СХІДНОУКРАЇНСЬКИЙ НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ ІМЕНІ ВОЛОДИМИРА ДАЛЯ

Навчально-науковий інститут (факультет) інформаційних технологій та електроніки

Кафедра інформаційних технологій та програмування

Пояснювальна записка

до магістерської дипломної роботи

магістр

(освітньо-кваліфікаційний рівень)

на тему Застосування DevOps практик для підвищення продуктивності та якості розробки хмарних інформаційних систем

Виконав: студент 2 курсу, групи ІСТ-22дм

126 «Інформаційні системи та технології»

(шифр і назва спеціальності)

Чжен А.К.

(прізвище та ініціали)

Керівник Захожай О.І.

(прізвище та ініціали)

Рецензент Меняйленко О.С.

(прізвище та ініціали)

Київ – 2023 року

ІНДИВІДУАЛЬНЕ ЗАВДАННЯ ДО МАГІСТЕРСЬКОЇ ДИПЛОМНОЇ РОБОТИ

СХІДНОУКРАЇНСЬКИЙ НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ

ІМЕНІ ВОЛОДИМИРА ДАЛЯ

Навчально-науковий інститут (факультет) інформаційних технологій та електроніки

Кафедра інформаційних технологій та програмування

Освітньо-кваліфікаційний рівень магістр

Спеціальність 126 «Інформаційні системи та технології

(шифр і назва спеціальності)

ЗАТВЕРДЖУЮ
Завідувач кафедри ПМ,
_____д.т.н., проф. Лифар В.О.
(підпис)
«____»_____2023 р.

ЗАВДАННЯ

на магістерську дипломну роботу студенту

_____Чжен Андрію Климентійовичу_____

(прізвище, ім’я, по батькові)

1. Тема роботи __Застосування DevOps практик для підвищення продуктивності____ та якості розробки хмарних інформаційних систем_____
керівник роботи _____Захожай Олег Ігорович, д.т.н., професор_____

(прізвище, ім’я, по батькові, науковий ступінь, вчене звання)

затверджені наказом вищого навчального закладу від «___»____2023 року №_____

2. Строк подання студентом роботи _____

3. Вихідні дані до роботи ___Матеріали науково-дослідницької практики, науково-методична література; дані інтернет-мережі_____
_____

4. Зміст розрахунково-пояснювальної записки (перелік питань, які потрібно розробити) _____
_____
_____

5. Перелік графічного матеріалу (з точним значенням обов’язків креслень)_____
_____
_____
_____

6. Консультанти розділів проекту (роботи)

Розділ	Прізвище, ініціали та посада консультанта	Підпис, дата	
		завдання видав	завдання прийняв

7. Дата видачі завдання_____20 жовтня 2023р._____

КАЛЕНДАРНИЙ ПЛАН

№ з\п	Назва етапів дипломної роботи	Строк виконання етапів роботи	Примітка
1	Одержання завдання на виконання роботи	20.10.2023	
2	Укладання і погодження з керівником плану і етапів виконання роботи	24.10.2023	
3	Узагальнення даних літературних джерел, укладання розділу «Аналіз предметної галузі»	28.10.2023	
4	Розробка технічного завдання	01.11.2023	
5	Аналіз технічних засобів та існуючих систем	07.11.2023	
6	Реалізація практичної частини завдання	24.11.2023	
7	Оформлення пояснювальної записки	05.12.2023	
8	Підготовка та подання магістерської роботи до захисту	09.12.2023	

Студент _____ ____Чжен А.К._____
(підпис) (прізвище та ініціали)

Керівник роботи _____ ____Захожай О.І.____
(підпис) (прізвище та ініціали)

РЕФЕРАТ

Магістерська дипломна робота: 55 стор., 20 рис., 11 джерел.

Об’єкт дослідження – процес розробки хмарних інформаційних систем.

Мета роботи – детальне вивчення та аналіз застосування практик DevOps для підвищення продуктивності та якості розробки в контексті хмарних інформаційних систем та на основі отриманої інформації визначення етапів впровадження DevOps практик.

Розглянуто процес розробки інформаційниї систем, зроблений аналіз існуючих проблем при розбробці. Висвітлено поняття DevOps та його підвид GitOps. Продемонстровано на практиці як впровадження DevOps практик впливає на якіть і продуктивність розробки.

КЛЮЧОВІ СЛОВА: РОЗРОБКА, РОЗГОРТАННЯ, ТЕСТУВАННЯ, CI/CD, GITOPS, IAC, KUBERNETES, DOCKER.

ABSTRACT

Master's thesis: 55 pages, 20 pictures, 11 sources.

Research Object is the development process of cloud information systems.

The purpose of the work is a detailed study and analysis of the application of DevOps practices to improve productivity and development quality in the context of cloud information systems. Based on the obtained information, the goal is to identify the stages of implementing DevOps practices.

The development process of information systems is examined, and an analysis of existing problems in development is conducted. The concept of DevOps and its subset, GitOps, is highlighted. Practical demonstrations showcase how the implementation of DevOps practices influences the quality and productivity of development.

KEYWORDS: DEVELOPMENT, DEPLOYMENT, TESTING, CI/CD, GITOPS, IAC, KUBERNETES, DOCKER.

ПЕРЕЛІК ПОЗНАЧЕНЬ ТА СКОРОЧЕНЬ

DevOps (Development and Operations) – розробка та операції

CD (Continuous Deployment or Delivery) – безперервне розгортання/доставка

CI (Continuous Integration) – неперервна інтеграція

CT (Continuous Testing) – безперервне тестування

IAC (Infrastructure as a Code) – інфраструктура як код

ОС – операційна система

ІС – інформаційна система

ПЗ – програмне забезпечення

API (Application Programming Interface) – програмний інтерфейс додатку

QA – (Quality Assurance) – забезпечення якості

VCS - (Version Control System) – система контролю версій

ЗМІСТ

ВСТУП

Актуальність досліджень в контексті сучасних технологічних тенденцій.

У сучасному світі спостерігається стрімке збільшення обсягів даних та складності інформаційних систем, особливо в контексті хмарних середовищ. Розширення функціональності та збільшення кількості користувачів вимагають нових підходів до розробки, впровадження та управління системами.

Сучасні користувачі та бізнес-замовники дедалі більше прагнуть до швидкості внесення змін та гнучкості в управлінні проектами. DevOps виявляється ключовим елементом для забезпечення цих вимог, дозволяючи здійснювати швидке внесення змін та автоматизувати весь цикл розробки.

З огляду на постійні кіберзагрози та вимоги до забезпечення безпеки даних, актуальним стає впровадження автоматизованих засобів моніторингу, тестування та реагування на інциденти. DevOps може служити ефективним інструментом для автоматизації процесів забезпечення безпеки в хмарних інформаційних системах.

Умови сучасного ринку вимагають від компаній не лише швидкості внесення змін, але й оптимізації ресурсів. DevOps дозволяє компаніям оптимізувати розробку, зменшувати терміни випуску продукту та забезпечувати стабільність роботи систем.

Розширення екосистеми відкритих джерел та стандартів у галузі хмарних технологій створює можливості для інтеграції різних інструментів та технологій DevOps у хмарному середовищі.

Враховуючи ці фактори, дослідження застосування DevOps практик для підвищення продуктивності та якості розробки хмарних інформаційних систем є надзвичайно актуальним та важливим для подальшого розвитку індустрії програмного забезпечення.

Метою даної роботи є детальне вивчення та аналіз застосування практик DevOps для підвищення продуктивності та якості розробки в контексті хмарних інформаційних систем. Робота спрямована на визначення переваг, викликів та

оптимальних стратегій впровадження DevOps у хмарних середовищах, а також на розробку рекомендацій для підприємств, що прагнуть оптимізувати свої розробницькі процеси та підвищити якість своїх продуктів.

Для досягнення поставленої мети робота вирішує наступні завдання:

- вивчення особливостей хмарних інформаційних систем.
- дослідження основних характеристик та викликів, що виникають при розробці та управлінні хмарними інформаційними системами.
- здійснення докладного аналізу основних принципів та інструментів DevOps, а також їх застосування в контексті розробки хмарних інформаційних систем.
- визначення оптимальної стратегії впровадження DevOps для підвищення ефективності та якості розробки в хмарних середовищах.
- реалізація та апробація практичних аспектів DevOps в хмарних інформаційних системах.
- виявлення можливих проблем та розробка рекомендацій для підприємств, які розглядають впровадження DevOps.

Ці завдання спрямовані на створення збалансованого та всебічного огляду можливостей та викликів, пов'язаних з застосуванням DevOps у хмарному середовищі, та на розробку конкретних рекомендацій для підприємств при розробці та впровадженні програмного забезпечення.

РОЗДІЛ 1. АНАЛІЗ ПРЕДМЕТНОЇ ОБЛАСТІ

1.1 Характеристика хмарних інформаційних систем

Хмарні інформаційні системи – це комплексна інфраструктура, що надає доступ до обчислювальних ресурсів, мережевих служб, а також платформ та додатків через мережу Інтернет. Хмарні обчислення (англ. cloud computing) або скорочено хмара – модель забезпечення повсюдного та зручного доступу на вимогу через мережу до спільного пулу обчислювальних ресурсів, що підлягають налаштуванню (наприклад, до комунікаційних мереж, серверів, засобів збереження даних, прикладних програм та сервісів), і які можуть бути оперативно надані та звільнені з мінімальними управлінськими затратами та зверненнями до провайдера[1]. Цей підхід дозволяє використовувати ресурси, коли це потрібно, забезпечуючи високий рівень гнучкості та ефективності. Перевага технології в тому, що користувач має доступ до власних даних, але не повинен піклуватися про фізичну інфраструктуру (серверне, мережеве обладнання тощо), а в окремих випадках і про операційну систему та програмне забезпечення, з яким він працює. Слово «хмара» – це метафора, що уособлює складну інфраструктуру, що приховує за собою всі технічні деталі.

За моделями розгортання розрізняють наступні категорії хмарних інформаційних систем [2]:

- публічна хмара (public cloud) – це хмарна інфраструктура, яка призначена для використання широким загалом. Публічна хмара може перебувати у власності, керуванні та експлуатації комерційних, академічних або державних організацій. Публічна хмара перебуває в юрисдикції постачальника хмарних послуг. Можливостей управляти і обслуговувати дану хмару у користувачів немає, вся відповідальність покладена на її власника. Абонентом пропонованих сервісів може стати будь-яка компанія або приватна особа. Найпопулярніші публічні комерційні хмари є – Amazon AWS, Google Cloud та Microsoft Azure.

- приватна хмара (private cloud) – IT-інфраструктура, яку контролює і експлуатує тільки один абонент у власних інтересах. Інфраструктура для управління приватною хмарою може розміщуватися або в приміщеннях користувача, або у зовнішнього оператора, або частково у користувача і оператора.

- громадська хмара (community cloud) – це хмарна інфраструктура, яка призначена для використання конкретною спільнотою споживачів із організацій, що мають спільні цілі (наприклад, місію, вимоги щодо безпеки, політику та відповідність різноманітним вимогам). Громадська хмара може перебувати у спільній власності, керуванні та експлуатації однієї чи більше організацій зі спільноти або третьої сторони (чи деякої їх комбінації). Така хмара може фізично знаходитись як в, так і поза юрисдикцією власника.

- гібридна хмара (hybrid cloud) – це хмарна інфраструктура, що складається з двох або більше різних хмарних інфраструктур (приватних, громадських або публічних), які залишаються унікальними сутностями, але з'єднані між собою стандартизованими або приватними технологіями, що уможливлюють переносимість даних та прикладних програм (наприклад, використання ресурсів публічної хмари для балансування навантаження між хмарами).

- персональна хмара (personal cloud) – це приватна колекція цифрового контенту та додаткових сервісів які доступні з будь-якого пристрою і призначена для використання окремою особою (власником) та особами яким надано доступ. Це місце де користувач має можливість зберігати, синхронізувати, транслювати в потік та розповсюджувати приватний контент на сумісні платформи, екрани, з одного місцеположення в інше.

Наступна характеристика хмарних інформаційних систем – це рівень хмарних обчислень. Існують наступні рівні:

- нижчий рівень “Інфраструктура як послуга” (IaaS, infrastructure as a service). Користувачі отримують базові обчислювальні ресурси: процесор, оперативна пам’ять, дисковий простір, мережеві адаптери, які в подальшому використовують для встановлення власних операційних систем і додатків. Споживач не керує базовою інфраструктурою хмари, але має контроль над операційними системами,

системами зберігання, розгорнутими додатками. Можливий обмежений контроль вибору мережевих компонентів, наприклад – хост з мережевими екранами. Приклади IaaS – Amazon Web Services (AWS), Google Compute Engine (GCE), Microsoft Azure.

- наступний рівень “Платформа як послуга” (PaaS, platform as a service). Користувачі мають можливість встановлювати власні додатки на платформі, що надається провайдером послуги. Користувач не керує базовою інфраструктурою хмари, але має контроль над розгорнутими додатками і деякими параметрами конфігурації середовища хостингу. Приклади PaaS – Google App Engine, AWS Elastic Beanstalk, Windows Azure, Heroku.

- вищий рівень хмарних обчислень “Програмне забезпечення як послуга” (SaaS, software as a service). У хмарі зберігаються не тільки дані, але і пов'язані з ними програми, а користувачеві для роботи потрібно тільки веб-браузер. Споживач користується додатками провайдера, який працює в хмарній інфраструктурі. При цьому користувач не керує базовою інфраструктурою хмари - мережами, серверами, операційними системами, системами зберігання, навіть індивідуальними налаштуваннями додатків за винятком деяких налаштувань конфігурації програми. Приклади SaaS – Microsoft Office 365, Google Apps, Dropbox. Також відмітимо, на цьому рівні надаються хмарні DevOps сервіси для побудови процесів CI/CD, наприклад - GitLab, Github, Teamcity. [2]

Хмарні технології мають численні переваги, які призводять до їх широкого використання в різних галузях:

- гнучкість та масштабованість: користувачі можуть легко збільшувати або зменшувати використання ресурсів згідно зі змінними потребами. Це забезпечує гнучкість та можливість швидко реагувати на зміни обсягу роботи.

