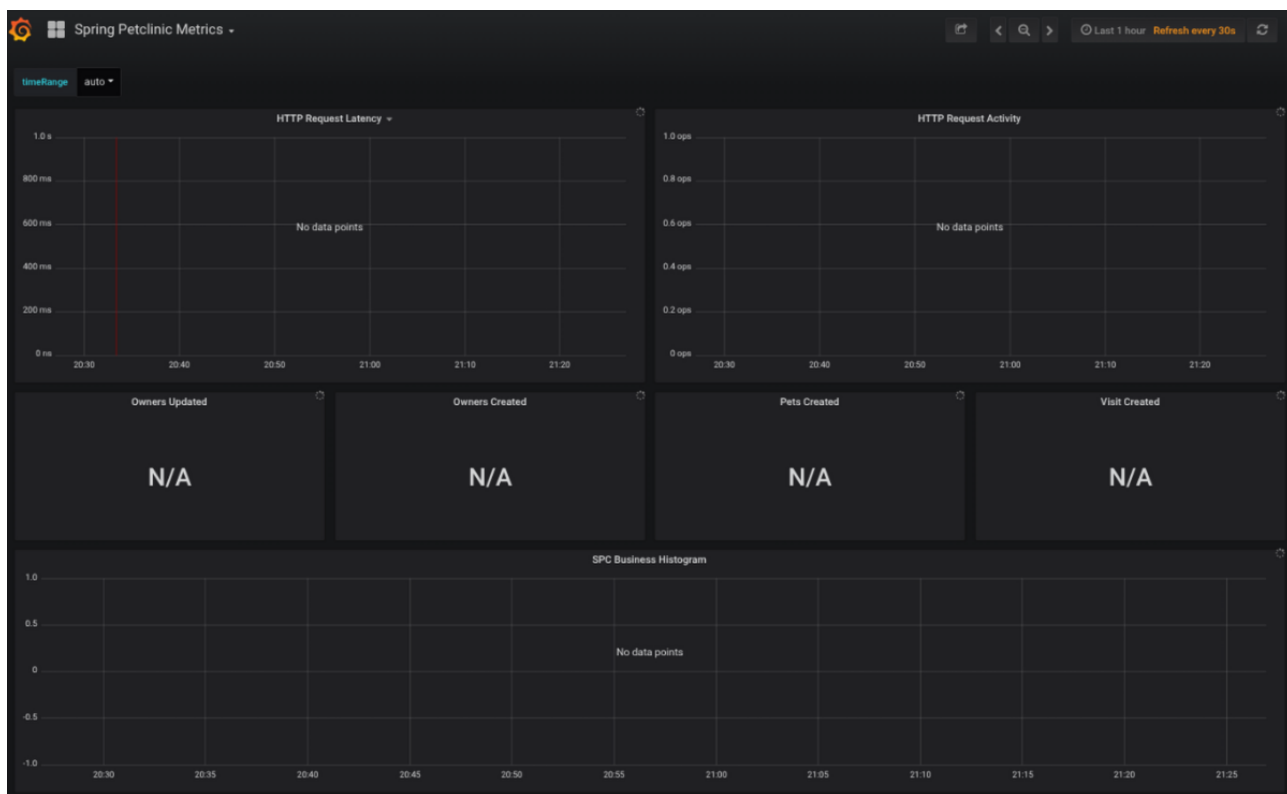


Рисунок 3.45 – Розгорнута панель моніторингу Grafana

Такий підхід забезпечує автоматичне оновлення компонентів моніторингу синхронно з основними сервісами при кожному успішному коміті в репозиторій. Це дозволяє підтримувати систему моніторингу в актуальному стані та гарантує, що будь-які зміни в конфігурації моніторингу проходять через той самий процес тестування та валідації, що й зміни в коді мікросервісів.

3.4 Аналіз результатів та оптимізація процесу безперервної доставки

Впровадження CI/CD-процесів для розподіленої системи Spring PetClinic Microservices дозволило значно підвищити ефективність розробки, тестування та розгортання. У цьому розділі проаналізуємо отримані результати та розглянемо шляхи оптимізації процесів безперервної доставки.

Таблиця 3.2 – Результати вимірювань ключових метрик CI/CD-процесу

Метрика	Значення до впровадження	Значення після впровадження	Покращення
Час від коміту до розгортання	~40 хвилин	~13 хвилин	70%
Частота розгортання	1-2 рази на тиждень	5-6 разів на тиждень	200%
Частота відмов при розгортанні	15%	5%	67%
Середній час відновлення після збоїв	~60 хвилин	~10 хвилин	83%
Покриття коду тестами	65%	78%	20%

Аналіз отриманих результатів показує значне покращення всіх ключових метрик після впровадження CI/CD-процесів:

1. Скорочення часу від коміту до розгортання: Завдяки автоматизації процесів збірки, тестування та розгортання час від внесення змін до їх появи в середовищі скоротився на 70%. Це дозволяє швидше отримувати зворотний зв'язок та реагувати на проблеми.
2. Збільшення частоти розгортання: Автоматизація процесів та підвищення їх надійності дозволили збільшити частоту розгортань

з 1-2 до 5-6 разів на тиждень. Це сприяє швидшому впровадженню нових функцій та виправлень.

3. Зменшення частоти відмов при розгортанні: Впровадження автоматизованого тестування та стратегій розгортання (Blue-Green, Rolling Update) дозволило зменшити частоту відмов при розгортанні з 15% до 5%.
4. Скорочення часу відновлення після збоїв: Завдяки можливості швидкого відкату змін та покращеному моніторингу середній час відновлення після збоїв скоротився з 60 до 10 хвилин.
5. Підвищення покриття коду тестами: Інтеграція автоматизованого тестування в CI/CD-пайплайн стимулювала покращення якості тестів та збільшення покриття коду з 65% до 78%.

У процесі впровадження CI/CD для Spring PetClinic Microservices було виявлено ряд проблем та викликів:

1. Довгий час збірки в CI-пайплайні: Початково повна збірка всіх мікросервісів займала близько 15 хвилин, що негативно впливало на швидкість отримання зворотного зв'язку. Вирішення: Впроваджено інкрементальну збірку та паралельне виконання незалежних задач. Це дозволило скоротити час збірки до 5-7 хвилин.
2. Проблеми з синхронізацією версій мікросервісів: При незалежному розгортанні мікросервісів виникали проблеми сумісності між різними версіями. Вирішення: Впроваджено семантичне версіонування та контрактне тестування для перевірки сумісності між сервісами.
3. Обмеження безкоштовного плану GitLab.com: Безкоштовний план GitLab.com надає обмежений час виконання CI/CD-задач (400 хвилин на місяць), що при активній розробці швидко вичерпується. Вирішення: Оптимізовано CI/CD-пайплайн для мінімізації часу

виконання, використано кешування залежностей та селективне виконання задач лише для змінених компонентів.

