Міністерство освіти і науки України
Національний технічний університет
«Дніпровська політехніка»

Інститут електроенергетики
(інститут)

Факультет інформаційних технологій
(факультет)

Кафедра Програмного забезпечення комп'ютерних систем
(повна назва)

ПОЯСНЮВАЛЬНА ЗАПИСКА
кваліфікаційної роботи ступеня

магістра
(назва освітньо-кваліфікаційного рівня)

студента	*Старишко Олександра Сергійовиач*
	(ПІБ)
академічної групи	*122М-22-4*
	(шифр)
спеціальності	*122 Комп'ютерні науки*
	(код і назва спеціальності)
освітньої програми	*«122 Комп'ютерні науки»*
	(назва освітньої програми)
на тему:	Дослідження інструментів та практики безперервної розробки та розгортання програмного забезпечення в сфері DevOps

_____*О.С. Старишко*

Керівники	**Прізвище, ініціали**	**Оцінка за шкалою**		**Підпис**
		рейтинговою	**інституційною**	
розділів кваліфікаційної роботи				
спеціальний	*доц. Спірінцев В.В.*			
економічний				
Рецензент				
Нормоконтролер	*проф. Лактіонов І.С.*			

Дніпро
2023

Міністерство освіти і науки України
Національний технічний університет
«Дніпровська політехніка»

ЗАТВЕРДЖЕНО:
Завідувач кафедри
Програмного забезпечення комп'ютерних систем
(повна назва)

(підпис) М.О. Алексєєв (прізвище, ініціали)

« » 20 23 року

ЗАВДАННЯ
на виконання кваліфікаційної роботи

спеціальності *122 Комп'ютерні науки*
(код і назва спеціальності)

студенту *122м-22-4* (група) Старишко Олександру Сергійовичу (прізвище та ініціали)

Тема кваліфікаційної роботи Дослідження інструментів та практик безперервної розробки та розгортання програмного забезпечення в сфері DevOps

1 ПІДСТАВИ ДЛЯ ПРОВЕДЕННЯ РОБОТИ

Наказ ректора НТУ «Дніпровська політехніка» від 09.10.2023 р. № 1227 - с

2 МЕТА ТА ВИХІДНІ ДАНІ ДЛЯ ПРОВЕДЕННЯ РОБІТ

Об'єкт досліджень – є процес безперервної розробки та розгортання програмного забезпечення.

Предмет досліджень – методології та підходи безперервної розробки та розгортання коду, інструменти в DevOps процесах.

Мета НДР – є підвищення ефективності процесу розробки програмного забезпечення за рахунок використання сучасних підходів self-serve DevOps та headless і інструментів CI/CD в них.

Вихідні дані для проведення роботи – теоретичні та практичні дослідження, використання підходів безперервної інтеграції та розгортання коду для покращення та налагодження процесу розробки коду в компанії.

3 ОЧІКУВАНІ НАУКОВІ РЕЗУЛЬТАТИ

Новизна запропонованих рішень отримав подальший розвиток напрямок безперервного розгортання та розробки коду, що спирається на застосування нових підходів self-serve DevOps та headless.

Практична цінність полягає в запропонованих сучасних для безперервного розгортання та розробки коду підходах self-serve DevOps та headless, які забезпечують

підвищену ефективність розробку програмного забезпечення в компанії завдяки налаштованому процесу доставки коду до релізу.

4 ВИМОГИ ДО РЕЗУЛЬТАТІВ ВИКОНАННЯ РОБОТИ

Результати досліджень мають бути подані у вигляді, що дозволяє побачити та оцінити безпосереднє використання CI/CD практик та інструментів. В результаті роботи повинен бути розроблений CI/CD pipeline для кращого розуміння, як будуються та використовуються DevOps підходи та інструменти на проекті.

5 ЕТАПИ ВИКОНАННЯ РОБІТ

Найменування етапів робіт	**Строки виконання робіт (початок – Кінець**
Аналіз теми та постановка задачі	12.10.2023 – 31.10.2023
Аналіз сфери DevOps. Дослідження іструментів, підходів та стратегій, особливості їх використання.	01.11.2023 – 20.11.2023
Створення практично прикладу CI/CD pipeline, для більш детального дослідження DevOps.	21.12.2023 – 06.12.2023

6 РЕАЛІЗАЦІЯ РЕЗУЛЬТАТІВ ТА ЕФЕКТИВНІСТЬ

Економічний ефект від реалізації підходів очікується позитивним завдяки налагодженому та якісному процесу оновлення програми, контролю продуктивності завдяки постійному моніторингу у реальному часі, та економічно правильно побудованій інфраструктурі проекту.