- ефективне використання ресурсів: хмарні постачальники можуть оптимізувати використання обладнання та забезпечити високий рівень ефективності в порівнянні з традиційними моделями власних серверів.

- доступність та надійність: великі хмарні платформи часто працюють на розподілених системах, що забезпечує високий рівень доступності та надійності послуг.

- самообслуговування та автоматизація: користувачі можуть легко самостійно керувати ресурсами, запускати віртуальні машини, налаштовувати сервіси та автоматизувати багато процесів, що прискорює розгортання та управління інфраструктурою.

- економія витрат: відсутність необхідності власних фізичних серверів та обладнання зменшує капітальні витрати. Крім того, оплата може базуватися на фактичному використанні ресурсів, що знижує витрати в порівнянні з традиційними моделями придбання обладнання.

- глобальний доступ: користувачі можуть отримувати доступ до своїх даних та ресурсів з будь-якого місця, де є Інтернет, що полегшує роботу на віддалених робочих місцях та сприяє спільній роботі.

- безпека: великі хмарні постачальники зазвичай вкладають значні зусилля в забезпечення безпеки даних, включаючи шифрування, автентифікацію та інші заходи.

- швидке впровадження нових функцій та оновлень: хмарні постачальники можуть швидко впроваджувати нові функції та оновлення, забезпечуючи користувачам доступ до останніх технологій без необхідності самостійного оновлення інфраструктури.

1.2 Використання послуг провайдерів хмар для побудови ІС

Найбільшими хмарними провайдерами є Amazon Web Services (AWS), Microsoft Azure і Google Cloud Platform (GCP). Ці компанії домінують на ринку хмарних обчислень і пропонують широкий спектр послуг, включаючи обчислення, зберігання, мережі, безпеку, бази даних, аналітику тощо. AWS є найбільшим постачальником хмарних послуг з часткою ринку близько 33%. Він пропонує понад 200 хмарних сервісів і має міцну репутацію завдяки своїй надійності,

масштабованості та безпеці. AWS використовується мільйонами клієнтів по всьому світу, включаючи стартапи, підприємства, уряди та некомерційні організації. Microsoft Azure є другим за величиною постачальником хмарних послуг з часткою ринку близько 18%. Він пропонує широкий спектр послуг, включаючи віртуальні машини, бази даних, сховища, мережі, штучний інтелект тощо. Azure відома своїми гібридними хмарними можливостями, які дозволяють клієнтам запускати свої програми як у хмарі, так і локально. Google Cloud Platform (GCP) володіє третьою найбільшою часткою ринку серед постачальників хмарних послуг – близько 10% [3]. Він пропонує широкий спектр послуг, включаючи обчислення, зберігання, мережу, аналіз даних, машинне навчання тощо. GCP відома своїм досвідом у сфері великих даних та аналітики і використовується багатьма клієнтами в науковому, дослідницькому та фінансовому секторах. Інші відомі хмарні провайдери включають IBM Cloud, Oracle Cloud, Alibaba Cloud і DigitalOcean. Ці компанії пропонують різноманітні хмарні сервіси та орієнтуються на різні ринки та сегменти клієнтів.

Вибір правильного постачальника хмарних послуг може бути складним рішенням, яке залежить від багатьох факторів, таких як тип програми, необхідний рівень безпеки та відповідності, бюджет та технічні знання команди. Важливо ретельно оцінити функції, ціни та продуктивність кожного постачальника та вибрати такий що задовольняє потреби бізнесу.

Хмарні обчислення пропонують організаціям кілька переваг, зниження витрат, масштабованість і гнучкість. Використовуючи хмарні сервіси, підприємства можуть скоротити капітальні витрати, усунувши необхідність мануального обслуговування обладнання та ПЗ. Хмарні технології дозволяють організаціям швидко масштабувати свої ресурси вгору або вниз, забезпечуючи швидкість, необхідну для реагування на мінливі ринкові умови.

Незважаючи на переваги хмарних обчислень, є також кілька проблем, з якими стикаються організації. Однією з основних проблем є складність міграції в хмару. Підприємства повинні ретельно планувати свою міграційну стратегію, щоб забезпечити плавний перехід, який може зайняти багато часу та ресурсів. Крім

того, хмарні обчислення створюють проблеми щодо конфіденційності, безпеки та відповідності даних.

Безпека є серйозною проблемою в хмарних обчисленнях, оскільки дані зберігаються та доступні віддалено. Хмарні провайдери використовують різні заходи безпеки для захисту своєї інфраструктури, такі як шифрування, брандмауери та багатофакторна аутентифікація. Однак організація несе відповідальність за забезпечення безпеки своїх даних у хмарі. Найкращі методи хмарної безпеки включають використання надійних паролів, обмеження доступу до конфіденційних даних, а також регулярний моніторинг і аудит хмарних ресурсів.

Відповідність вимогам є ще однією критичною проблемою для компаній, які використовують хмарні обчислення. Організації повинні забезпечити дотримання різних норм, таких як GDPR, HIPAA та PCI-DSS, під час використання хмарних сервісів [4]. Постачальники хмарних послуг пропонують сертифікати відповідності вимогам, наприклад SOC 2, ISO 27001 і FedRAMP, які демонструють, що їхня інфраструктура відповідає певним вимогам безпеки та відповідності. Підприємства також повинні розробити власну політику та процедури відповідності вимогам, щоб забезпечити виконання своїх регуляторних зобов'язань.

Щоб максимізувати переваги хмарних обчислень при мінімізації їх проблем, організації повинні дотримуватися найкращих практик. До них відносяться розробка хмарної стратегії, вибір правильного постачальника хмарних послуг, ретельне планування міграції, впровадження надійних заходів безпеки, а також регулярний моніторинг та оптимізація хмарних ресурсів.

1.3 Аналіз існуючих проблем в розробці хмарних ІС

Традиційні методології розробки, такі як водоспад (рисунок 1.1), в умовах сучасного динамічного середовища в більшості випадків є не достатньо ефективними.

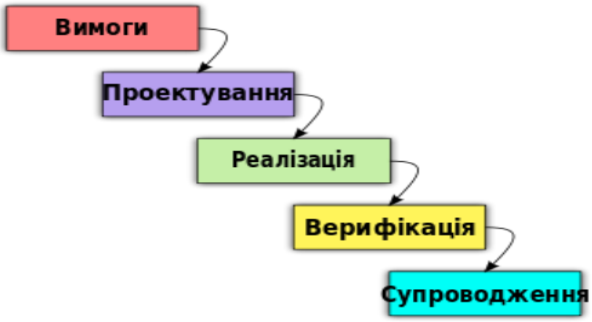

Рисунок 1.1 – Структурна схема методології “Водоспад” (Waterfall)

Найпоширенішими проблемами процесу розробки на основі традиційних методів є [5]:

- жорстка послідовність етапів – традиційні методології передбачають жорстку послідовність етапів розробки, що ускладнює внесення змін під час процесу розробки.

- обмежена здатність адаптації до змін – зміни вимог або функціональності в середині проекту можуть бути складними та витратними.

- довгі цикли розробки – довгі цикли розробки можуть вести до того, що продукт може застаріти, або його власник може втратити інтерес до нього перед завершенням.

- низька залученість замовників – традиційні методології не можуть забезпечити достатнього рівня взаємодії з клієнтом протягом процесу розробки, що може вести до невірного розуміння вимог.

- низька гнучкість в управлінні ризиками – відсутність регулярного оцінювання ризиків: традиційні методології можуть не забезпечити достатній механізм для постійного оцінювання та управління ризиками.

- відсутність реального продукту на ранніх етапах – клієнти можуть не отримати чіткого уявлення про продукт до завершення його розробки.

- високі витрати на зміни – виявлення та виправлення помилок на пізніх етапах може вимагати великих витрат.

- брак гнучкості в розподілі завдань – відсутність можливості гнучко розподіляти завдання: один етап має бути завершений повністю перед переходом

до наступного, що ускладнює паралельну роботу різних команд (дизайнери, розробники, тестувальники).

- недостатнє фокусування на користувачів – відсутність регулярного отримання зворотного зв'язку від користувачів: традиційні методології можуть недостатньо акцентувати увагу на потребах та зворотному зв'язку від кінцевих користувачів.

Ці недоліки традиційних методологій розробки можуть призвести до затримок у виконанні проекту, витрат та несприятливо впливати на його успішність, особливо в умовах швидкозмінюваного та невизначеного оточення розробки програмного забезпечення.

Також потрібно відмітити про проблеми інтеграції та впровадження нових функціональностей. Інтеграція та впровадження нових функціональностей в програмне забезпечення може стати джерелом ряду проблем, які можуть впливати на ефективність та якість розробки [5]. Ось деякі з найпоширеніших проблем у цьому контексті:

- конфлікти з існуючою функціональністю – нові функціональності можуть конфліктувати або переплутатися з існуючими, що може викликати непередбачувану поведінку.

- залежності між функціями – впровадження нових функціональностей може виявити неочікувані залежності між різними частинами системи, що може ускладнити процес інтеграції.

- проблеми зі сумісністю – нові функціональності можуть бути несумісні з існуючими чи іншими новими компонентами системи, що призводить до проблем у роботі програми.

- великі обсяги коду та даних – додавання нових функціональностей може призвести до збільшення обсягів коду та даних, що може ускладнити обслуговування та зробити систему менш ефективною.

- проблеми з безпекою – провадження нових функціональностей може викликати нові точки вразливості, які можуть бути використані для атак, якщо не враховані відповідні заходи безпеки.

- недостатнє тестове покриття – нові функціональності можуть бути погано протестовані, що може призвести до виявлення помилок та неполадок після впровадження.

- незадовільна продуктивність – нові функціональності можуть призвести до збільшення навантаження на систему та негативно вплинути на її продуктивність.

- висока вартість інтеграцій – складність інтеграції нових функціональностей може призвести до великих витрат на час та ресурси.

Для подолання наведених вище проблем потрібні зміни в процесах розробки, нові підходи та нові інструменти.

1.4 Гнучка модель розробки (Agile)

Розглянемо іншу методолгію розробки – гнучка (Agile, рисунок 1.2), яка може подолати більшість вищезгаданих проблем. Цей підхід до розробки програмного забезпечення та управління проектами став важливою частиною сучасного світу програмування. На відміну від традиційних, жорстких і запланованих підходів до розроблення, Agile являє собою гнучку й ітеративну методологію, яка дає змогу командам швидко адаптуватися до вимог, що змінюються, і постійно покращувати свій продукт.

Рисунок 1.2 – Графічне відображення гнучкої моделі розробки ПЗ

Цикл розробки за Agile являє собою серію повторюваних ітерацій, які складають основу процесу розробки ПЗ. У кожній ітерації команда зосереджується на розробці певних частин продукту і доставці працюючого функціонального прототипу або інкременту продукту. Протягом цього часу команда планує, розробляє, тестує і демонструє результати роботи. Завершивши одну ітерацію, команда приступає до наступної, повторюючи процес до досягнення бажаного рівня функціональності продукту. Такий підхід дає змогу команді швидко адаптуватися до змін, покращувати продукт на основі зворотного зв’язку та максимізувати цінність кожної ітерації для замовника [6].

Agile поділяється на декілька окремих гнучких підходів:

1. Scrum – структурований підхід. У Scrum робиться акцент на планомірному контролі процесу розробки. Головна особливість скраму – це розбивка процесу розробки на ітерації з чіткими відрізками часу, зазвичай 2-6 тижнів. На початку спринта проводится “планування спринту” – нарада, де обговорюються головні завдання, які будуть виконуватися протягом спринта. В кінці спринта проводиться “демо” – демонстрація результатів роботи команд за цей спринт. Перед самим першим спринтом замовник, або його представник формує список вимог до майбутнього продукту котрий розроблятиметься. Такі вимоги називають User Story, а самого замовника Product Owner. Для кожного завдання вказується пріоритет і оцінка складності, спочатку будуть реалізовуватися задачі з більш високим пріоритетом. Увесь список називається Product backlog або “резерв продукту”.

2. Kanban (з японского перекладається як “картка”). Даний підтип розробки відрізняється своєю візуалізацією життєвого циклу. Команда орієнтується на виконання завдань, які задаються індивідуально: завдання переходить через всі етапи – розробка коду, перегляд коду, тестування, введення в експлуатацію (етапи можуть бути змінені в індивідуальному порядку). Даний наглядний підхід дозволяє зрозуміти, де саме виникла, або може виникнути проблема, а також дозволяє просто побачити організацію всього проекту [6].

У сучасному світі програмування Agile став невід'ємною частиною успішних проєктів. Гнучкий підхід, орієнтований на людей та їхні потреби, дає змогу створювати якісні продукти та ефективно керувати проектами.

РОЗДІЛ 2. DEVOPS ПІДХІД ДО РОЗРОБКИ ПРОГРАМНОГО ЗАБЕСПЕЧЕННЯ НА ОСНОВІ ХМАРНИХ ТЕХНОЛОГІЙ

2.1 Поняття та принципи DevOps

Поняття DevOps походить від розробка та операції (*development* and *operations*) і характерезує низку практик, призначених для прискорення процесів взаємодії розробників із фахівцями інформаційно-технологічного обслуговування або ті хто використовує програмний продукт, та зближення їхніх робочих процесів одне з одним. Ґрунтується на думці про тісну взаємозалежність між розробкою та використанням програмного забезпечення і має на меті допомогти організаціям швидше створювати та оновлювати програмні продукти та послуги. Завдання DevOps полягає в узгодженні розробки й постачання програмного забезпечення із його використанням. Це завдання вирішується за допомогою автоматичних засобів [7].