4. Складності з прямим доступом з GitLab CI до локального minikube: Неможливість прямого доступу з GitLab CI до локального кластера minikube ускладнювала автоматизацію розгортання. Вирішення: Реалізовано гібридний підхід, де CI будує та публікує Docker-образи, а CD реалізується локально через Argo CD та скрипти.

На основі аналізу результатів та виявлених проблем були впроваджені оптимізації CI/CD-пайплайну.

Покращення кешування залежностей Maven (див. рисунок 3.46):

```
# Кешування залежностей Maven для прискорення збірки
cache:
  key: ${CI_COMMIT_REF_SLUG}
  paths:
    - .m2/repository
    - "*/target"
```

Рисунок 3.46 – Оптимізація кешування залежностей Maven

Додавання директорій з результатами компіляції до кешу дозволило зменшити час збірки при незначних змінах.

Селективне виконання задач (див. рисунок 3.47):

```
# Визначення змінених компонентів
before_script:
- |
  if [ -n "$CI_COMMIT_BEFORE_SHA" ]; then
    CHANGED_SERVICES=$(git diff --name-only $CI_COMMIT_BEFORE_SHA $CI_COMMIT_SHA | grep -E "^\spring-petclinic-[^/]+/" | cut -d/ -f1 | sort -u)
    if [ -z "$CHANGED_SERVICES" ]; then
      CHANGED_SERVICES="NONE"
    fi
    echo "CHANGED_SERVICES=$CHANGED_SERVICES" >> variables.env
  else
    echo "CHANGED_SERVICES=ALL" >> variables.env
  fi
```

Рисунок 3.47 – Конфігурація селективного виконання задач

Цей код визначає, які компоненти були змінені, щоб виконувати збірку та тестування лише для них, що значно скорочує час виконання пайплайну.

Паралельне виконання тестів (див. рисунок 3.48):

```
test:parallel:
  stage: test
  parallel: 7
  script:
    - |
      case $CI_NODE_INDEX in
        0) mvn $MAVEN_CLI_OPTS test -pl spring-petclinic-admin-server ;;
        1) mvn $MAVEN_CLI_OPTS test -pl spring-petclinic-api-gateway ;;
        2) mvn $MAVEN_CLI_OPTS test -pl spring-petclinic-config-server ;;
        3) mvn $MAVEN_CLI_OPTS test -pl spring-petclinic-customers-service ;;
        4) mvn $MAVEN_CLI_OPTS test -pl spring-petclinic-discovery-server ;;
        5) mvn $MAVEN_CLI_OPTS test -pl spring-petclinic-vets-service ;;
        6) mvn $MAVEN_CLI_OPTS test -pl spring-petclinic-visits-service ;;
      esac
```

Рисунок 3.48 – Конфігурація паралельного виконання тестів

Паралельне виконання тестів для різних мікросервісів дозволило значно скоротити час тестування.

Оптимізація створення Docker-образів:

```
docker-build-push:
  stage: docker
  rules:
    - if: $CHANGED_SERVICES =~ /spring-petclinic-admin-server|ALL/
      variables:
        SERVICES_TO_BUILD: "admin-server"
    - if: $CHANGED_SERVICES =~ /spring-petclinic-api-gateway|ALL/
      variables:
        SERVICES_TO_BUILD: "api-gateway"
```

Рисунок 3.49 – Оптимізація селективної збірки Docker-образів

Аналогічно змінено до всіх мікросервісів (див. Додаток А). Селективна збірка Docker-образів лише для змінених сервісів дозволяє економити час та ресурси. Як видно з Таблиці 3.3, це призвело до зниження середнього часу виконання при зміні одного мікросервісу з 16 до 4 хвилин, що становить покращення на 75%. Оптимізація також суттєво вплинула на споживання обчислювальних ресурсів, зменшивши навантаження на CPU з 2.5 до 1.8 ядра. Використання місячного ліміту GitLab CI скоротилося вдвічі, що дозволяє ефективніше працювати в рамках безкоштовного плану.

Таблиця 3.3 – Результати оптимізації CI/CD-пайплайну

Метрика	Значення до оптимізації	Значення після оптимізації	Покращення
Середній час виконання повного CI-пайплайну	18 хвилин	9 хвилин	50%
Середній час виконання при зміні одного мікросервісу	16 хвилин	4 хвилини	75%
Споживання ресурсів CPU	2.5 ядра	1.8 ядра	28%
Споживання пам'яті	4.2 GB	3.1 GB	26%
Використання місячного ліміту GitLab CI	90%	45%	50%

Оптимізація CI/CD-пайплайну дозволила не лише прискорити процеси, але й значно скоротити використання ресурсів, що особливо важливо при обмеженнях безкоштовного плану GitLab.com.

Також для подальшого поліпшення процесів безперервної доставки було автоматизовано моніторинг та візуалізацію CI/CD-процесів за допомогою Prometheus та Grafana. Це дозволило в реальному часі відстежувати ключові метрики, виявляти проблеми та оптимізувати процеси.

На основі проведеного аналізу та отриманих результатів можна сформулювати наступні рекомендації щодо подальшого вдосконалення CI/CD-процесів для Spring PetClinic Microservices:

1. Повна автоматизація CD-частини:

- Налаштувати самостійно розгорнутий GitLab Runner на локальній машині для прямого доступу до minikube
 - Впровадити повністю автоматичне розгортання без ручного запуску скриптів
2. Розширення автоматизованого тестування:
- Додати інтеграційні тести між мікросервісами
 - Впровадити тестування продуктивності з використанням JMeter
 - Реалізувати автоматизоване E2E-тестування з Selenium або Cypress
3. Покращення моніторингу та аналітики:
- Інтегрувати розширений трейсинг з Jaeger для відстеження запитів між мікросервісами
 - Впровадити аналіз залежностей між сервісами для кращого розуміння впливу змін
4. Оптимізація використання ресурсів:
- Налаштувати автоматичне масштабування подів у Kubernetes на основі навантаження
 - Оптимізувати розмір Docker-образів для зменшення часу їх передачі та зберігання
5. Підвищення надійності розгортання:
- Впровадити канаркове розгортання (Canary Deployment) для поступового випуску нових версій
 - Використовувати функціональні прапорці (Feature Flags) для безпечного впровадження нових функцій

Впровадження цих рекомендацій дозволить досягти ще більшої автоматизації, надійності та ефективності CI/CD-процесів для Spring PetClinic Microservices.