Соціальний ефект від реалізації результатів роботи очікується позитивним завдяки DevOps методологіям впроваджених в компанію. Культура розробки програмного забезпечення DevOps сприяє кращому спілкуванню та співпраці в командах, оскільки їхня увага зосереджена на продуктивності.

7 ДОДАТКОВІ ВИМОГИ

Завдання видав _____ *Спірінцев В.В.*
(підпис) (прізвище, ініціали)

Завдання прийняв до виконання _____ *Старишко О.С.*
(підпис) (прізвище, ініціали)

Дата видачі завдання: *09.10.2023*

Термін подання кваліфікаційної роботи до ЕК *06.12.2023 р.*

РЕФЕРАТ

Пояснювальна записка: 84 стор., 61 рис., 4 таблиці, 2 додатка, 23 джерел.

Об'єкт дослідження: процес безперервної розробки та розгортання програмного забезпечення.

Предмет дослідження: методології та підходи безперервної розробки та розгортання коду, інструменти в DevOps процесах.

Мета роботи: підвищення ефективності процесу розробки програмного забезпечення за рахунок використання сучасних підходів self-serve DevOps та headless і інструментів CI/CD в них.

Методи дослідження. З метою досягнення поставленої мети та вирішення завдань у кваліфікаційній роботі було застосовано методи: абстрактно-логічний – для теоретичного узагальнення та формулювання висновків; аналітичний – для порівняння підходів та інструментів з безперервної інтеграції та розгортання коду, теорії баз даних – для побудови баз даних.

Новизна отриманих результатів. Отримуємо подальший розвиток напрямку безперервного розгортання та розробки коду, за рахунок застосування нових підходів self-serve DevOps та headless.

Практична цінність. Полягає в запропонованих типових для безперервного розгортання та розробки коду рішеннях, які забезпечують підвищену ефективність розробку програмного забезпечення в компанії завдяки налаштованому процесу доставки коду до релізу.

Область застосування. Усі DevOps процеси, методології та іструменти поцілені на те, щоб збалансувати потреби компанії протягом розробки та забезпечення життєвого циклу продукту, від кодування та розгортання до обслуговування та оновлень .Тому DevOps можна сміло впроваджувати в майже любі види проектів, від середніх до великих.

Значення роботи та висновки. Розглянуті DevOps методології, стратегії та інструменти дозволяють проектувати CI/CD pipeline, які допомагають покращити роботу на проекті, якість коду, створити повністю автоматизовану доставку та розгортання додатку, а також зменшити матеріальні витрати, так і часові, що підтверджується розробленим нами CI/CD pipeline.

Прогнози щодо розвитку досліджень. Досліджуючи DevOps на теоретичному та практичному прикладу, дізнаємося про можливі недоліки та шляхи покращення. Таким чином отримуємо подальший розвиток напрямку безперервного розгортання та розробки коду, можливість використання нових технологій та методологій.

Список ключових слів: CI/CD, pipeline, DevOps, Git, GitLab, CVS, SVN, GitFlow, GitHub Flow, GitHub, Trunk-Based, Code Review, Quality Gateway, Code Analysis, feature, hotfix, master, unit test, Docker, Kubernetis, QA, UAT, staging, IaC, API, VM, Terraform, Ansible, AWS, GCP, Azure, Recreate, Blue/Green, Canary, A/B testin, ПЗ, БД.

ABSTRACT

Explanatory note: 84 pages, 61 figures, 4 tables, 2 applications, 23 sources.

Object of research: the task of research is to find the best approaches and tools in our opinion, which will help us implement continuous development and deployment of our application faster and more efficiently.

Subject of research: methodologies and approaches of continuous code development and deployment, tools in DevOps processes.

Purpose of Master's thesis: development and support of a software tool on the Internet platform using CI/CD approaches and tools. Researching the best CI/CD techniques and options for our project, as well as choosing the best tools.

Research methods: In order to achieve the set goal and solve the tasks in the qualification work, methods were used: abstract-logical - for theoretical generalization and formulation of conclusions; analytical - for comparing approaches and tools for continuous integration and code deployment, database theory - for building databases.