При становленні культури DevOps були сформовані 5 основних принципів, які впливають на процес розробки, запровадження та підтримки продукту. Фахівці коротко називають ці принципи CALMS, які розшифровуються як [7]:

- Culture (Культура) – створення спільного інформаційного простору для комплексної та ефективної роботи команд і фахівців з різною спеціалізацією. Принцип також означає відкриту комунікацію для якісної розробки, тестування та впровадження продукту.

- Automatization (Автоматизація) – автоматизація всіляких завдань для спрощення робочого процесу та підвищення якості продукту. При цьому впроваджується технологія безперервного розгортання ПЗ.

- Lean (Ощадливість) – можливість постійного аналізу процесів і виключення тих процесів, які уповільнюють розробку та знижують ефективність. Це допомагає швидше знаходити помилки, виправляти їх і тестувати нові можливості й інструменти.

- Measurement (Вимірювання) або Monitoring (моніторинг) – принцип має на увазі, що проводиться оцінка частоти появ помилок, тривалості використання продукту споживачами та інших критеріїв продуктивності для поліпшення продукту. Що дозволяє оперативно виявити проблеми для їх виправлення або вдосконалити продуктивность.

- Shared Responsibility – розділена відповідальність за помилки й успіхи розробки та впровадження продукту. Робота девелоперів та інших фахівців тісно пов'язана. Чим краще вони взаємодіють, тим краще проходять цикли проекту.

DevOps визначається культурою співпраці, автоматизацією процесів та наголосом на інтеграції та безпеці. Застосування цих понять і принципів сприяє створенню ефективних та надійних розробницько-експлуатаційних практик, що в свою чергу дозволяє компаніям швидше реагувати на зміни та забезпечувати високий рівень якості програмного забезпечення.

2.2 DevOps у розробці програмного забезпечення

Розробка сучасного програмного забезпечення – складний і багатофакторний процес, який передбачає реалізацію ряду етапів, які проходить програмний продукт від появи ідеї до реалізації цієї ідеї в коді, імплементації в бізнес та подальшої підтримки. Життєвий цикл розробки ПЗ є застосуванням інженерних практик, спрямованих на підтримку працездатності, якості та надійності ПЗ. Серед них вагоме місце займає DevOps (Development & Operations), а також відповідні інструменти автоматизації для підвищення ефективності цих процесів за рахунок безперервної інтеграції та активної взаємодії профільних фахівців [8]. З точки зору управління, DevOps позиціонується як agile-підхід для усунення організаційних та тимчасових бар'єрів між командами розробників та

інших учасників життєвого циклу ПЗ, щоб учасники могли швидше і надійніше збирати, тестувати і випускати нові релізи програмних продуктів. Основними перевагами DevOps над іншими методологіями є повна автоматизація, вищий рівень якості ПЗ, менша кількість часу на повний цикл розробки ПЗ. Якщо механізм доставки змін даватиме регулярні збої, то час необхідний для всього циклу розробки збільшиться. Також важливо розуміти, що під поняттям DevOps криється практика взаємодії двох команд розробників та адміністраторів. DevOps підхід реалізується в декілька етапів, які зображені на рисунку 2.1, за допомогою певних інструментів [7].

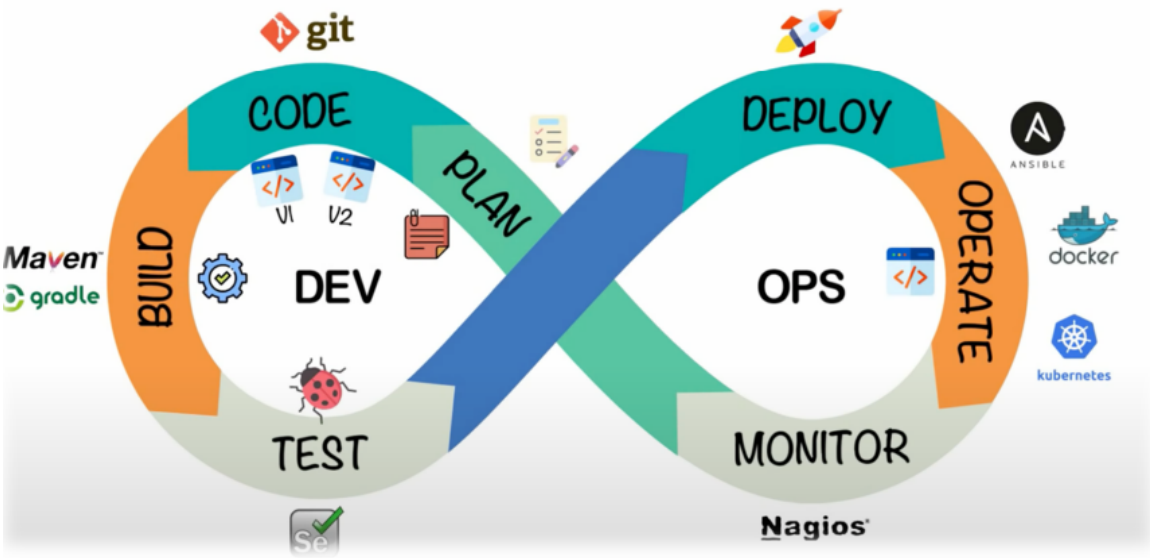

Рисунок 2.1 – Етапи DevOps підходу

Розглянемо кожен з цих етапів:

1. Етап планування: на цьому етапі здійснюється планування, визначення того, що повинно бути реалізовано для клієнта.

2. Етап розробки: здійснюється розробка ПЗ. Вихідний код ПЗ зберігається в системі керування версіями, наприклад Git, яка дозволяє записувати зміни в файлі або в наборі файлів (директорії) та в будь-який момент часу переключатися між різними записаними версіями.

3. Етап збірки: код пакується у виконувані файли для того, щоб він був готовий для розгортання та виконання. Також на цьому етапі можливо юніт тестування, ці тести зазвичай створюють розробники.

4. Етап тестування: на цьому етапі здійснюються різні типи тестування, в тому числі end-to-end тестування, яке дозволяє протестувати систему в цілому. Найбільш популярними інструментами - є Selenium та Cypress, вони дозволяють автоматизувати тестування використовуючи браузер.

5. Етап інтеграції: ядро життєвого циклу DevOps. На цьому етапі відбувається заливка виконуваних файлів (артефакти) на віддалені сервери. Основним відкритим інструментом для цього етапу є Jenkins.

6. Етап доставки та розгортання: як тільки ПЗ протестовано, воно може бути доставлено на продуктивне середовище та розгорнуто на ньому. Команда адміністраторів (devops інженери) здійснює розгортання ПЗ, використовуючи такі системи автоматизації, як Docker (система керування контейнерами), Kubernetes.

7. Етап моніторингу: після розгортання ПЗ його стан постійно моніториться з метою виявлявлення аномалій в роботі. На цьому етапі використовуються такі інструменти автоматизації моніторингу як Prometheus, Grafana.

2.3 Основні практики DevOps: безперервна інтеграція, розгортання та тестування

Основною практикою DevOps – є CI/CD (Continuous Integration/Continuous Deployment) – безпервна інтеграція та доставка, також можна виділити CT (Continuous Testing) – безперервне тестування. Разом їх використовують у CI/CD pipeline – конвеєр. Розглянемо кожну з цих практик.

2.3.1 Безперервна інтеграція коду

Безперервна інтеграція (Continuous Integration, CI) – це практика розробки програмного забезпечення DevOps, при якій розробники регулярно об'єднують зміни програмного коду в центральному репозиторії, після чого автоматично виконується збірка, тестування і розгортання [8]. Поняття безперервної інтеграції

найчастіше застосовується до стадії складання чи інтеграції процесу розробки ПЗ і включає в себе як компонент автоматизації (сервіс безперервної інтеграції або збірки), так і компонент культури розробки (заохочення до частішого внесення змін вихідного коду). Головне завдання безперервної інтеграції – швидше знаходити і виправляти помилки, та скорочувати тимчасові витрати на перевірку і випуск нових оновлень ПЗ. Основна ідея такого підходу полягає в тому, щоб звести до мінімуму вартість інтеграції, зробивши це на ранньому етапі. Розробники можуть виявити конфлікти між новим та існуючим кодом на ранній стадії, коли їх ще відносно легко усунути. Після вирішення конфлікту робота може бути продовжена з упевненістю, що новий код відповідає вимогам існуючої кодової бази. Інтеграція коду часто сама по собі не дає ніяких гарантій щодо якості або функціональності нового коду. У багатьох організаціях інтеграція коштує дорого, адже для забезпечення того, щоб код відповідав стандартам, не вносив помилок і не порушував існуючу функціональність, використовуються ручні процеси. Часта інтеграція може створювати перешкоди, коли рівень автоматизації не відповідає заходам забезпечення якості. Щоб усунути ці перешкоди в процесі інтеграції, на практиці CI спирається на надійні набори тестів і автоматизовану систему для їх виконання. Коли розробник об'єднує свій код з основним сховищем (репозитарій), автоматизовані процеси запускають збірку нового коду. Після цього для нової збірки запускаються тестові набори (юніт тести), які перевіряють, чи не виникли якісь проблеми. Якщо на етапі складання або тестування відбувається збій, команда отримує попередження і може виправити помилки ще на ранньому етапі. Кінцева мета безперервної інтеграції – зробити інтеграцію простим, повторювальним процесом, який є частиною повсякденного робочого процесу розробки, знизити витрати на інтеграцію і своєчасно реагувати на дефекти коду. Робота над надійністю і швидкістю автоматизованої системи, а також розвиток командної культури, яка заохочує часті ітерації і швидке реагування на проблеми, мають основоположне значення для успіху стратегії CI. При безперервній інтеграції розробники часто вносять зміни в спільно використовуваний репозиторій, використовуючи систему контролю версій, наприклад Git. Перед

внесенням кожної зміни у репозиторій, розробники можуть запускати локальні модульні тести програмного коду в якості додаткового рівня перевірки перед інтеграцією. Сервіс безперервної інтеграції (рис. 2.2) автоматично виконує складання (build) та запуск модульних тестів (unit testing) для змін коду, що дозволяє моментально виявляти помилки.

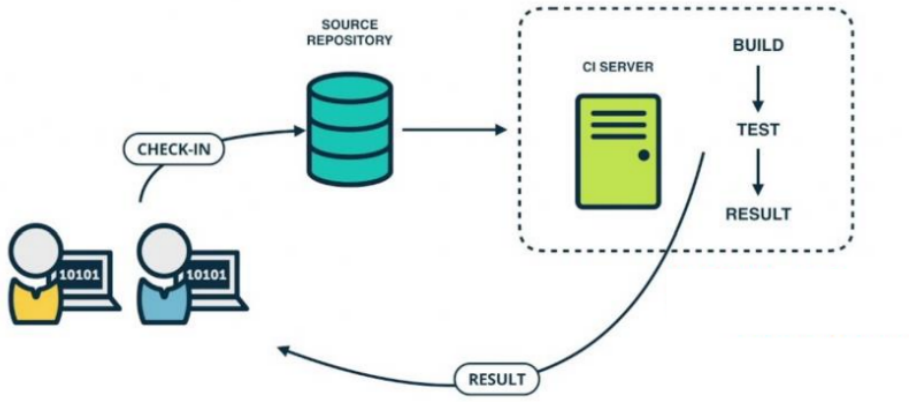

Рисунок 2.2 – Графічне відображення використання сервісу безперервної інтеграції коду

Безперервна інтеграція відноситься до стадії складання і модульного тестування процесу випуску ПЗ. Кожна підтверджена зміна коду запускає автоматичний процес складання та тестування.

Використання та підтримування процесу безперервної інтеграції дозволяє значно прискорити процес розробки та тестування ПЗ та збільшити ефективність використання часу розробки, підвищує продуктивність команди за рахунок звільнення розробників від ручної роботи, автоматизація заохочує до частіших внесень змін у вихідний код.

2.3.2 Безперервна доставка та розгортання коду

Безперервна доставка (continuous delivery/deployment, CD) – це практика розробки програмного забезпечення, коли при будь-яких змінах в програмному коді виконується автоматичний збір (build), тестування і підготовка до остаточного

випуску. Безперервна доставка є одним із основоположних принципів розробки сучасних додатків, оскільки розширює практику безперервної інтеграції за рахунок того, що всі зміни коду після стадії складання розгортаються в тестовому та/або робочому середовищі. При правильному впровадженні у розробників завжди буде готовий до розгортання зібраний екземпляр ПЗ, що пройшов стандартизовану процедуру тестування. Безперервна доставка дозволяє розробникам не тільки автоматизувати тестування на рівні модулів, але і виконувати різнопланову перевірку оновлень ПЗ перед тим, як розгортати його для кінцевих користувачів.

Все це дозволяє розробникам ретельніше перевіряти оновлення і завчасно виявляти можливі проблеми. На відміну від застарілих локальних рішень, хмарна середовище дозволяє легко і економічно автоматизувати створення і реплікацію декількох середовищ для тестування або навчання новому фунціоналу. Як показано на рис. 2.3, під час використання безперервної доставки кожна зміна програмного коду проходить збірку, тестується і потім вирушає в підготовчу (тестову або імітаційну) середу. Перед розгортанням у робочому середовищі можна використовувати кілька паралельних стадій тестування. Відмінність безперервної доставки від безперервного розгортання полягає в ступені автоматизації процесів, безперервне розгортання виконується повністю автоматично.