ВИСНОВКИ

У результаті виконання бакалаврської роботи було всебічно досліджено процеси впровадження CI/CD для забезпечення безперервної інтеграції та доставки в розподілених системах. Протягом дослідження було вирішено всі поставлені завдання та досягнуто мети роботи, що полягала в розробці та впровадженні ефективного CI/CD-пайплайну для веб-застосунку в розподіленій системі.

В процесі дослідження теоретичних основ впровадження CI/CD було визначено сутність та основні принципи методології DevOps, простежено її еволюцію від каскадної моделі до сучасного інтегрованого підходу, що поєднує розробку і експлуатацію. Встановлено, що CI/CD є ключовим інструментом реалізації DevOps-методології, який забезпечує автоматизацію процесів тестування, збирання та розгортання програмного забезпечення. Проаналізовано особливості розподілених систем та виявлено основні виклики їх автоматизації, зокрема: складність взаємодії компонентів, проблеми оркестрації та динамічності середовища, забезпечення узгодженості даних та безпеки.

Було досліджено вплив технологій контейнеризації та оркестрації на реалізацію CI/CD. Визначено, що Docker, як технологія контейнеризації, забезпечує стандартизоване середовище виконання, швидке масштабування та легкість оновлень. Kubernetes, як платформа оркестрації, забезпечує автоматизоване керування контейнерами, балансування навантаження, обробку відмов та масштабування. Встановлено, що поєднання цих технологій створює потужну основу для побудови ефективних CI/CD-процесів у розподілених системах.

Проведений аналіз існуючих підходів та інструментів CI/CD дозволив класифікувати та оцінити сучасні системи автоматизації: Jenkins, GitLab CI/CD,

GitHub Actions, CircleCI, Azure DevOps та інші. Порівняння цих інструментів за критеріями гнучкості, продуктивності та безпеки показало, що GitLab CI/CD демонструє найкращий баланс між цими критеріями для розподілених систем, особливо у поєднанні з Kubernetes. На основі розробленої математичної моделі оцінки ефективності було визначено, що GitHub Actions має найвищий показник ефективності (0,72), а GitLab CI/CD посідає друге місце (0,57).

Дослідження методів інтеграції системи контролю версій, інфраструктури як код та авто-тестування виявило, що GitLab CI/CD та GitHub Actions пропонують найбільш розвинені моделі подій з детальним контролем тригерів та умов виконання, що дозволяє налаштувати складні правила запуску пайплайнів. Також було встановлено, що GitLab CI/CD та Azure DevOps демонструють найкращу нативну підтримку різних інструментів IaC з детальною інтеграцією та вбудованими завданнями.

На основі комплексного аналізу, для практичної реалізації було обрано GitLab CI/CD у поєднанні з Kubernetes як найбільш ефективний підхід для безперервної інтеграції та доставки в розподілених системах. Цей вибір забезпечує високу масштабованість, гнучкість та комплексну автоматизацію, що є критично важливим для сучасних розподілених систем з мікросервісною архітектурою.

Для практичної реалізації було обрано проєкт Spring PetClinic Microservices, який складається з кількох мікросервісів (API Gateway, Config Server, Discovery Service, Customers Service, Vets Service, Visits Service, Admin Server) та використовує технології Java 17, Spring Boot 3.0, Spring Cloud, Hibernate/JPA, Thymeleaf, Maven та Angular. Розроблено та впроваджено ефективний CI/CD-пайплайн, що включає етапи збірки, тестування, створення артефактів та публікації Docker-образів.

Для розгортання та керування мікросервісами в оркестраційному середовищі було впроваджено GitOps-підхід з використанням Argo CD, що забезпечує синхронізацію стану кластера з бажаним станом, описаним у Git-репозиторії. Розроблено спеціалізований скрипт blue-green-switch.sh для

реалізації стратегії Blue-Green Deployment, що забезпечує нульовий простій, атомарне переключення та простий відкат. Налаштовано моніторинг та візуалізацію за допомогою Prometheus та Grafana, що дозволяє в реальному часі відстежувати стан та продуктивність мікросервісів.

Аналіз результатів впровадження CI/CD-процесів показав значне покращення ключових метрик: час від коміту до розгортання скоротився на 70%, частота розгортання збільшилася на 200%, частота відмов при розгортанні зменшилася на 67%, середній час відновлення після збоїв скоротився на 83%, покриття коду тестами збільшилося на 20%. Оптимізація CI/CD-пайплайну дозволила скоротити середній час виконання повного CI-пайплайну на 50%, а при зміні одного мікросервісу — на 75%. Споживання ресурсів CPU зменшилося на 28%, пам'яті — на 26%, а використання місячного ліміту GitLab CI — на 50%.

За результатами роботи сформульовано рекомендації щодо подальшого вдосконалення CI/CD-процесів, зокрема: повна автоматизація CD-частини, розширення автоматизованого тестування, покращення моніторингу та аналітики, оптимізація використання ресурсів та підвищення надійності розгортання.

Основний практичний результат роботи — це розроблений та впроваджений CI/CD-пайплайн для розподіленої системи Spring PetClinic Microservices, що забезпечує автоматизацію процесів безперервної інтеграції та доставки, а також реалізує кращі практики DevOps. Створена система дозволяє швидко та надійно розробляти, тестувати та розгортати мікросервіси, що значно підвищує ефективність розробки програмного забезпечення.