Originality of research: We receive further development of the direction of continuous deployment and code development, due to the use of new self-serve DevOps and headless approaches.

Practical value of the results: it consists of proposed solutions typical for continuous deployment and code development, which ensure increased efficiency of software development in the company thanks to a customized process of delivering code to release

Scope of application: All DevOps processes, methodologies, and tools aim to balance a company's needs throughout the development and provisioning lifecycle of a product, from coding and deployment to maintenance and upgrades. Therefore, DevOps can be confidently implemented in almost any type of project, from medium to large.

The value of the work and conclusions: The considered DevOps methodologies, strategies and tools allow you to design a CI/CD pipeline that helps improve work on the project, code quality, create a fully automated delivery and deployment of the application, as well as reduce material costs and time, which is confirmed by our developed CI/CD pipeline.

Research forecast and development: Studying DevOps on a theoretical and practical example, we learn about possible shortcomings and ways to improve. In this way, we get further development of the direction of continuous deployment and code development, the possibility of using new technologies and methodologies.

Keywords: CI/CD, pipeline, DevOps, Git, GitLab, CVS, SVN, GitFlow, GitHub Flow, GitHub, Trunk-Based, Code Review, Quality Gateway, Code Analysis, feature, hotfix, master, unit test, Docker, Kubernetis, QA, UAT, staging, IaC, API, VM, Terraform, Ansible, AWS, GCP, Azure, Recreate, Blue/Green, Canary, A/B testin, Software, Database.

ПЕРЕЛІК УМОВНИХ ПОЗНАЧЕНЬ

DevOps – розробка/експлуатація системи;

QA – гарантія якості (тестувальник);

UAT – приймальне тестування;

IaC – Інфраструктура як код;

API – інтерфейс програмування застосунків;

VM – віртуальна машина;

БД - база даних;

ПЗ - програмне забезпечення;

CI/CD/CD/CO - безперервна інтеграція/доставка/розгортання/підтримка;

ЗМІСТ

ВСТУП

Актуальність теми. Зараз інтернет є дуже великою платформою для бізнесу, а в бізнесі дуже важливо щоб потік праці не зупинявся, бо будь-які зупинки можуть коштувати великих затрат. Так як в наш час, майже всі бізнес продукти, чиїсь стартапи тощо, перейшли до інтернету, виникла велика потреба в надійності, швидкості та безперебійній роботі всіх додатків, які працюють на платформі інтернет. Ні для кого не є секретом, що всі сучасні програми перед кінцевим розташуванням повинні пройти тести на помилки, на якість тощо. Код який використовується для їх написання повинен постійно надходити від одного розробника до іншого, і вдало з'єднуватися та передаватися у подальший реліз. У свою чергу треба достатньо часу для того щоб весь готовий продукт упакувати та розташувати в інтернеті. З цього ми робимо висновок, що на все це нам потрібно десь брати час, тож було б гарно усе це діло якось автоматизувати. Саме цим питанням і займається DevOps розробник. Такі спеціалісти дуже гарно знаються на методології Continuous Integration & Continuous Delivery/Deployment (CI/CD) ("безперервна інтеграція, доставка та розгортання"). Саме CI/CD стає нашим порятунком. Ця концепція допомагає вдало з'єднувати код, робити автоматичне тестування, а також автоматично розгортати та розташовувати наш готовий продукт у сітку інтернету. Методологія CI/CD так само дуже схожа на конвеєр, який допомагає оптимізувати процес доставки коду, підвищуючи якість усього цього дійства.

Мета дослідження. Розробка та підтримка демонстративного програмного забезпечення на платформі інтернет з використанням підходів та інструментів CI/CD. Дослідження сучасних підходів self-serve DevOps та headless і інструментів CI/CD в них.

Завдання дослідження. Завданням дослідження є пошук кращих на наш погляд підходів та інструментів, які допоможуть нам реалізувати безперервну розробку та розгортання нашої програми швидше та ефективніше.

Об'єкт дослідження. Процес безперервної розробки та розгортання програмного забезпечення.

Предмет дослідження. Методології та підходи безперервної розробки та розгортання коду, інструменти в DevOps процесах.