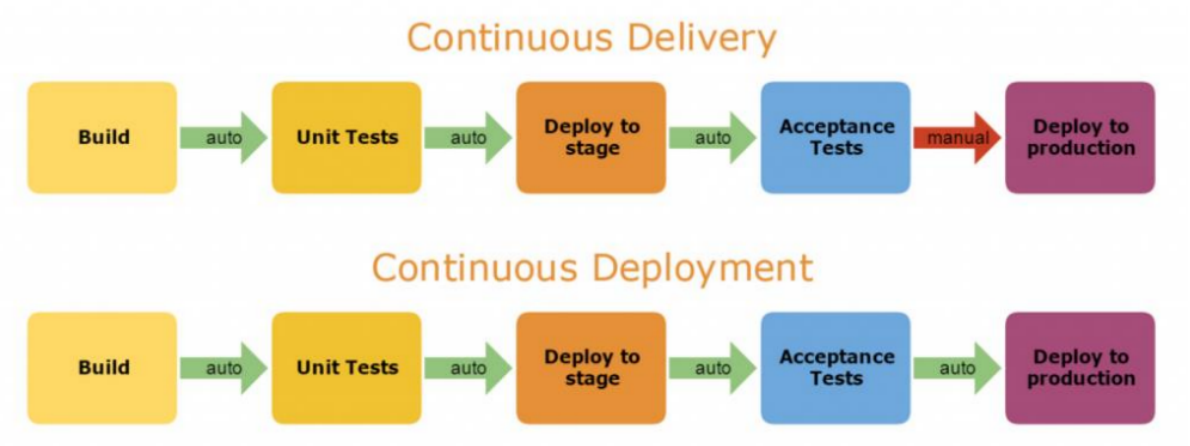

Рисунок 2.3 – Графічне відображення безперервної доставки та розгортання коду

Підтвердженням ефективності використання безперервної доставки є наступні переваги:

1. Автоматизація процесу випуску ПЗ: безперервна доставка дозволяє команді автоматично виконувати збірку, тестувати і підготувати зміни коду до запуску в робочому середовищі, що забезпечує більш ефективну і швидку доставку ПЗ.

2. Більш продуктивна розробка: застосування практики безперервної доставки підвищує продуктивність команди за рахунок звільнення розробників від ручної роботи і стимуляції підходів, які допомагають зменшити кількість помилок і дефектів в розгортанні для кінцевих користувачів.

3. Швидка доставка оновлень: безперервна доставка дає команді можливість доставляти оновлення кінцевим користувачам швидше і частіше. При правильному впровадженні безперервної доставки у команди завжди буде готовий до розгортання зібраний екземпляр ПЗ, що пройшов всі види автоматизованого тестування.

2.3.3 Безперервне тестування

Безперервне тестування – це процес виконання автоматизованих тестів у рамках конвеєра доставки програмного забезпечення для негайного зворотного зв’язку щодо бізнес-ризиків, пов'язаних з кандидатом на випуск програмного забезпечення. Причини виникнення безперервного тестування знову таки спираються на ті ж самі проблеми які виникли перед усіма сферами бізнесу, які так чи інакше пов’язані з IT. З початком широкого використання методології DevOps та Agile, організації очікують, що команди з розробки програмного забезпечення будуть готувати все більше і більше інноваційного програмного забезпечення протягом коротших циклів доставки. Отже, організації приймають рішення використовувати постійне тестування (рис. 2.4), оскільки вони визнають, що є ряд проблем, які заважають їм доставляти якісне програмне забезпечення з

потрібною швидкістю. Вони визнають зростаючу важливість програмного забезпечення, а також зростаючу вартість відмови програмного забезпечення, і більше не готові робити компроміс між часом, обсягом та якістю.

Рисунок 2.4 – Графічне відображення процесу безперервного тестування

Метою постійного тестування є забезпечення швидкого та постійного зворотного зв’язку щодо рівня бізнес-ризику в останньому кандидаті на складання чи реліз. Потім ця інформація може бути використана для того, щоб визначити, чи програмне забезпечення готове просуватись через конвеєр доставки в будь-який момент часу. Оскільки тестування починається рано і виконується постійно, ризики застосування піддаються негайному результату після їх введення. Потім команди розробників можуть запобігти переходу цих проблем на наступний етап CI/CD. Це зменшує час і зусилля, які потрібно витратити на пошук та виправлення дефектів. Як результат, можна збільшити швидкість і частоту, з якою постачається якісне ПЗ (програмне забезпечення, яке відповідає очікуванням прийнятного рівня ризику), а також зменшити технічну заборгованість. Більше того, коли зусилля щодо якості програмного забезпечення та тестування узгоджуються з очікуванням бізнесу, виконання тесту створює пріоритетний перелік діючих завдань (а не потенційно переважна кількість висновків, які потребують ручного огляду). Це допомагає командам зосередити зусилля на якісних завданнях, які матимуть найбільший вплив, виходячи з цілей та пріоритетів організації. Крім того, коли

команди постійно проводять широкий набір безперервних тестів по всьому циклу розробки, вони збирають показники щодо якості процесу, а також стану програмного забезпечення. Отримані показники можна використовувати для повторного вивчення та оптимізації самого процесу, включаючи ефективність цих тестів. Ця інформація може бути використана для встановлення циклу зворотного зв'язку, який допомагає командам поступово вдосконалювати процес. Часті вимірювання, постійний зворотній зв'язок та постійне вдосконалення є ключовими принципами DevOps.

Безперервне тестування включає перевірку як функціональних, так і нефункціональних вимог. Для тестування функціональних вимог (функціональне тестування) безперервне тестування часто передбачає тестування API та UI, тестування інтеграції та тестування системи. Для тестування нефункціональних вимог (нефункціональне тестування – щоб визначити, чи відповідає програма додаткові очікування щодо продуктивності, безпеки, відповідності тощо), вона передбачає такі практики, як аналіз статичного коду, тестування безпеки, тестування продуктивності тощо. Тести повинні бути розроблені для забезпечення якнайшвидшого виявлення (або запобігання) ризиків, які є найважливішими для бізнесу чи організації. Тестування проводяться під час або поряд із постійною інтеграцією (CI). Для команд, які практикують безперервну доставку, тести зазвичай виконуються багато разів на день, кожного разу, коли додаток оновлюється в системі контролю версій. В ідеалі всі тести виконуються в усіх невиробничих тестових середовищах. Для забезпечення точності та послідовності тестування слід проводити в найбільш повному, виробничо-подібному середовищі. Стратегії підвищення стабільності тестового середовища включають програмне забезпечення для віртуалізації (для залежностей, якими може керувати ваша організація та зображення), віртуалізацію послуг (для залежностей, що виходять за рамки вашої сфери контролю або непридатні для зображень) та управління тестовими даними [8].

Головні вимоги для використання безперервного тестування:

– тестування повинно бути співпрацею з розвитку, забезпечення якості та операцій, узгоджених з пріоритетами напряму бізнесу в рамках координованого, якісного та якісного процесу;

– тести повинні бути логічно-компонентними, поступовими та повторюваними; результати повинні бути детермінованими та значущими;

– усі випробування потрібно проводити в якийсь момент конвеєра, але не всі тести потрібно виконувати весь час;

– тестові дані та середовища мають бути незалежними, щоб тести могли постійно та послідовно виконуватись в усіх середовищах - тестових і продакшин.

2.3.4 Інтеграція CI/CD/CT у конвеєри

Конвеєр (англ. pipeline) – це процес, який керує розробкою програмного забезпечення через послідовність етапів збірки, тестування та розгортання коду, також відомий як CI/CD. Автоматизуючи цей процес, метою є мінімізація людських помилок та підтримання стійкого процесу для випуску програмного забезпечення. Інструменти, які включаються в пайплайн, можуть включати: компіляцію коду, модульні тести, аналіз коду, безпеку та створення бінарних файлів [8]. У випадку контейнеризованих середовищ, цей пайплайн також буде включати упаковку коду в образ контейнера для розгортання в гібридному хмарі.

При запуску конвеєра, запускається специфікація кроків, які потрібно було б виконати розробнику для впровадження нової версії програмного продукту. У відсутності автоматизованого пайплайну інженерам довелося б виконувати ці кроки вручну, що призводило б до значно меншої продуктивності.

Розглянемо схему конвеєра на схемі (рис. 2.4). Спочатку зміни вихідного коду попадають у репозитарій, CI конвеєр гарантує, що запускається збірка та тестування. Після того, він передається до CD конвеєру, де зміни у коді піддають рев’ю та далі розгортають в інші середовища та нарешті на виробництво.

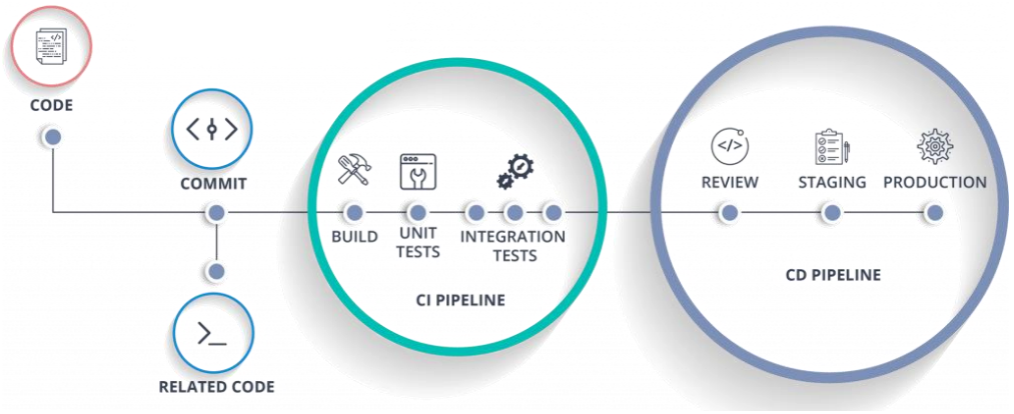

Рисунок 2.4 – Структурна схема конвеєру розробки сучаного ПЗ

2.4. GitOps – як різновид DevOps у хмарах

GitOps є одним із варіантів реалізації DevOps, який здійснює підтримку безперервної доставки та розгортання. Базовим елементом GitOps є IaC (Infrastructure as Code – інфраструктура як код) – модель відповідно якої опис і управління інфраструктурою проєкту здійснюється через конфігураційні файли, код, тобто програмно, на противагу ручному редагуванню конфігурацій на серверах чи інтерактивну взаємодію. Опис інфраструктури здійснюється в декларативному стилі, це дає змогу легко робити копії елементів інфраструктури або відновлювати її стан [9]. В даному випадку поняття IaC слід розуміти в широкому сенсі – воно включає в себе також Network as Code (мережа як код), Policy as Code (політики як код), Configuration as Code (конфігурація як код), Security as Code (безпека як код). Популярними інструментами, які дозволяють реалізувати IaC є Terraform, CloudFormation, файли-маніфести Kubernetes. Отже, замість того, щоб в ручному режимі створювати сервери, мережі, конфігурації, політики в AWS та Kubernetes-кластер з потрібними елементами, можна описати всю хмарну інфраструктуру за допомогою коду. В такому випадку отримаємо файли декларативних форматів (наприклад, yaml), які повністю описують інфраструктуру, платформу та їхню конфігурацію. GitOps передбачає версійність як програмного, так і конфігураційного коду шляхом розміщення коду інфраструктури у репозиторіях системи керування версіями, зазвичай Git, разом із

вихідним кодом програмного продукту. Git-репозиторій є єдиним джерелом істини в GitOps, єдине джерело істини – це практика структурування вихідого коду і пов'язаних даних таким чином, що кожен елемент даних редагується лише в одному місці (git-репозиторії). Всі зміни вихідного коду застосунку та інфраструктури здійснюються через git та механізм Pull Request (PR), що дозволяє забезпечити взаємодію усіх розробників, які повинні схвалити запропоновані зміни, у процесі рев'ю та тестування [9]. Це дозволить убезпечити систему від випадкових помилок, які можуть повністю її зламати. Таким чином, Git і поділ доступу в Git стає єдиним місцем (єдиним джерелом істини) не тільки для коду та інфраструктури, але і для політики доступу до різних частин проєкту.

Ще однією особливістю GitOps-підходу (рис. 2.5) є автоматизація процесу управління інфраструктурою на основі CI/CD. Система сама визначає та усуває різницю між тим, що перебуває в git-репозитарії та тим, що реально виконується на серверах. Тобто відбувається процес синхронізації git-репозиторія із станом Kubernetes-кластеру, який представляє собою сукупність вузлів, які запускають контейнеризовані програми.

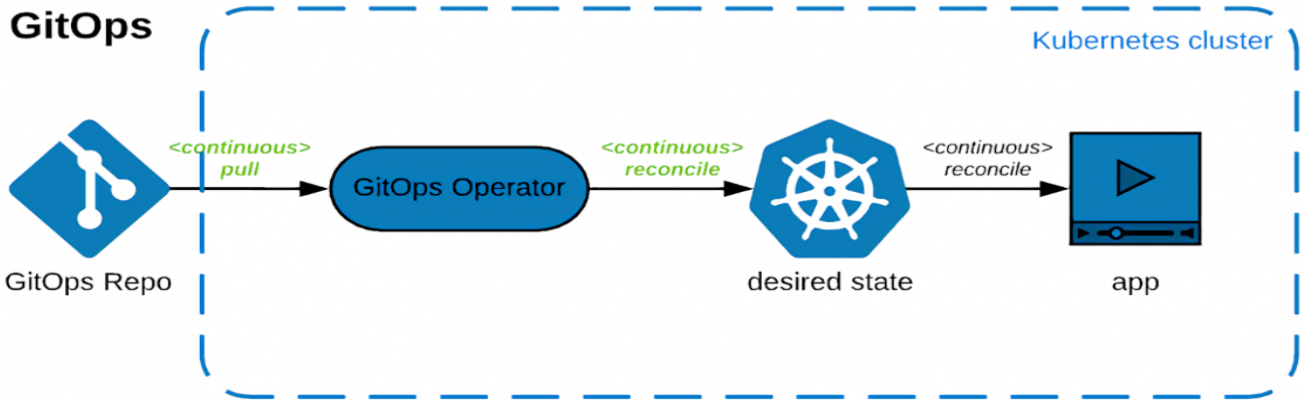

Рисунок 2.5 Схема GitOps-методології

Для підтримки процесу управління інфраструктурою зазвичай запускають спеціальну програму-агент, яка працює прямо в оточенні системи та змінює її зсередини. Агент є відповідальним за те, щоб здійснювати оновлення даних з системи контролю версій та при наявних змінах оновлювати стан Kubernetes-кластеру. Найпопулярнішими реалізаціями агентів є Weave Flux,

ArgoCD, Gitkube, Watchtower, keel.sh. На рисунку 2.6 наведемо діаграму процесу CD для Weave Flux.