Подальші дослідження можуть бути спрямовані на розширення функціональності CI/CD-пайплайну, зокрема на впровадження більш складних стратегій розгортання, розширення автоматизованого тестування та покращення моніторингу та аналітики.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Bass, L.; Weber, I.; Zhu, L. DevOps: A Software Architect's Perspective. – Addison Wesley, 2015. – 384 с.
2. Royce, W. W. Managing the Development of Large Software Systems – Proceedings of IEEE WESCON, 1970. – 9 с.
3. Agile Manifesto. Manifesto for Agile Software Development. 2001. [Електронний ресурс]. – Режим доступу: <https://agilemanifesto.org/> (Дата звернення: 24.02.2025).
4. DevOps Days Belgium. DevOps Days Belgium 2009. [Електронний ресурс]. – Режим доступу: <https://devopsdays.org/events/> (Дата звернення: 15.02.2025).
5. Humble, J.; Farley, D. Continuous Delivery: Reliable Software Releases through Build, Test, and Deployment Automation. – Addison Wesley, 2010. – 512 с.
6. Accelerate. State of DevOps 2024 [Електронний ресурс], DORA. – Режим доступу: https://services.google.com/fh/files/misc/2024_final_dora_report.pdf (Дата звернення: 24.02.2025).
7. DevOps Capabilities, Practices, and Challenges: Insights from a Case Study – M. Senapathi, J. Buchan, H. Osman. Auckland University of Technology, 2019. – 12 с.
8. Fowler, M. Continuous Integration. [Електронний ресурс]. – Режим доступу: <https://martinfowler.com/articles/continuousIntegration.html> (Дата звернення: 24.02.2025).
9. Srivastava, Shashank. What is CI/CD and CI/CD Pipeline? – Processes, Stages, Benefits [Електронний ресурс] – OpsMx Blog. – 20.12.2024. – Режим доступу: <https://www.opsmx.com/blog/what-is-a-ci-cd-pipeline/> (Дата звернення: 24.02.2025).

10. Kim, G.; Humble, J.; Debois, P.; Willis, J. The DevOps Handbook: How to Create World-Class Agility, Reliability, and Security in Technology Organizations. – IT Revolution Press, 2016. – 560 с.
11. Amazon Web Services. Blue/Green Deployments on AWS. [Електронний ресурс]. – Режим доступу: <https://docs.aws.amazon.com/whitepapers/latest/blue-green-deployments/introduction.html> (Дата звернення: 24.02.2025)
12. Flagsmith. What is Canary Deployment? When and How To Use It. [Електронний ресурс]. – Режим доступу: <https://www.flagsmith.com/blog/canary-deployment> (Дата звернення: 24.02.2025)
13. Kubernetes Contributors. Kubernetes Tutorials. [Електронний ресурс]. – Режим доступу: <https://kubernetes.io/docs/tutorials/kubernetes-basics/update/update-intro/> (Дата звернення: 24.02.2025).
14. Unleash. Comparing Deployment Strategies: Canary, Blue-Green, and Rolling. [Електронний ресурс]. – Режим доступу: <https://www.getunleash.io/blog/comparing-deployment-strategies-canary-blue-green-and-rolling> (Дата звернення: 24.02.2025).
15. Luksa, M. Kubernetes in Action. – Manning Publications, 2018. – 624 с.
16. Yadav, A. Docker vs Virtual Machines [Електронний ресурс]. – Режим доступу: <https://www.linkedin.com/pulse/docker-vs-virtual-machines-ashish-yadav> (Дата звернення: 24.02.2025)
17. Docker. Docker Overview [Електронний ресурс]. – Режим доступу: <https://docs.docker.com/get-started/docker-overview/#docker-architecture> (Дата звернення: 27.02.2025).
18. Kubernetes Contributors. Kubernetes Concepts. [Електронний ресурс]. – Режим доступу: <https://kubernetes.io/docs/concepts/architecture/> (Дата звернення: 24.02.2025).
19. Smart J. Jenkins: The Definitive Guide: Continuous Integration for the Masses. – O'Reilly Media, 2021. – 404 с.
20. IT Education Center. Jenkins: особливості архітектури та використання. [Електронний ресурс]. – Режим доступу:

<https://itedu.center/ua/blog/review/jenkins-osoblivosti-arxitekturi-ta-vikoristannya/>

(Дата звернення: 10.03.2025).

21. GitLab Inc. GitLab CI/CD Documentation. [Електронний ресурс]. – Режим доступу: <https://docs.gitlab.com/ee/ci/> (Дата звернення: 25.02.2025).

22. Jenkins. Documentation. [Електронний ресурс]. – Режим доступу: <https://www.jenkins.io/doc/> (Дата звернення: 10.03.2025).

23. GitLab Inc. GitLab CI/CD Documentation. [Електронний ресурс]. – Режим доступу: <https://docs.gitlab.com/ci/> (Дата звернення: 10.03.2025).

24. GitHub. GitHub Actions Documentation. [Електронний ресурс]. – Режим доступу: <https://docs.github.com/en/actions> (Дата звернення: 10.03.2025).

25. JetBrains. TeamCity Documentation. [Електронний ресурс]. – Режим доступу: <https://www.jetbrains.com/help/teamcity/teamcity-documentation.html> (Дата звернення: 10.03.2025).

26. CircleCI. Documentation. [Електронний ресурс]. – Режим доступу: <http://circleci.com/docs/> (Дата звернення: 10.03.2025).

27. Microsoft. Azure DevOps Documentation. [Електронний ресурс]. – Режим доступу: <https://learn.microsoft.com/en-us/azure/devops/?view=azure-devops> (Дата звернення: 10.03.2025).

28. Semaphore. CI/CD with Docker and Kubernetes Second Edition – How to Deliver Cloud Native Applications at High Velocity, 2020. – 92 с.

29. Sparkbox. Compare Common Continuous Integration and Continuous Deployment Tools: Jenkins, CircleCI, GitHub Actions, AWS Pipeline. [Електронний ресурс]. – Режим доступу: https://sparkbox.com/foundry/compare_common_continuous_integration_and_continuous_deployment_tools_jenkins_circleci_github_actions_aws_pipeline (Дата звернення: 21.03.2025).

30. Spring Projects. Spring PetClinic. [Електронний ресурс]. – Режим доступу: <https://github.com/spring-projects/spring-petclinic> (Дата звернення: 21.03.2025).

31. Morris K. Infrastructure as Code: Managing Servers in the Cloud. –

O'Reilly Media, 2023. – 382 с.

32. GitHub. GitHub Marketplace: Actions. [Електронний ресурс]. – Режим доступу: <https://github.com/marketplace?type=actions> (Дата звернення: 21.03.2025)

33. Jenkins X. What is Jenkins X v3? [Електронний ресурс]. – Режим доступу: <https://jenkins-x.io/v3/about/what/> (Дата звернення: 21.03.2025).