Методи дослідження. З метою досягнення поставленої мети та вирішення завдань у кваліфікаційній роботі було застосовано методи: абстрактно-логічний – для теоретичного узагальнення та формулювання висновків; аналітичний – для порівняння підходів та інструментів з безперервної інтеграції та розгортання коду, теорії баз даних – для побудови баз даних.

Новизна запропонованих рішень. Отримуємо подальший розвиток напрямку безперервного розгортання та розробки коду, за рахунок застосування нових підходів self-serve DevOps та headless.

Практичне значення. Полягає в запропонованих типових для безперервного розгортання та розробки коду рішеннях, які забезпечують підвищену ефективність розробку програмного забезпечення в компанії завдяки налаштованому процесу доставки коду до релізу.

Структура та обсяг кваліфікаційної роботи. Відповідно до мети, задач і предмета дослідження, кваліфікаційна робота складається з реферату, вступу, трьох основних розділів і висновків, списку використаних джерел та 2 додатків. Загальний обсяг роботи містить 84 сторінок друкованого тексту, із них основна частина - 62 сторінки з 61 рис., спеціальна – 89 сторінок, списку використаних джерел з 61 найменувань на 3 сторінках, 2 додатки на 10 сторінках.

РОЗДІЛ 1
ТЕОРЕТИЧНІ АСПЕКТИ DEVOPS-РОЗРОБКИ

1.1. Понятійний апарат інтеграції та розгортання коду

Розглянемо термін CI/CD, а також проаналізуємо його вплив на сферу розробки в цілому. Дуже часто до терміну CI/CD добавляють слово pipeline (CI/CD pipeline). CI/CD тут виступає методологією, як робити правильно, а pipeline – це “конвеєр”, фізичний термін (якщо так можна мовити) в розробці. Далі поговоримо про це більш розгорнуто.

Безперервна інтеграція (CI, Continuous Integration) та безперервна доставка (CD, Continuous Delivery) або безперервне розгортання (CD, Continuous Deployment) (рис. 1.1). Всі ці терміни представляють собою культуру, набір принципів та практик, які дозволяють швидше розробляти та надійніше розвертати зміни програмного забезпечення. Також їх іноді називають безперервною розробкою програмного забезпечення[1].