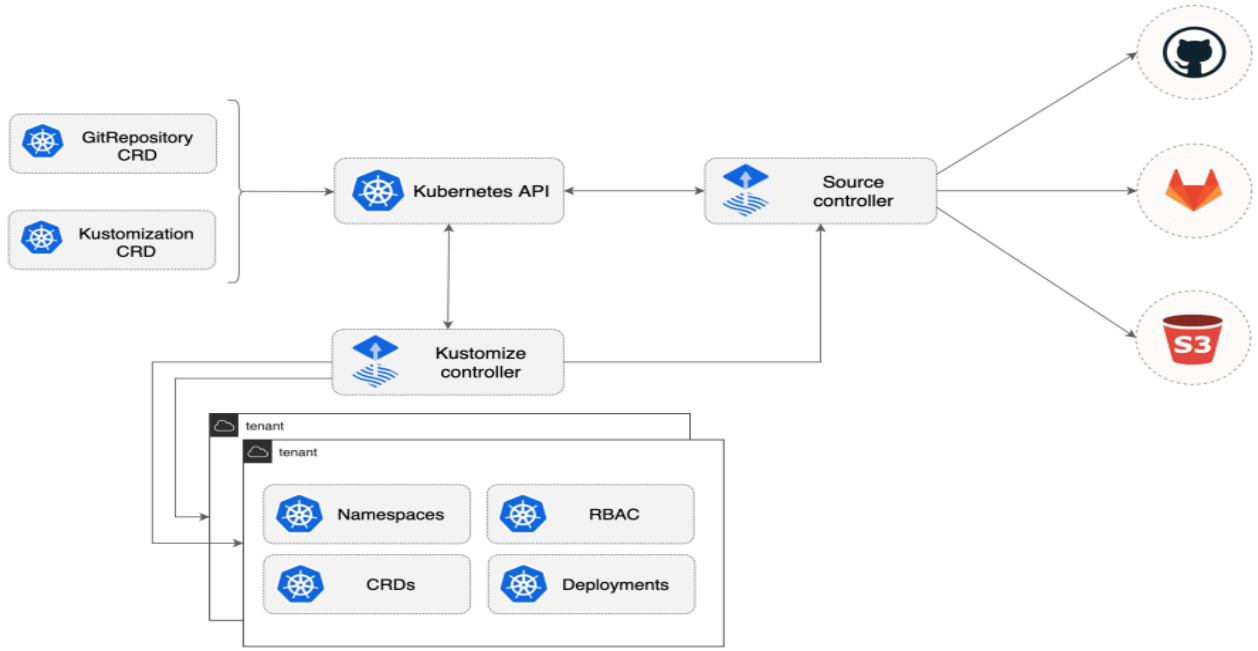

Рисунок 2.6 – Процес CD для Weave Flux [17]

Як можемо бачити з діаграми gitops-агент має два компоненти: source-контроллер та kustomize-контролер. Source-контролер є відповідальним за підтримку процесу авторизації в VCS, моніторингу змін в VCS, валідації цих змін у відповідності із синтаксисом мови розмітки (наприклад, yaml), створення CRD-об'єктів (Custom Resource Definition), що являють собою користувацькі Kubernetes-об'єкти на основі яких буде здійснюватися оновлення стану Kubernetes-кластеру. Kustomize-контролер здійснює оновлення стану кластеру на основі CRD-об'єктів, створених source-контролером, видалення Kubernetes-об'єктів відповідно до змін в VCS. Самі контролери знаходяться всередині Kubernetes та працюють як звичайні Kubernetes-оператори, які мають доступ до Kubernetes-API для того, щоб змінювати стан кластеру [9].

Розглянемо імплементацію GitOps за допомогою ArgoCD та GitLab (рисунок 2.7):

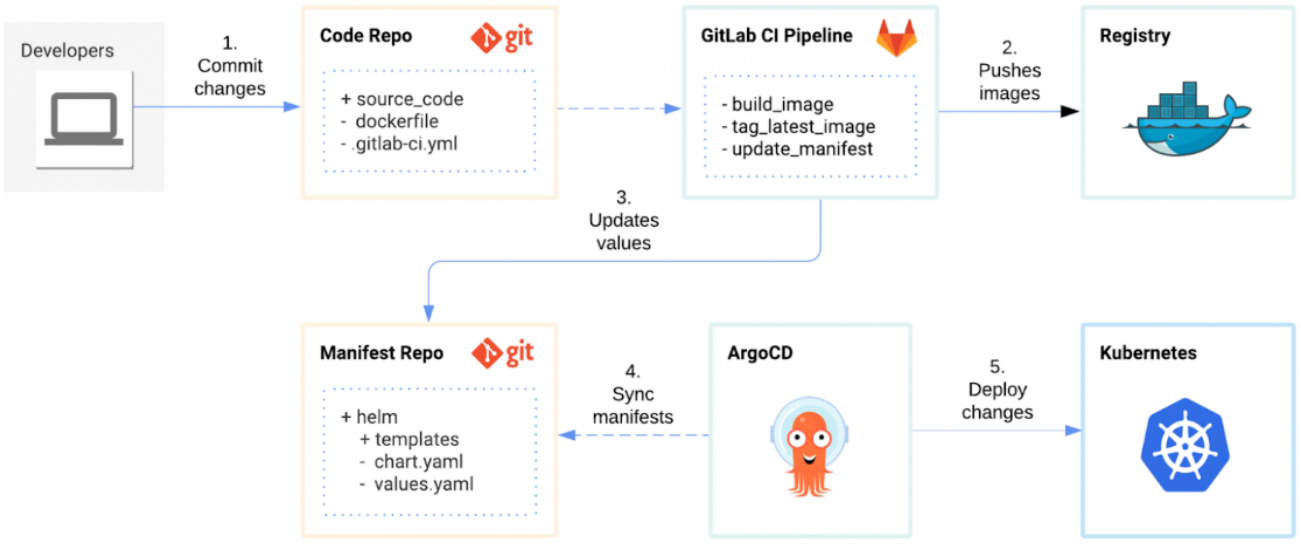

Рисунок 2.7 Еталоний варіант імплементації GitOps від ArgoCD

Еталоний варіант імплементації GitOps від ArgoCD пропонує використання двох Git-репозитаріїв. У першому репозиторії містяться вихідні коди додатка, а також Helm-чарти та Dockerfiles. Фактично, в цьому репозиторії знаходиться все, що не залежить від конкретного середовища. Другий репозиторій призначений для маніфестів – він описує конкретний додаток та його розгортання в кластері. Для продакшен середовища там є Helm-чарт із певними значеннями, а для стейджингу – з іншими. Helm тут використовується як приклад, насправді це можуть бути прості YAML-маніфести, розміщені у папках, або будь-який інший інструмент, який дозволяє отримати відрендерений YAML, готовий для розгортання додатка в середовище. При цьому ніщо не заважає нам налаштувати конвеєр, який буде оновлювати стан другого репозиторію з першого. Тут вже залежить від зручності кожного [10].

Другий репозиторій використовується GitOps-оператором як єдина точка правди – все, що в ньому описано, повинно бути синхронізовано з Kubernetes.

Основною задачею, яку покликаний вирішувати GitOps, є безперервна доставка та розгортання в Kubernetes-середовищі. Доставка (Continuous Delivery) в Kubernetes - це процес, який передбачає, що оператор, відповідальний за цей етап (системний адміністратор або спеціалізоване ПЗ) здійснює створення

Kubernetes-об’єктів, які власне і будуть відповідати за роботу необхідного ПЗ. Розгортання в Kubernetes-кластері може відбуватися такими способами:

- розгортання здійснює Kubernetes-кластер. В цьому випадку використовуються нативні Kubernetes-об’єкти, які є базовими;
- розгортання здійснюється завдяки спеціальному інструменту Kustomize. Це більш функціональний спосіб. У цьому випадку можна групувати Kubernetes-об’єкти. Kubernetes підтримує цей варіант без встановлення додаткових розширень та бібліотек;
- розгортання здійснюється завдяки спеціальному інструменту Helm. Це найбільш функціональний спосіб. Дозволяє групувати Kubernetes-об’єкти, застосовувати шаблонізацію, зберігати різні версії чартів – базових елементів Helm, що, як правило, складаються з опису метаданих чарту та файлів-шаблонів ПЗ, яке необхідно доставити та розгорнути. Для роботи Helm необхідно встановлення стороннього ПЗ, що являє собою виконуваний файл, який здійснює керування чартами (компіляцію, валідацію, завантаження з віддалених репозиторіїв).

Для вирішення задачі доставки та розгортання в Kubernetes-кластері найкраще використовувати комбінацію варіантів з Kustomize та нативними розгортаннями, які здійснює Kubernetes-кластер. Такий підхід дозволяє уникнути складнощів з встановленням та підтримкою роботи стороннього ПЗ. Також він є простим у розумінні та дозволяє вирішити будь-яку задачу з доставки та розгортання ПЗ в Kubernetes-кластері.

Отже, для задачі підвищення ефективності процесу автоматизації безперервної доставки та розгортання для багатокластерних систем найоптимальнішим є підхід GitOps. Серед основних переваг цього підходу, можна виділити наступні:

- зберігання кодової бази в одному місці;
- автоматизована доставка змін завдяки використанню gitops-агента;
- механізм PR, який дозволяє групувати зміни за задачами.

2.5. Інструменти DevOps для автоматизації процесів розробки, тестування та впровадження.

Одним із важливих факторів успіху DevOps є використання різноманітних інструментів, які сприяють безперебійній співпраці та автоматизації.

Інструменти DevOps можна розділити на кілька ключових категорій:

1. Continuous Integration (CI) і Continuous Deployment (CD) інструменти.

Приклади:

- Jenkins – сервер автоматизації з відкритим кодом, який допомагає автоматизувати різні етапи процесу розробки програмного забезпечення. Jenkins відомий своєю гнучкістю та розширюваністю, що дозволяє розробникам інтегрувати його з численними іншими інструментами та платформами. Завдяки величезній екосистемі плагінів Jenkins можна налаштувати відповідно до конкретних потреб будь-якого проекту.
- Travis CI – комерційний хмарний сервіс для автоматизації тестування та розгортання, добре підходить для невеликих проектів.
- GitLab CI/CD – інтегрований інструмент для автоматизації CI/CD процесів в GitLab.

2. Контейнеризація та оркестрація.

Приклади:

- Docker – платформа, яка спрощує процес створення, розгортання та запуску програм у контейнерах. Контейнери легкі, портативні та самодостатні, що дозволяє розробникам пакувати програму та її залежності в єдиний блок, який може працювати узгоджено в різних середовищах. Цей підхід усуває проблему “це працює на моєму комп’ютері”, забезпечуючи однакову поведінку програми в середовищах розробки, тестування та виробництва. Популярність Docker можна пояснити простотою використання, масштабованістю та сумісністю з різними платформами та інструментами.
- Kubernetes – є основним рішенням для оркестровки контейнерів, що дозволяє командам DevOps керувати розгортанням, масштабуванням і

обслуговуванням контейнерних програм. Kubernetes, розроблений Google, автоматизує процес розгортання, масштабування та керування контейнерами додатків, полегшуючи командам керування складними розподіленими системами. Його надійні функції, такі як самовідновлення, горизонтальне масштабування та поточні оновлення, роблять його важливим інструментом для керування контейнерними програмами в масштабі.

- OpenShift – корпоративна платформа для управління контейнерами, що базується на Kubernetes.

3. Інструменти для конфігурації та управління інфраструктурою.

Приклади:

- Ansible – програмне забезпечення, що надає засоби для управління конфігурацією, оркестровки, централізованої установки застосунків і паралельного виконання типових завдань на групі систем.

- Puppet – багатоплатформний клієнт-серверний застосунок, який дозволяє централізовано керувати конфігурацією операційних систем та програм, встановлених на кількох комп'ютерах. Puppet написано мовою програмування Ruby.

4. Інструменти для моніторингу та логування.

Приклади:

- Elastic Stack – включає Elasticsearch, Logstash і Kibana, забезпечує потужне та гнучке рішення для збору, обробки та візуалізації даних у режимі реального часу. Elasticsearch – це система розподіленого пошуку та аналітики, Logstash – конвеєр обробки даних, а Kibana – інструмент візуалізації та дослідження даних. Разом ці компоненти дають змогу командам отримати уявлення про свої програми та інфраструктуру, допомагаючи їм швидко виявляти та вирішувати проблеми.
- Prometheus – відкритий інструмент для моніторингу та алертингу.
- Grafana – інтерфейс для візуалізації даних моніторингу.

5. Системи керування версіями.

Приклади:

- Git – є однією з найефективніших, надійних і високопродуктивних систем керування версіями, що надає гнучкі засоби нелінійної розробки, що базуються на відгалуженні і злитті гілок.
- Subversion – централізована система (на відміну від розподілених систем, таких як Git або Mercurial), тобто дані зберігаються в єдиному сховищі. Сховище може розташовуватися на локальному диску або на мережевому сервері.