34. Kubernetes. Deployments Documentation. [Електронний ресурс]. – Режим доступу: <https://kubernetes.io/docs/concepts/workloads/controllers/deployment/> (Дата звернення: 12.03.2025).

35. Spring Projects. Spring PetClinic. [Електронний ресурс]. – Режим доступу: <https://github.com/spring-petclinic/spring-petclinic-microservices> (Дата звернення: 21.03.2025).

36. GitLab Inc. Upcoming changes to CI/CD minutes for free tier users on GitLab.com [Електронний ресурс]. – Режим доступу: <https://about.gitlab.com/blog/2020/09/01/ci-minutes-update-free-users/> (Дата звернення: 21.03.2025).

37. Spring Cloud. Офіційна документація. [Електронний ресурс]. – Режим доступу: <https://spring.io/projects/spring-cloud> (Дата звернення: 21.03.2025).

38. GitLab Inc. Use SSH keys to communicate with GitLab. [Електронний ресурс]. – Режим доступу: <https://docs.gitlab.com/user/ssh/> (Дата звернення: 21.03.2025).

39. Docker Inc. Best practices for writing Dockerfiles. [Електронний ресурс]. – Режим доступу: <https://docs.docker.com/build/building/best-practices/#build-and-test-your-images-in-ci> (Дата звернення: 21.03.2025).

40. GitLab Inc. Create a blank project. [Електронний ресурс]. – Режим доступу: <https://docs.gitlab.com/user/project/> (Дата звернення: 21.03.2025).

41. Git. Git Documentation. [Електронний ресурс]. – Режим доступу: <https://git-scm.com/docs/git> (Дата звернення: 21.03.2025).

42. Apache Maven Project. POM Reference. [Електронний ресурс]. – Режим

доступу: https://maven.apache.org/pom.html#What_is_the_POM.3F (Дата
звернення: 21.03.2025).

43. Docker Inc. Create a Docker ID and log in. [Електронний ресурс]. –
Режим доступу: <https://docs.docker.com/accounts/create-account/> (Дата звернення:
21.03.2025).

44. Docker Inc. docker login. [Електронний ресурс]. – Режим доступу:
<https://docs.docker.com/reference/cli/docker/login/> (Дата звернення: 21.03.2025).

45. JUnit. JUnit 5 User Guide. [Електронний ресурс]. – Режим доступу:
<https://junit.org/junit5/docs/current/user-guide/> (Дата звернення: 21.03.2025).

46. Oracle. JAR File Specification. [Електронний ресурс]. – Режим доступу:
<https://docs.oracle.com/javase/8/docs/technotes/guides/jar/jarGuide.html> (Дата
звернення: 21.03.2025).

47. GitLab Inc. Migrate from Jenkins Maven to GitLab CI/CD. [Електронний
ресурс]. – Режим доступу: [https://docs.gitlab.com/ci/migration/examples/jenkins-
maven](https://docs.gitlab.com/ci/migration/examples/jenkins-maven) (Дата звернення: 21.03.2025).

48. Docker Inc. Docker-in-Docker: Containerized CI Workflows.
[Електронний ресурс]. – Режим доступу:
[https://www.docker.com/resources/docker-in-docker-containerized-ci-workflows-
dockercon-2023/](https://www.docker.com/resources/docker-in-docker-containerized-ci-workflows-dockercon-2023/) (Дата звернення: 21.03.2025).

49. GitLab Inc. Cosign with GitLab Container Registry. [Електронний
ресурс]. – Режим доступу:
https://docs.gitlab.com/user/packages/container_registry/cosign_tutorial/ (Дата
звернення: 21.03.2025).

50. GitLab Inc. CI/CD variables in AutoDevOps. [Електронний ресурс]. –
Режим доступу: https://docs.gitlab.com/topics/autodevops/cicd_variables/ (Дата
звернення: 21.03.2025).

51. GitLab Inc. CI/CD variables: Step 4: Use CI/CD variables as part of jobs.
[Електронний ресурс]. – Режим доступу: <https://docs.gitlab.com/ci/> (Дата
звернення: 21.03.2025).

52. Kubernetes. Working with Kubernetes Objects. [Електронний ресурс]. –

Режим доступа: <https://kubernetes.io/docs/concepts/overview/working-with-objects/>
(Дата звернення: 21.03.2025).

53. Kubernetes. DNS for Services and Pods. [Електронний ресурс]. – Режим доступа: <https://kubernetes.io/docs/concepts/services-networking/dns-pod-service/>
(Дата звернення: 21.03.2025).

54. Spring. Spring Cloud Netflix Configuration Properties. [Електронний ресурс]. – Режим доступа: <https://docs.spring.io/spring-cloud-netflix/reference/configprops.html> (Дата звернення: 21.03.2025).

55. Spring. Spring Cloud Netflix. [Електронний ресурс]. – Режим доступа: <https://docs.spring.io/spring-cloud-netflix/reference/spring-cloud-netflix.html> (Дата звернення: 21.03.2025).

56. Spring. CORS Support in Spring Web MVC. [Електронний ресурс]. – Режим доступа: <https://docs.spring.io/spring-framework/reference/web/webmvc-cors.html> (Дата звернення: 21.03.2025).

57. Spring. Spring Cloud Netflix Ribbon. [Електронний ресурс]. – Режим доступа: https://cloud.spring.io/spring-cloud-netflix/multi/multi_spring-cloud-ribbon.html (Дата звернення: 21.03.2025).

58. Oracle. The Java Persistence API. [Електронний ресурс]. – Режим доступа: <https://docs.oracle.com/javaee/6/tutorial/doc/bnbpz.html> (Дата звернення: 21.03.2025).

59. GitLab Inc. GitOps. [Електронний ресурс]. – Режим доступа: <https://about.gitlab.com/topics/gitops/> (Дата звернення: 21.03.2025).

60. Kubernetes SIGs. Minikube Documentation. [Електронний ресурс]. – Режим доступа: <https://minikube.sigs.k8s.io/docs/> (Дата звернення: 21.03.2025).

61. Monokle. Kubernetes Manifest Files Explained. [Електронний ресурс]. – Режим доступа: <https://monokle.io/learn/kubernetes-manifest-files-explained> (Дата звернення: 21.03.2025).

62. Kubernetes SIGs. Minikube Image Commands. [Електронний ресурс]. – Режим доступа: <https://minikube.sigs.k8s.io/docs/commands/image/> (Дата звернення: 21.03.2025).

63. Argo CD. Argo CD Documentation. [Електронний ресурс]. – Режим доступу: <https://argo-cd.readthedocs.io/en/stable/> (Дата звернення: 21.03.2025).

64. Argo CD. Getting Started. [Електронний ресурс]. – Режим доступу: https://argo-cd.readthedocs.io/en/stable/getting_started/ (Дата звернення: 21.03.2025).

65. iambryancs. GitHub Gist. [Електронний ресурс]. – Режим доступу: <https://gist.github.com/iambryancs/fa4051f2cd828f58c04fc968b546a28e> (Дата звернення: 21.03.2025).

66. Data Hints. Ідемпотентність або стійкість до постійного навантаження в однакових відповідях сервера. [Електронний ресурс]. – Режим доступу: <https://data-hints.com/idempotentnist-abo-stiykist-do-postiynoho-navantazhennia-v-odnakovykh-vidpovidiakh-servera/> (Дата звернення: 21.03.2025).

67. Arlingappa, I. Kubernetes Reconciliation Loop. [Електронний ресурс]. – Режим доступу: <https://medium.com/@inchararlingappa/kubernetes-reconciliation-loop-74d3f38e382f> (Дата звернення: 21.03.2025).

68. GitLab Inc. Personal Access Tokens. [Електронний ресурс]. – Режим доступу: https://docs.gitlab.com/user/profile/personal_access_tokens/ (Дата звернення: 21.03.2025).

69. Kubernetes. kubectl create secret. [Електронний ресурс]. – Режим доступу: https://kubernetes.io/docs/reference/kubectl/generated/kubectl_create/kubectl_create_secret/ (Дата звернення: 21.03.2025).

70. Kubernetes. kubectl apply. [Електронний ресурс]. – Режим доступу: https://kubernetes.io/docs/reference/kubectl/generated/kubectl_apply/ (Дата звернення: 21.03.2025).

71. Kubernetes SIGs. Minikube Service Command. [Електронний ресурс]. – Режим доступу: <https://minikube.sigs.k8s.io/docs/commands/service/> (Дата звернення: 21.03.2025).

72. Jayway. JsonPath Documentation. [Електронний ресурс]. – Режим доступу: <https://javadoc.io/doc/com.jayway.jsonpath/json->

path/2.2.0/com/jayway/jsonpath/JsonPath.html (Дата звернення: 21.03.2025).

73. Kubernetes. kubectl patch. [Електронний ресурс]. – Режим доступу: https://kubernetes.io/docs/reference/kubectl/generated/kubectl_patch/ (Дата звернення: 21.03.2025).

74. Spring PetClinic Microservices. Prometheus Dockerfile. [Електронний ресурс]. – Режим доступу: <https://github.com/spring-petclinic/spring-petclinic-microservices/blob/main/docker/prometheus/Dockerfile> (Дата звернення: 21.03.2025).

75. Spring PetClinic Microservices. Prometheus Configuration. [Електронний ресурс]. – Режим доступу: <https://github.com/spring-petclinic/spring-petclinic-microservices/blob/main/docker/prometheus/prometheus.yml> (Дата звернення: 21.03.2025).

76. VictoriaMetrics. Kubernetes service discovery configurations. [Електронний ресурс]. – Режим доступу: https://docs.victoriametrics.com/victoriametrics/sd_configs/#kubernetes_sd_configs (Дата звернення: 21.03.2025).

77. Spring PetClinic Microservices. Grafana Dockerfile. [Електронний ресурс]. – Режим доступу: <https://github.com/spring-petclinic/spring-petclinic-microservices/blob/main/docker/grafana/Dockerfile> (Дата звернення: 21.03.2025).

78. Grafana. Provisioning Grafana. [Електронний ресурс]. – Режим доступу: <https://grafana.com/docs/grafana/latest/administration/provisioning/> (Дата звернення: 21.03.2025).

ДОДАТКИ

Додаток А. Файл .gitlab-ci.yml

stages:

- build
- test
- package
- docker
- monitoring

variables:

```

MAVEN_OPTS: "-Dmaven.repo.local=.m2/repository -
Dorg.slf4j.simpleLogger.log.org.apache.maven.cli.transfer.Slf4jMavenTransfer
Listener=WARN -Dorg.slf4j.simpleLogger.showDateTime=true -
Djava.awt.headless=true"
MAVEN_CLI_OPTS: "--batch-mode --errors --fail-at-end --show-version"
DOCKER_HOST: tcp://docker:2375
DOCKER_DRIVER: overlay2
DOCKER_TLS_CERTDIR: ""
IMAGE_TAG: ${CI_COMMIT_SHORT_SHA}

# Кешування залежностей Maven для прискорення збірки
cache:
  key: ${CI_COMMIT_REF_SLUG}
  paths:
    - .m2/repository
    - */target
```

Етап збірки - компіляція коду

build:

stage: build

image: maven:3.8-openjdk-17-slim

script:

- mvn \$MAVEN_CLI_OPTS compile

rules:

- if: \$CI_COMMIT_BRANCH

Етап тестування - паралельний запуск модульних тестів

test:parallel:

stage: test

image: maven:3.8-openjdk-17-slim

parallel: 7

script:

- |

case \$CI_NODE_INDEX in

0) mvn \$MAVEN_CLI_OPTS test -pl spring-petclinic-admin-server ;;

1) mvn \$MAVEN_CLI_OPTS test -pl spring-petclinic-api-gateway ;;

2) mvn \$MAVEN_CLI_OPTS test -pl spring-petclinic-config-server ;;

3) mvn \$MAVEN_CLI_OPTS test -pl spring-petclinic-customers-service

;;

4) mvn \$MAVEN_CLI_OPTS test -pl spring-petclinic-discovery-server ;;

5) mvn \$MAVEN_CLI_OPTS test -pl spring-petclinic-vets-service ;;

6) mvn \$MAVEN_CLI_OPTS test -pl spring-petclinic-visits-service ;;

esac

artifacts:

reports:

junit:

- "*/target/surefire-reports/TEST-*.xml"

needs:

- build

rules:

- if: \$CI_COMMIT_BRANCH

Етап пакування - створення JAR-файлів

package:

stage: package

image: maven:3.8-openjdk-17-slim

script:

- mvn \$MAVEN_CLI_OPTS package -DskipTests

artifacts:

paths:

- spring-petclinic-admin-server/target/*.jar
- spring-petclinic-api-gateway/target/*.jar
- spring-petclinic-config-server/target/*.jar
- spring-petclinic-customers-service/target/*.jar
- spring-petclinic-discovery-server/target/*.jar
- spring-petclinic-vets-service/target/*.jar
- spring-petclinic-visits-service/target/*.jar

expire_in: 1 week

needs:

- test:parallel

rules:

- if: \$CI_COMMIT_BRANCH

Етап збірки та публікації Docker-образів для основних сервісів

docker-build-push:

stage: docker

image: docker:20.10.16

services:

- docker:20.10.16-dind

script:

- # Логін in Docker Hub

- echo "\${DOCKER_HUB_PASSWORD}" | docker login -u

"\${DOCKER_HUB_USERNAME}" --password-stdin

- # Збірка всіх основних образів

- echo "Building and pushing Docker images with tag \${IMAGE_TAG}"

- # Admin Server

- docker build -t \${DOCKER_HUB_USERNAME}/petclinic-admin-server:\${IMAGE_TAG} -f docker/admin-server/Dockerfile .

- docker tag \${DOCKER_HUB_USERNAME}/petclinic-admin-server:\${IMAGE_TAG} \${DOCKER_HUB_USERNAME}/petclinic-admin-server:latest

- docker push \${DOCKER_HUB_USERNAME}/petclinic-admin-server:\${IMAGE_TAG}

- docker push \${DOCKER_HUB_USERNAME}/petclinic-admin-server:latest

- # API Gateway

- docker build -t \${DOCKER_HUB_USERNAME}/petclinic-api-gateway:\${IMAGE_TAG} -f docker/api-gateway/Dockerfile .

- docker tag \${DOCKER_HUB_USERNAME}/petclinic-api-gateway:\${IMAGE_TAG} \${DOCKER_HUB_USERNAME}/petclinic-api-gateway:latest

- docker push \${DOCKER_HUB_USERNAME}/petclinic-api-gateway:\${IMAGE_TAG}

```
- docker push ${DOCKER_HUB_USERNAME}/petclinic-api-  
gateway:latest
```

Config Server

```
- docker build -t ${DOCKER_HUB_USERNAME}/petclinic-config-  
server:${IMAGE_TAG} -f docker/config-server/Dockerfile .
```

```
- docker tag ${DOCKER_HUB_USERNAME}/petclinic-config-  
server:${IMAGE_TAG} ${DOCKER_HUB_USERNAME}/petclinic-config-  
server:latest
```

```
- docker push ${DOCKER_HUB_USERNAME}/petclinic-config-  
server:${IMAGE_TAG}
```

```
- docker push ${DOCKER_HUB_USERNAME}/petclinic-config-  
server:latest
```

Customers Service

```
- docker build -t ${DOCKER_HUB_USERNAME}/petclinic-customers-  
service:${IMAGE_TAG} -f docker/customers-service/Dockerfile .
```

```
- docker tag ${DOCKER_HUB_USERNAME}/petclinic-customers-  
service:${IMAGE_TAG} ${DOCKER_HUB_USERNAME}/petclinic-customers-  
service:latest
```

```
- docker push ${DOCKER_HUB_USERNAME}/petclinic-customers-  
service:${IMAGE_TAG}
```

```
- docker push ${DOCKER_HUB_USERNAME}/petclinic-customers-  
service:latest
```

Discovery Server

```
- docker build -t ${DOCKER_HUB_USERNAME}/petclinic-discovery-  
server:${IMAGE_TAG} -f docker/discovery-server/Dockerfile .
```

```
- docker tag ${DOCKER_HUB_USERNAME}/petclinic-discovery-
server:${IMAGE_TAG} ${DOCKER_HUB_USERNAME}/petclinic-discovery-
server:latest
```

```
- docker push ${DOCKER_HUB_USERNAME}/petclinic-discovery-
server:${IMAGE_TAG}
```

```
- docker push ${DOCKER_HUB_USERNAME}/petclinic-discovery-
server:latest
```

Vets Service

```
- docker build -t ${DOCKER_HUB_USERNAME}/petclinic-vets-
service:${IMAGE_TAG} -f docker/vets-service/Dockerfile .
```

```
- docker tag ${DOCKER_HUB_USERNAME}/petclinic-vets-
service:${IMAGE_TAG} ${DOCKER_HUB_USERNAME}/petclinic-vets-
service:latest
```

```
- docker push ${DOCKER_HUB_USERNAME}/petclinic-vets-
service:${IMAGE_TAG}
```

```
- docker push ${DOCKER_HUB_USERNAME}/petclinic-vets-
service:latest
```

Visits Service

```
- docker build -t ${DOCKER_HUB_USERNAME}/petclinic-visits-
service:${IMAGE_TAG} -f docker/visits-service/Dockerfile .
```

```
- docker tag ${DOCKER_HUB_USERNAME}/petclinic-visits-
service:${IMAGE_TAG} ${DOCKER_HUB_USERNAME}/petclinic-visits-
service:latest
```

```
- docker push ${DOCKER_HUB_USERNAME}/petclinic-visits-
service:${IMAGE_TAG}
```

```
- docker push ${DOCKER_HUB_USERNAME}/petclinic-visits-
service:latest
```

```

# Запис інформації про деплой
- echo "Deployed images with tag ${IMAGE_TAG} at $(date)" >
deployment.txt
artifacts:
  paths:
    - deployment.txt
  expire_in: 1 week
needs:
  - package
rules:
  - if: $CI_COMMIT_BRANCH

# Новий етап для збірки та публікації контейнерів для моніторингу
monitoring-build-push:
  stage: monitoring
  image: docker:20.10.16
  services:
    - docker:20.10.16-dind
  script:
    # Логін in Docker Hub
    - echo "${DOCKER_HUB_PASSWORD}" | docker login -u
"${DOCKER_HUB_USERNAME}" --password-stdin

    # Prometheus
    - echo "Building and pushing Prometheus image with tag
${IMAGE_TAG}"
    - docker build -t ${DOCKER_HUB_USERNAME}/petclinic-
prometheus:${IMAGE_TAG} -f docker/prometheus/Dockerfile .