6. Інструменти автоматизованого тестування.

Приклади:

- Selenium – популярний інструмент для автоматизації тестування web-застосунків.
- JUnit – бібліотека для тестування програмного забезпечення для мови Java.
- Jest – популярний тестовий фреймворк для JavaScript.

Інструменти DevOps утворюють комплексну систему, що допомагає командам розробки автоматизувати, контролювати та оптимізувати процеси виробництва програмного забезпечення. Їхнє використання дозволяє компаніям швидше реагувати на зміни, покращувати якість та забезпечувати високий рівень надійності та ефективності програмного забезпечення.

РОЗДІЛ 3. ЗАСТОСУВАННЯ DEVOPS ПРАКТИК У РОЗРОБЦІ КОМЕРЦІЙНОГО ПРОЕКТУ У ХМАРНОМУ СЕРЕДОВИЩІ

3.1. Визначення етапів впровадження DevOps.

Впровадження DevOps – це ітеративний та еволюційний процес, який вимагає співпраці між командами розробки та операцій для досягнення автоматизації процесів розробки, тестування, впровадження та експлуатації, а також для поліпшення комунікації та зменшення часу циклу розробки.

Нижче приведем загальний план етапів впровадження DevOps:

1. Аналіз та планування:

- оцінка потреб: ретельний аналіз сучасних процесів розробки та впровадження, ідентифікація проблем та визначення вимог.

- визначення мети: встановлення конкретних цілей, які ви хочете досягти внаслідок впровадження DevOps.

- створення стратегії: розроблення стратегії впровадження, включаючи визначення кроків, необхідних для досягнення мети.

2. Залучення DevOps команди:

- створення команди: формування DevOps команди, яка буде відповідальною за впровадження та підтримку нових практик.

- освіта та тренінг: надання навчання та тренінгів для команди, щодо принципів та інструментів DevOps.

- впровадження культури: поширення культури співпраці, відкритості та взаєморозуміння в середовищі розробки та оперативного управління.

3. Вибір та впровадження інструментів:

- вибір інструментів: визначення необхідних інструментів для автоматизації процесів розробки, тестування, впровадження та моніторингу.

- налаштування інфраструктури: створення і налаштування інфраструктури для використання обраних інструментів.

- інтеграція з існуючими системами: забезпечення сумісності нових інструментів з існуючими системами.

4. Провадження автоматизації:

- автоматизація збірки та розгортання: впровадження автоматизованих процесів збірки, тестування та розгортання коду.

- автоматизація тестування: впровадження автоматизованих тестів для забезпечення високої якості коду.

- автоматизація моніторингу та логування: налаштування інструментів для автоматичного моніторингу та збору логів.

5. Запровадження принципів Continuous Integration та Continuous Deployment (CI/CD):

- CI/CD Pipelines: створення і вдосконалення CI/CD пайплайнів для автоматичного тестування та впровадження.

- оптимізація циклу розробки: зменшення часу циклу розробки через автоматизовані процеси.

- моніторинг CI/CD: постійне вдосконалення та моніторинг CI/CD пайплайнів.

6. Вдосконалення комунікації та співпраці:

- інтеграція команд: створення ефективного зв'язку та співпраці між розробниками, тестувальниками, операторами та іншими командами.

- використання спільних інструментів: впровадження спільних інструментів для обміну інформацією та спрощення співпраці.

7. Впровадження моніторингу та постійного вдосконалення:

- метрики та моніторинг: створення системи збору метрик та моніторингу для виявлення та усунення проблем.

- post-deployment аналіз: аналіз результатів розгортання для вдосконалення процесів.

- ретроспектива: проведення регулярних ретроспектив для оцінки ефективності та вдосконалення процесів.

8. Широке застосування та масштабування:

- розширення DevOps практик: розгортання DevOps підходів на більшу кількість проектів та відділень.

- масштабування інфраструктури: забезпечення горизонтального масштабування для обробки збільшеного навантаження.

- підтримка і поширення культури: поширення культури співпраці та автоматизації на всю організацію.

Ці етапи можуть варіюватися в залежності від конкретних потреб та контексту кожного проекту та організації. Важливо розглядати впровадження DevOps як постійний процес постійного вдосконалення.

3.2. Побудова архітектури CI/CD на прикладі GitLab

Розглянемо впровадження DevOps практик на прикладі GitLab.

GitLab – це хмарне веб-сховище Git, яке надає безкоштовні відкриті та приватні сховища, можливості відстеження проблем і вікі. Це повна платформа DevOps, яка дозволяє автоматизувати всі етапи розробки на проекті – від його планування та керування вихідним кодом до моніторингу та безпеки. Крім того, він дозволяє командам співпрацювати та створювати краще програмне забезпечення.

Для початку нам буде потрібний лінукс сервер, який буде використовуватись як GitLab сервер. Сервер може бути як окремий залізний сервер так і хмарний інстанс. Візьмемо для прикладу популярний дистрибутив лінукс – Ubuntu.

Нам буде потрібно встановити docker, це робиться за допомогою команди:

sudo apt install docker-ce

Далі завантажуємо інсталлятор:

curl -L

https://packages.gitlab.com/install/repositories/gitlab/gitlab-ce/script.deb.sh

Запускаємо скрипт:

sudo bash /tmp/script.deb.sh

Цей скрипт підготавлює сервер для використання GitLab репозитаріїв. Це надає змогу керувати GitLab тими самими інструментами керування пакетів що використовується для інших системних пакетів. Після виконання скрипту, можна відразу інсталювати актуальну версію GitLab за допомогою apt:

sudo apt install gitlab-ce

На цьому етапі ми маємо встановлений на сервері GitLab.

Далі можемо приступити безпосередньо до конфігурування репозитарія і побудові CI/CD пайплайнів, для цього необхідні наступні кроки:

1. Створення GitLab репозитарія:

   Зайдемо у GitLab інстанс і натиснемо кнопку *New project (рис. 3.1)*:

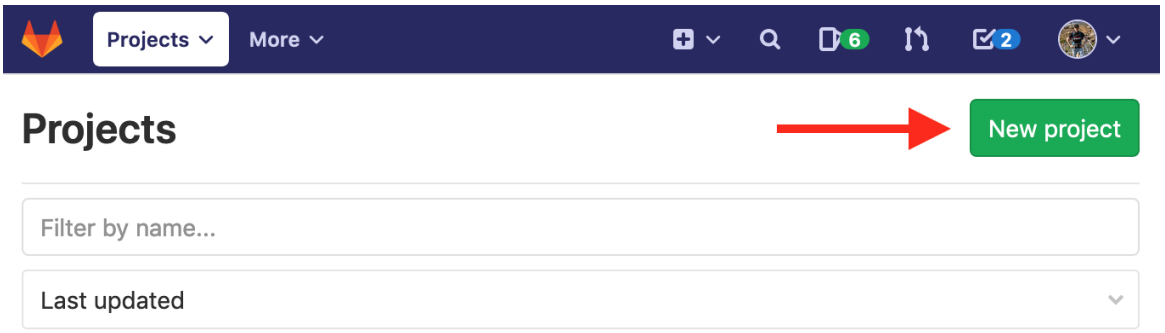

Рисунок 3.1 Створення нового проекту GitLab

Задамо ім’я проекта та опис, встановимо рівень видимості – приватний або публічний в залежності від потреб, накінець натиснемо *Create Project* (рис. 3.2).

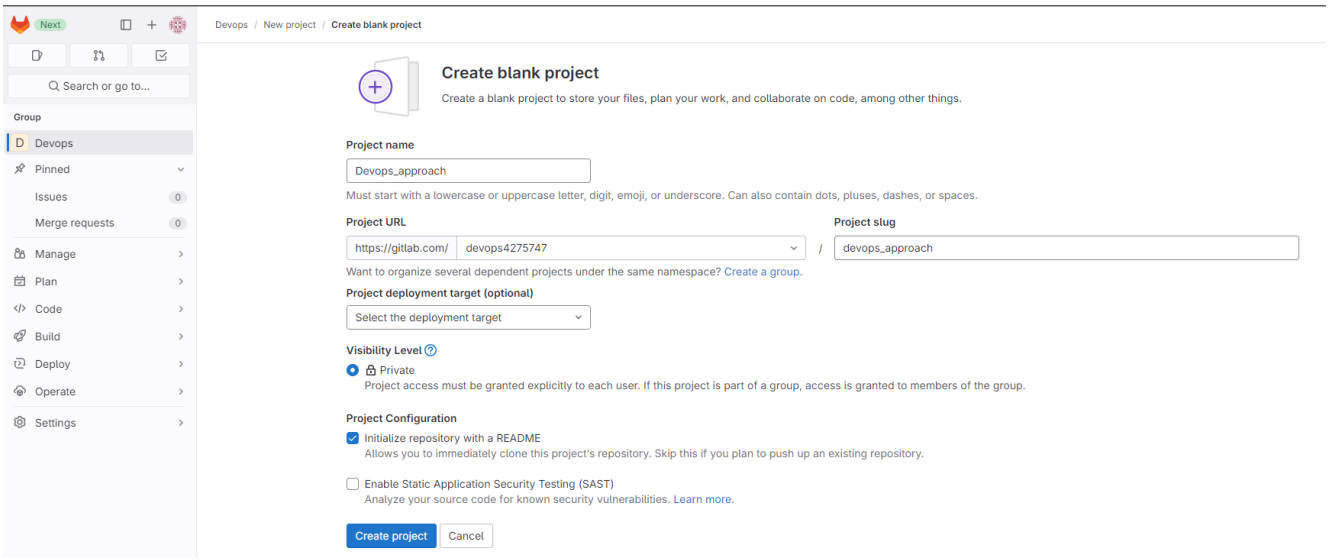

Рисунок 3.2 Створення нового проекту GitLab

Нас перенаправить на сторінку огляду проекта (рис. 3.3).

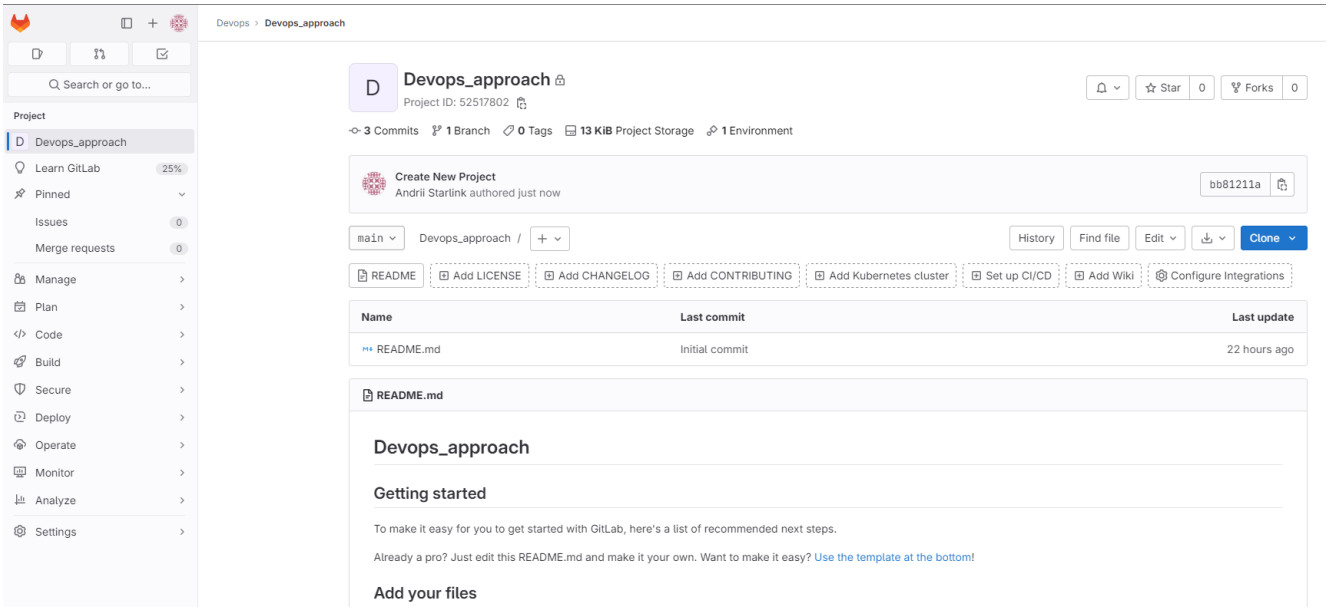

Рисунок 3.3 Огляд новоствореного проекту GitLab

2. Внесення вихідного коду у репозитарій.

Наступним кроком буде внесення вихідного коду у створений репозитарій. Для цього, натиснувши на кнопку *Clone* ми отримаємо адресу репозитарія, яку можна використати для внесення коду у репозитарій (рис. 3.4).

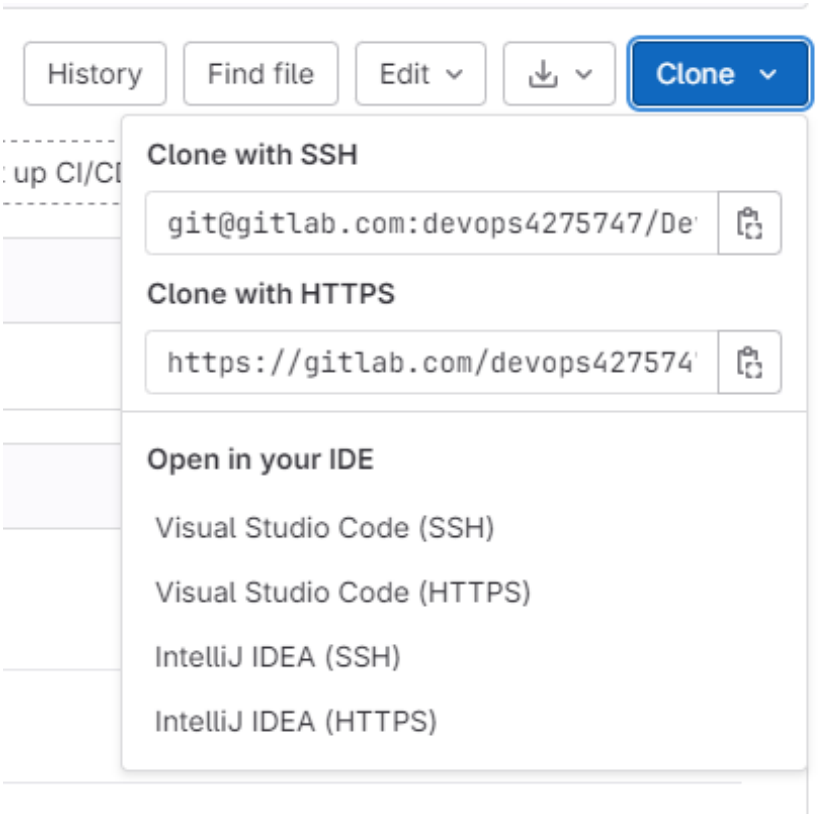

Рисунок 3.4 Clone віконце з адресою Git-репозитарію

3. Реєстрація GitLab Runner.

GitLab Runner – це агент, додаток з відкритим вихідним кодом, яке виконує завдання конвеєрної конвеєрної обробки GitLab CI/CD за спеціальними інструкціями.

Для відстеження середовищ, які матимуть зв'язок з SSH приватним ключем, нам потрібно зареєструвати свій сервер як GitLab runner. У пайплайнах CI/CD нам потрібно буде заходити на сервер за допомогою SSH. Для досягнення цього ми збережемо приватний ключ SSH у змінній GitLab CI/CD (Крок 5). Приватний ключ SSH є дуже чутливим елементом даних, оскільки він є квитком для входу на наш сервер. Зазвичай приватний ключ ніколи не залишає систему, на якій він був згенерований. У звичайному випадку генерується SSH-ключ на своєму хості, потім авторизуєте його на сервері (тобто копіюєте публічний ключ на сервер), щоб увійти вручну та виконати розгортання.

Тут ситуація трошки змінюється: ми хочемо надати автономній сутності (GitLab CI/CD) доступ до свого сервера для автоматизації процедур розгортання. Тому приватний ключ повинен залишити систему, на якій він був згенерований, і бути переданим у довіру GitLab та іншим зацікавленим сторонам. Ми ніколи не хочемо, щоб приватний ключ потрапив в середовище, яке не контролюється або не довіряється нами.

Крім GitLab, ще однією системою, до якої потрапить приватний ключ, буде GitLab агент. Для кожного конвеєра GitLab використовуємо раннери для виконання завдань, які вказано в конфігурації CI/CD. Це означає, що завдання розгортання в кінцевому рахунку буде виконане на GitLab runner, отже, приватний ключ буде скопійовано на runner так, щоб він міг увійти на сервер за допомогою SSH.

Для початку увійдемо на сервер та виконаємо команди для скачування і конфігурування агента:

curl -L

https://packages.gitlab.com/install/repositories/runner/gitlab-runner/script.deb.sh > script.deb.sh

sudo bash script.deb.sh

sudo apt install gitlab-runner

Перевіримо інсталляцію, для цього переглянемо статус:

systemctl status gitlab-runner

Має бути *active (running)* (рис. 3.5)

```
Output
● gitlab-runner.service - GitLab Runner
   Loaded: loaded (/etc/systemd/system/gitlab-runner.service; enabled; vendor preset:
   Active: active (running) since Mon 2020-06-01 09:01:49 UTC; 4s ago
 Main PID: 16653 (gitlab-runner)
    Tasks: 6 (limit: 1152)
   CGroup: /system.slice/gitlab-runner.service
           └─16653 /usr/lib/gitlab-runner/gitlab-runner run --working-directory /home
```

Рисунок 3.5 Відображення статусу *gitlab-runner* сервісу

Для реєстрації агента нам потрібно отримати токен проекту та URL GitLab:

- у нашому проекті GitLab перейдем до налаштування > *CI/CD* > *Runners*.
- у розділі “Set up a specific Runner manually” ми знайдемо токен реєстрації та URL GitLab (рис. 3.6). Скопіюємо обидва у текстовий редактор, вони знадобляться для наступної команди. Вони будуть вказані як https://your_gitlab.com та project_token.

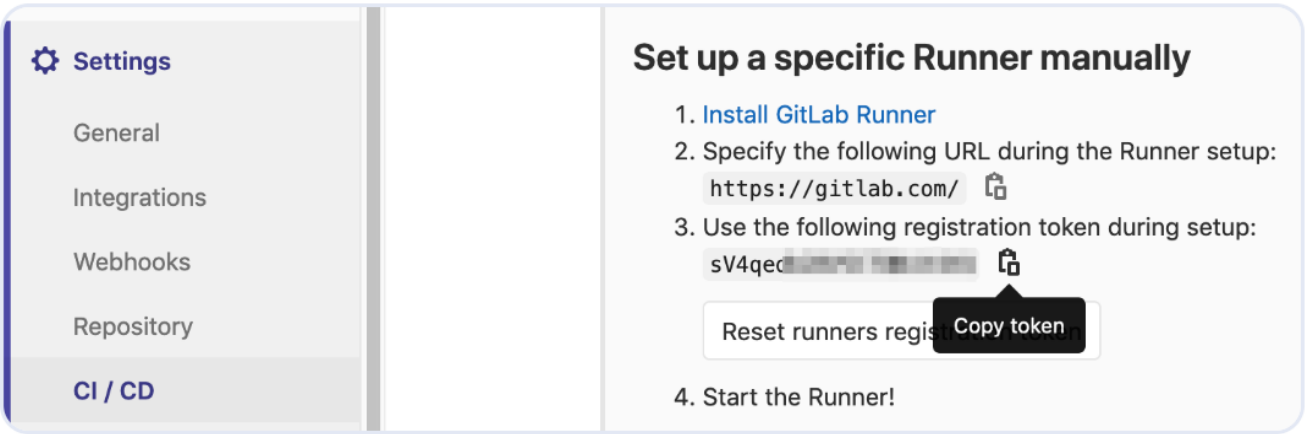

Рисунок 3.6 Налаштування агента GitLab

Повернемось до терміналу, зареєструємо агент для нашого проекту:

*sudo gitlab-runner register *

*-n --url https://your_gitlab.com --registration-token project_token *

*--executor docker --description "Deployment Runner" *

--docker-image "docker:stable" --tag-list deployment --docker-privileged

Після запуску команди маємо отримати успішне повідомлення про реєстрацію агента.

Перевіримо процес реєстрації, для цього перейдемо в Settings > CI/CD > Runners, зареєстрований агент має показатись (рис. 3.7):

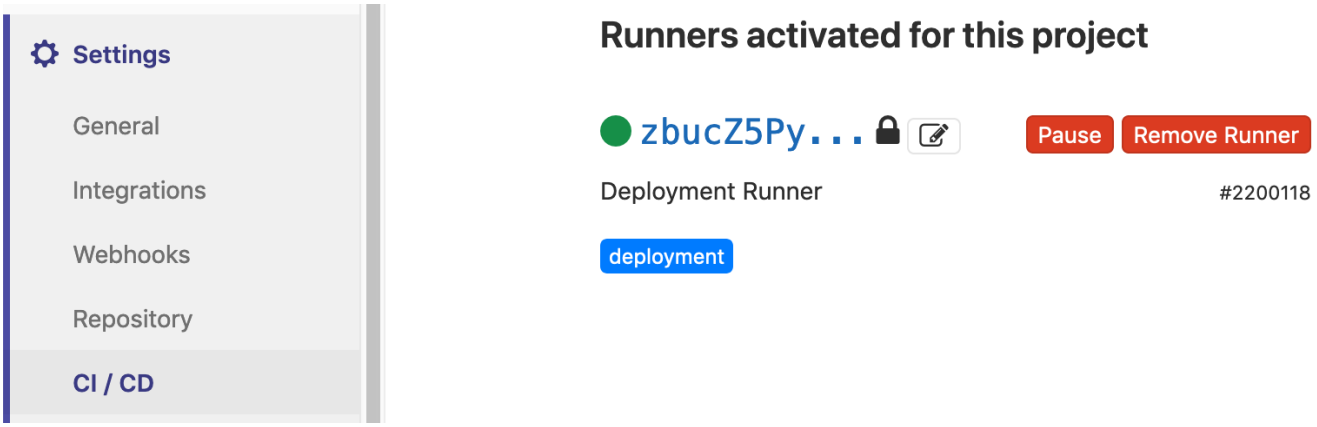

Рисунок 3.7 Активний статус агента GitLab

4. Створення користувача для розгортання.

Створимо користувача тільки для задачі розортання: для цього виконаємо команду створення користувача –

sudo adduser deployer

та додамо його у групу docker –

sudo usermod -aG docker deployer

Це дозволить оператору розгортання виконати команду *docker*, яка необхідна для виконання розгортання.

5. Створення SSH-ключа та збереження його в файлі змінних GitLab CI/CD.

Згенеруємо SSH-ключ за допомогою команди *ssh-keygen -b 4096*, відповідний публічний ключ помістимо в *authorized_keys* файл. Далі потрібно перейти в *Settings > CI / CD > Variables* на GitLab CI/CD та натиснути Add Variable, заповнити форму:

- Key: *ID_RSA*
- Mask variable: *Unchecked*
- Value: скопіюємо SSH приватний ключ за допомогою буфера (clipboard)
- Type: *File*
- Environment Scope: *All (default)*
- Protect variable: *Checked*
- Mask variable: *Unchecked*

6. Конфігурування .gitlab-ci.yml файла

Ми збираємося налаштувати конвеєр (pipelines) GitLab CI/CD. Конвеєр буде збирати Docker-образ і відправляти його до реєстру контейнерів. GitLab надає реєстр контейнерів для кожного проекту. Є можливість дослідити реєстр контейнерів, перейшовши до *Packages & Registries > Container Registry* у нашому проекті GitLab. Останній крок у нашому конвеєрі – увійти на сервер, витягти останній Docker-образ, видалити старий контейнер і запустити новий.

Створимо файл .gitlab-ci.yml, який містить конфігурацію конвеєру. В GitLab перейдемо на сторінку *Project overview*, натиснемо кнопку + і потім *New file*, встановлюємо ім'я файлу .gitlab-ci.yml. Повний вміст файла виглядатиме таким чином (рис 3.8).

Розглянемо більш детально файл конфігурації .gitlab-ci.yml. Кожне завдання призначено для певного етапу (stage). Завдання, призначені одному й тому самому етапу, виконуються паралельно (якщо доступно достатньо ранерів). Етапи будуть виконуватися в порядку, вказаному в конфігурації. Тут, етап публікації виконується першим, а етап розгортання другим. Наступні етапи починають виконуватися лише після успішного завершення попереднього етапу

(тобто після успішного проходження всіх завдань). Назви етапів можуть бути вибрані довільно.

```
stages:
  - publish
  - deploy

variables:
  TAG_LATEST: $CI_REGISTRY_IMAGE/$CI_COMMIT_REF_NAME:latest
  TAG_COMMIT: $CI_REGISTRY_IMAGE/$CI_COMMIT_REF_NAME:$CI_COMMIT_SHORT_SHA

publish:
  image: docker:latest
  stage: publish
  services:
    - docker:dind
  script:
    - docker build -t $TAG_COMMIT -t $TAG_LATEST .
    - docker login -u gitlab-ci-token -p $CI_BUILD_TOKEN $CI_REGISTRY
    - docker push $TAG_COMMIT
    - docker push $TAG_LATEST

deploy:
  image: alpine:latest
  stage: deploy
  tags:
    - deployment
  script:
    - chmod og= $ID_RSA
    - apk update && apk add openssh-client
    - ssh -i $ID_RSA -o StrictHostKeyChecking=no $SERVER_USER@$SERVER_IP "docker login -u gitlab-ci-token -p $CI_BUILD_TOKEN $CI_REGISTRY"
    - ssh -i $ID_RSA -o StrictHostKeyChecking=no $SERVER_USER@$SERVER_IP "docker pull $TAG_COMMIT"
    - ssh -i $ID_RSA -o StrictHostKeyChecking=no $SERVER_USER@$SERVER_IP "docker container rm -f my-app || true"
    - ssh -i $ID_RSA -o StrictHostKeyChecking=no $SERVER_USER@$SERVER_IP "docker run -d -p 80:80 --name my-app $TAG_COMMIT"
  environment:
    name: production
    url: http://your_server_IP
  only:
    - master
```

Рисунок 3.8 Лістінг файла .gitlab-ci.yml

Якщо ми хочемо поєднати цю конфігурацію CD з існуючим конвеєром CI, який тестує та будує додаток, нам може бути корисно додати етапи публікації та розгортання після існуючих етапів, таким чином, щоб розгортання відбувалося лише у випадку успішного проходження тестів.

Розділ "variables" визначає змінні середовища, які будуть доступні в контексті скрипту завдання. Ці змінні будуть доступні як звичайні змінні середовища Linux; іншими словами, ми можемо звертатися до них у скрипті, наприклад $TAG_LATEST. GitLab створює кілька заздалегідь визначених змінних для кожного завдання, які надають інформацію, специфічну для контексту, таку як ім'я гілки або хеш коміту, над яким працює завдання. В нас є такі змінні середовища:

- CI_REGISTRY_IMAGE: Представляє URL реєстру контейнерів, пов'язаний із конкретним проектом. Цей URL залежить від екземпляра GitLab. Наприклад, URL реєстру для проектів gitlab.com має вигляд: registry.gitlab.com/your_user/your_project. Але оскільки GitLab надасть цю змінну, нам не потрібно знати точний URL.
- CI_COMMIT_REF_NAME: Ім'я гілки чи тегу, для якого будується проект.
- CI_COMMIT_SHORT_SHA: Перші вісім символів хешу коміту, для якого будується проект.