```

```
- docker tag ${DOCKER_HUB_USERNAME}/petclinic-
prometheus:${IMAGE_TAG} ${DOCKER_HUB_USERNAME}/petclinic-
prometheus:latest
```

```
- docker push ${DOCKER_HUB_USERNAME}/petclinic-
prometheus:${IMAGE_TAG}
```

```
- docker push ${DOCKER_HUB_USERNAME}/petclinic-
prometheus:latest
```

```
# Grafana
```

```
- echo "Building and pushing Grafana image with tag ${IMAGE_TAG}"
```

```
- docker build -t ${DOCKER_HUB_USERNAME}/petclinic-
grafana:${IMAGE_TAG} -f docker/grafana/Dockerfile .
```

```
- docker tag ${DOCKER_HUB_USERNAME}/petclinic-
grafana:${IMAGE_TAG} ${DOCKER_HUB_USERNAME}/petclinic-
grafana:latest
```

```
- docker push ${DOCKER_HUB_USERNAME}/petclinic-
grafana:${IMAGE_TAG}
```

```
- docker push ${DOCKER_HUB_USERNAME}/petclinic-grafana:latest
```

```
# Запис інформації про деплой моніторингу
```

```
- echo "Deployed monitoring images with tag ${IMAGE_TAG} at $(date)"
```

```
> monitoring_deployment.txt
```

```
artifacts:
```

```
paths:
```

```
- monitoring_deployment.txt
```

```
expire_in: 1 week
```

```
needs:
```

```
- docker-build-push
```

```
rules:
```

```
- if: $CI_COMMIT_BRANCH
```

Додаток Б. Маніфест .config-server.yaml

```

apiVersion: apps/v1
kind: Deployment
metadata:
  name: config-server
  labels:
    app: config-server
spec:
  replicas: 1
  selector:
    matchLabels:
      app: config-server
  template:
    metadata:
      labels:
        app: config-server
    spec:
      containers:
        - name: config-server
          image: xchubuk/petclinic-config-server:latest
          imagePullPolicy: Always
          ports:
            - containerPort: 8888
          env:
            - name: SPRING_PROFILES_ACTIVE
              value: "kubernetes"
---
apiVersion: v1

```

```

kind: Service
metadata:
  name: config-server
spec:
  selector:
    app: config-server
  ports:
    - port: 8888
      targetPort: 8888

```

Додаток В. Маніфест customers-service.yaml

```

apiVersion: apps/v1
kind: Deployment
metadata:
  name: customers-service
  labels:
    app: customers-service
spec:
  replicas: 1
  selector:
    matchLabels:
      app: customers-service
  template:
    metadata:
      labels:
        app: customers-service
    spec:
      containers:
        - name: customers-service
          image: xchubuk/petclinic-customers-service:latest

```

```

    imagePullPolicy: Always
  ports:
    - containerPort: 8081
  env:
    - name: SPRING_PROFILES_ACTIVE
      value: "kubernetes"
    - name: SPRING_CONFIG_IMPORT
      value: "optional:configserver:http://config-server:8888"
---
apiVersion: v1
kind: Service
metadata:
  name: customers-service
spec:
  selector:
    app: customers-service
  ports:
    - port: 8081
      targetPort: 8081

```

Додаток Г. Маніфест discovery-server.yaml

```

apiVersion: apps/v1
kind: Deployment
metadata:
  name: discovery-server
  labels:
    app: discovery-server
spec:
  replicas: 1
  selector:

```

```

    matchLabels:
      app: discovery-server
  template:
    metadata:
      labels:
        app: discovery-server
    spec:
      containers:
        - name: discovery-server
          image: xchubuk/petclinic-discovery-server:latest
          imagePullPolicy: Always
          ports:
            - containerPort: 8761
          env:
            - name: SPRING_PROFILES_ACTIVE
              value: "kubernetes"
---
apiVersion: v1
kind: Service
metadata:
  name: discovery-server
spec:
  selector:
    app: discovery-server
  ports:
    - port: 8761
      targetPort: 8761

```

Додаток Г. Маніфест discovery-server.yaml

```
apiVersion: apps/v1
```

```
kind: Deployment
metadata:
  name: discovery-server
  labels:
    app: discovery-server
spec:
  replicas: 1
  selector:
    matchLabels:
      app: discovery-server
  template:
    metadata:
      labels:
        app: discovery-server
    spec:
      containers:
        - name: discovery-server
          image: xchubuk/petclinic-discovery-server:latest
          imagePullPolicy: Always
          ports:
            - containerPort: 8761
          env:
            - name: SPRING_PROFILES_ACTIVE
              value: "kubernetes"
---
apiVersion: v1
kind: Service
metadata:
  name: discovery-server
spec:
```

```

selector:
  app: discovery-server
ports:
- port: 8761
  targetPort: 8761

```

Додаток Д. Маніфест vets-service.yaml

```

apiVersion: apps/v1
kind: Deployment
metadata:
  name: vets-service
  labels:
    app: vets-service
spec:
  replicas: 1
  selector:
    matchLabels:
      app: vets-service
  template:
    metadata:
      labels:
        app: vets-service
    spec:
      containers:
      - name: vets-service
        image: xchubuk/petclinic-vets-service:latest
        imagePullPolicy: Always
        ports:
        - containerPort: 8083
        env:

```

```

- name: SPRING_PROFILES_ACTIVE
  value: "kubernetes"
- name: SPRING_CONFIG_IMPORT
  value: "optional:configserver:http://config-server:8888"

```

```
---
```

```

apiVersion: v1
kind: Service
metadata:
  name: vets-service
spec:
  selector:
    app: vets-service
  ports:
    - port: 8083
      targetPort: 8083

```

Додаток Д. Маніфест visits-service.yaml

```

apiVersion: apps/v1
kind: Deployment
metadata:
  name: visits-service
  labels:
    app: visits-service
spec:
  replicas: 1
  selector:
    matchLabels:
      app: visits-service
  template:
    metadata:

```

```
  labels:
    app: visits-service
spec:
  containers:
  - name: visits-service
    image: xchubuk/petclinic-visits-service:latest
    imagePullPolicy: Always
    ports:
    - containerPort: 8082
    env:
    - name: SPRING_PROFILES_ACTIVE
      value: "kubernetes"
    - name: SPRING_CONFIG_IMPORT
      value: "optional:configserver:http://config-server:8888"
---
apiVersion: v1
kind: Service
metadata:
  name: visits-service
spec:
  selector:
    app: visits-service
  ports:
  - port: 8082
    targetPort: 8082
```