Розділ "publish" є першим завданням у нашій конфігурації CI/CD. Давайте розглянемо його детальніше:

- image – це Docker-образ, який буде використовуватися для цього завдання. GitLab агент створить Docker-контейнер для кожного завдання та виконає скрипт у цьому контейнері, docker:latest забезпечує доступність команди docker.
- stage – присвоює завдання до етапу "publish".
- services – вказує Docker-in-Docker - сервіс dind. Це причина, чому ми реєстрували GitLab агент у привілейованому режимі.

Розділ "script" завдання "publish" вказує команди оболонки для виконання цього завдання. Робочий каталог буде встановлено в корінь репозиторію, коли ці команди будуть виконуватися.

- docker build ...: збирає Docker-образ на основі Dockerfile і позначає його останнім тегом коміту, визначеним у розділі "variables".
- docker login ...: увійти в систему Docker в реєстр контейнерів проекту. Ми використовуєм заздалегідь визначену змінну $CI_BUILD_TOKEN як маркер автентифікації. GitLab буде генерувати маркер і залишати його дійсним протягом часу виконання завдання.
- docker push ...: відправляє обидва теги образу до реєстру контейнерів.

Наступний розділ “deploy”.

Alpine - це легкий дистрибутив Linux, якого як раз вистачає для цієї задачі. Ми призначаємо завдання для етапу розгортання (deploy). Тег deployment забезпечує

виконання завдання на агентах, які мають тег deployment, таких як агент, який ми налаштували на кроці 3.

Розділ "script" завдання "deploy" починається з двох конфігураційних команд:

- chmod og= $ID_RSA: відкликає всі дозволи для групи та інших відносно приватного ключа, щоб тільки власник міг його використовувати. Це вимога, інакше SSH відмовиться працювати з приватним ключем.
- apk update && apk add openssh-client: оновлює менеджер пакетів Alpine (apk) та встановлює openssh-client, який надає команду ssh.

Наступні строки - чотири послідовні команди ssh. Шаблон для кожної з них такий:
ssh -i $ID_RSA -o StrictHostKeyChecking=no
$SERVER_USER@$SERVER_IP "command"
У кожному разі за допомогою ssh виконується команда на віддаленому сервері. Для цього виконується аутентифікуєтеся за допомогою приватного ключа.
Розгортання відбувається завдяки виконанню цих чотирьох команд на сервері:

- docker login ...: увійти в систему Docker в реєстр контейнерів.
- docker pull ...: витягнути останній образ з реєстру контейнерів.
- docker container rm ...: видалити існуючий контейнер, якщо він існує.
- docker run ...: запускає новий контейнер, використовуючи останній образ із реєстру. Контейнер буде названий my-app. Порт 80 на хості буде зв'язаний з портом 80 контейнера (-p host:container). -d запускає контейнер у фоновому режимі, інакше конвеєр застрягнув би, чекаючи завершення команди.

Середовища GitLab дозволяють керувати розгортаннями в межах GitLab. Ми можемо переглядати середовища у своєму проекті GitLab, перейшовши до Operations > Environments. Коли завдання конвеєра визначає розділ "environment", GitLab буде створювати розгортання для вказаного середовища (тут production) кожен раз, коли завдання успішно завершується. Це дозволяє вам відстежувати всі

розгортання, створені GitLab CI/CD. Для кожного розгортання можна побачити пов'язаний коміт та гілку, для якої воно було створено.

Розділ "only" визначає імена гілок та тегів, для яких виконуватиметься завдання. За замовчуванням GitLab запускатиме конвеєр для кожного пушу до репозиторію та виконуватиме всі завдання (з умовою, що файл .gitlab-ci.yml існує). Розділ "only" - це один з варіантів обмеження виконання завдання для певних гілок/тегів. Тут ми хочемо виконати завдання розгортання лише для гілки master.

Нарешті, натиснемо "Commit changes" внизу сторінки в GitLab, щоб створити файл .gitlab-ci.yml.

Ми створили конфігурацію GitLab CI/CD для збірки Docker-образу та розгортання його на сервер. На наступному етапі перевіримо правильність розгортання.

7. Перевірка розгортання

Тепер перевіримо розгортання в GitLab. Перейдемо до CI/CD > Pipelines у проекті GitLab, щоб переглянути статус конвеєра (рис. 3.9). Якщо завдання ще виконуються/очікують виконання, потрібно зачекати поки вони завершаться. Ми побачимо Passed конвеєр із двома зеленими галочками, що вказує на успішне виконання завдань publish та deploy.

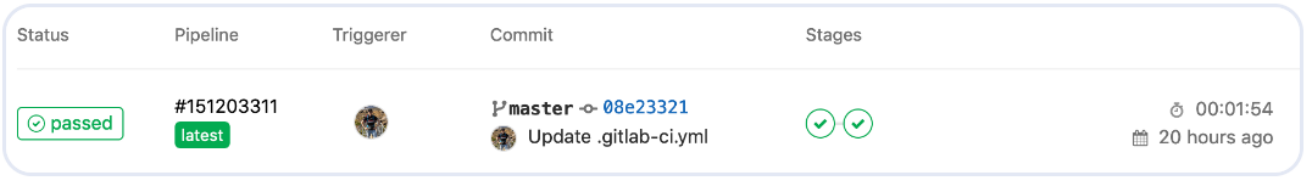

Рисунок 3.9 Перевірка стутуса конвеєра

Натиснемо deploy (перша зелена галочка), відкриється сторінка з результатами розгортання (рис. 3.10). Ми бачимо вивід оболонки для скрипта завдання. Це місце, куди слід звертатися при відлагодженні невдалих конвеєрів. У правій бічній панелі знаходиться тег розгортання, який ми додали до цього завдання, і як бачимо, що воно виконувалося на агенті розгортання.

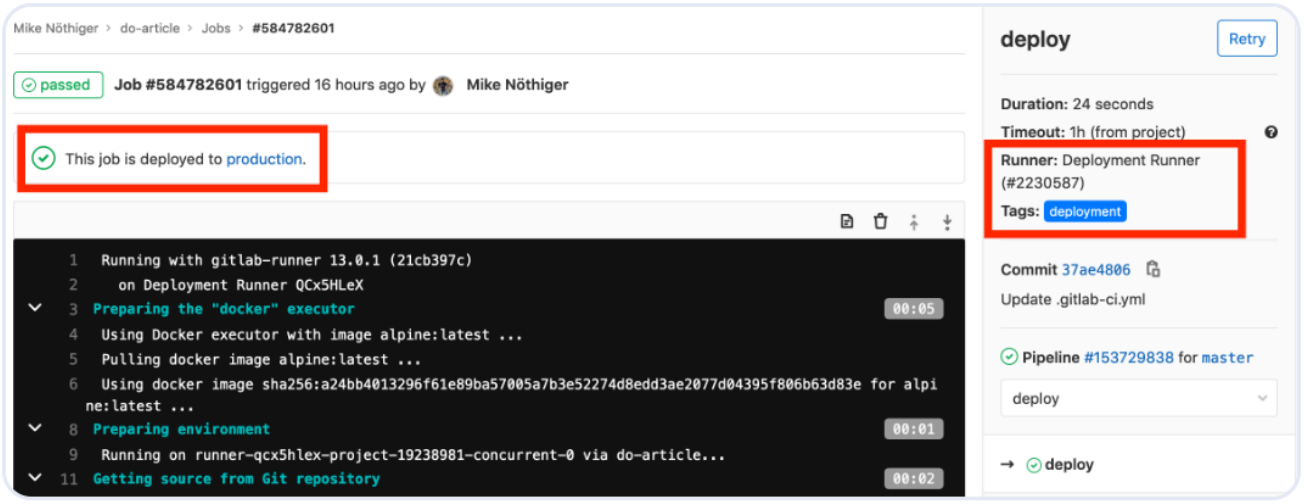

Рисунок 3.10 Перевірка логів розгортання

Зліва вверху є повідомлення "This job is deployed to production". GitLab визнає, що розгортання відбулося через розділ "environment" завдання. Натиснемо по посиланню production, щоб перейти до середовища production (рис. 3.11).

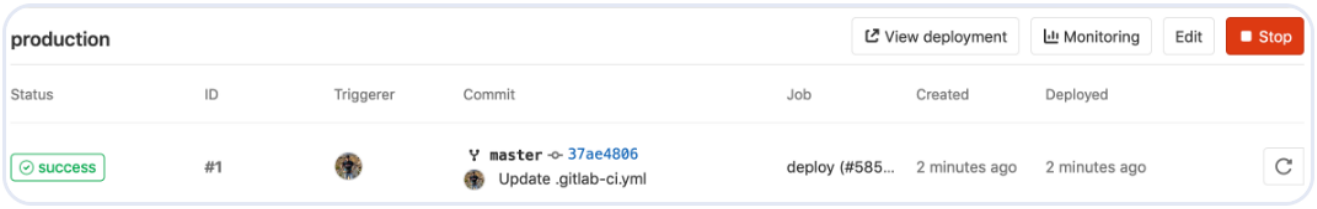

Рисунок 3.11 Перевірка стутуса розгортання у виробниче середовище

Тут ми бачимо огляд всіх розгортань в середовищі production. До цього часу було лише одне розгортання. Для кожного розгортання доступна кнопка повторного розгортання з самого правого боку. Повторне розгортання повторить завдання розгортання цього конвеєру. Далі, якщо натиснемо кнопку "View deployment", то відкриється http://your_server_IP у браузері, і ми повинні побачити наш веб-застосунок.

Таким чином ми налаштували конвеєр неперервного розгортання з використанням GitLab CI/CD. Наш конвеєр сконфігурований за допомогою файла .gitlab-ci.yml виконує такі кроки:

1. Збірка Docker-образу.

2. Відправка Docker-образу до реєстру контейнерів.

3. Входження на сервер, витягування останнього образу, запуск нового контейнера.

Тепер GitLab буде автоматично розгортати веб-сторінку на сервері для кожного пушу до репозиторію.

Крім того, ми перевірили розгортання в GitLab та на своєму сервері. Ми також створили друге розгортання і повернулися до першого, використовуючи середовища GitLab, що демонструє, як ви вирішуються проблеми з розгортанням.

На цьому етапі ми автоматизували весь ланцюг розгортання. У результаті цикли розробки стануть коротшими, оскільки для збірки та публікації змін коду буде потрібно менше часу. Звичайно ми навели приклад дуже простого конвеєра. На реальному проекті до конвеєру будуть додаватись десятки різних завдань, таких як: юніт тести, перевірка стилю коду (linters), перевірка якості коду (SonarQube), інтеграційні тести та багато інших, які безпосередньо впливають на якість та продуктивність розробки взагалом. Тепер, замість того щоб виконувати всі ці етапи вручну для того щоб побачити зміни, які були внесені девелоперами, нам залишиться тільки передивлятися стан конвеєру та при необхідності дивитись логи етапів конвеєру.

ВИСНОВКИ

В ході дипломної роботи було досліджено вплив застосування DevOps практик на розробку інформаційних систем у хмарному середовищі. У першому розділі було розглянуто специфіку хмарних інформаційних систем. Був зроблений аналіз проблем, які виникають при розробці хмарних інформаційних систем та як їх можливо мінімізувати за допомогою гнучких моделей розробки.

Другий розділ було присвячено DevOps підходу до розробки інформаційних систем. Наведено поняття та принципи DevOps, як DevOps застосовується у розробці програмного забеспечення. Визначені основні практики DevOps, такі як: Continuous Integration (безперервна інтеграція), Continuous Deployment (безперервне розгортання), Continuous Testing (безперервне тестування). Також розглянутий GitOps – дуже важливий підхід у розробці ІС у хмарному середовищі. Перелічені інструменти DevOps для автоматизації всіх процесів розробки.

Далі, у рамках третього розділу було досліджено як саме відбувається впровадження DevOps у розробку хмарних інформаціних систем. Наведено етапи впровадження DevOps. Далі на прикладі CI/CD платформи GitLab був побудований конвейєр для інтеграції та розгортання ІС, який ілюструє DevOps підхід на практиці, виводить девелоперські та операційні процеси на більш якісний рівень та при цьому команді розробників необхідо витрачати меньше зусиль.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. P. Mell, T. Grance, The NIST Definition of Cloud Computing. URL: https://csrc.nist.gov/pubs/sp/800/145/final
2. T. Erl, R. Puttini, Z. Mahmood, Cloud Computing: Concepts, Technology & Architecture 2nd Edition, 2023
3. Intelli Paat: AWS vs Azure vs Google Cloud: Choosing the Right Cloud Platform. URL : https://intellipaat.com/blog/aws-vs-azure-vs-google-cloud
4. Chaput, Shawn R., and Katarina Ringwood. "Cloud compliance: A framework for using cloud computing in a regulated world." Cloud computing: Principles, systems and applications, 2010
5. G. Blokdyk, Software Development Methodologies A Complete Guide, 2020
6. K. Rigby, S. Elk, S. Berez, Doing Agile Right: Transformation Without Chaos, 2020
7. L. Bass, I. Weber, L. Zhu, DevOps. A software architect`s perspective, 2015
8. M. Loukides, What is DevOps? O'Reilly Media, Inc., 2012.
9. T. A. Limoncelli, GitOps: A Path to More Self-service IT: IaC+ PR= GitOps, 2018
10. L. Costea, S. Economakis, Argo CD in Practice: The GitOps way of managing cloud-native applications, 2022
11. GitLab Handbook, URL: https://about.gitlab.com/handbook/