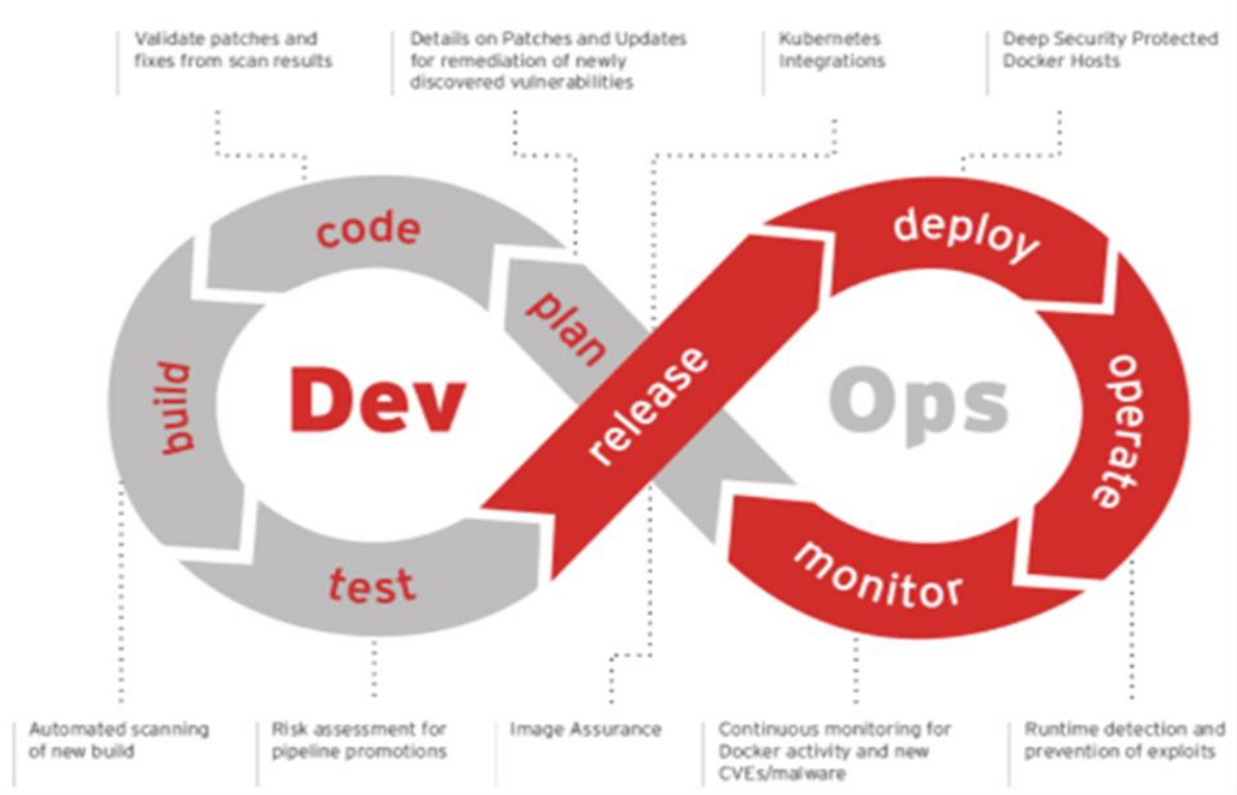

Рис. 1.1. Схема CI/CD

CI/CD – це одна з DevOps-практик. Також вона відноситься до agile-практик: автоматизація розгортання коду дозволяє розробникам зосередитися на реалізації безпеки, якості коду та бізнес-вимогах [2].А також CI/CD є рішенням проблем, які інтеграція нового коду може спричинити для команд розробки та операцій (втім відомі як «інтеграційне пекло»).

А зараз трохи більше про pipeline (рис. 1.2) і його визначення [3]. У контексті галузі розробки під pipeline маємо на увазі “конвеєр”, яким доставляється код. Говорячи про конвеєрну розробку, ми повинні розуміти, що в будь-якій команді розробників конвеєр – це набір автоматизованих процесів, або ж серія кроків, які необхідно виконати, та завдяки яким розробники DevOps можуть дуже ефективно створювати, компілювати та розгортати код на своїх платформах, а також надавати цю нову версію програмного забезпечення іншим розробникам[4].

Не існує правил, що визначають, як правильно повинен виглядати конвеєр розробки програмного забезпечення і який набір інструментів він повинен використовувати. Тим не менш, фактично, в ньому описаний набір методів та автоматизації розгортання додатків у різних середовищах, який дозволяє робити викатки коду частішими, зменшувати кількість помилок, а також в цілому прискорювати роботу різних груп працівників, наприклад: Dev, DevOps, QA.

Рис. 1.2. Модель Pipeline

З огляду на всі вище наведені факти щодо терміну pipeline (конвеєр), можемо зробити невеликий висновок о перевагах DevOps конвеєра:

- автоматизація створення та випуску програмного забезпечення;

- запровадження ефективних рішень, щодо забезпечення якості;
- швидкий життєвий цикл розробки програмного забезпечення;
- налаштування плавної роботи програмного забезпечення.

1.1.1. Визначення CI/CD

Безперервна інтеграція – це набір методологій або набір практик, сенс яких донести до інженера, що код треба вносити невеликими змінами з частими комітами [2]. В наш час, більшість програм створюються з використанням багатьох інструментів та розробляються на різних платформах, тож з'являється необхідність в інтеграції та тестуванні внесених змін.

З технічної точки зору, безперервна інтеграція – це методи які дозволяють робити збірку програм послідовною та автоматизованою. При налагодженому процесові безперервної інтеграції розробник буде робити частіші коміти, що у свою чергу дасть кращу комунікацію між групами та покращить якість створюваного програмного забезпечення.

Безперервна доставка (рис. 1.3) – це фактично продовження після закінчення безперервної інтеграції, її розширення. CD автоматизує розвертання всіх змін коду в різних середовищах розробки (тестове або виробниче середовище) після етапу збірки [5]. Іншими словами, в кінці отримаємо автоматизований процес випуску і зможемо в будь-який момент розгорнути програму, натиснувши на кнопку. При безперервній доставці зможемо випускати версію програмного забезпечення щодня, щотижня і т. д., відповідаючи потребам нашого бізнесу. Однак, якщо дійсно хочемо отримати всі переваги безперервної доставки, то повинні також дуже пильно притримуватися правил безперервної інтеграції, та розпочати інтегрування якомога раніше, щоб процеси плавно переходили один в одного. Тобто, всі ці процеси повинні працювати разом, аби програмний код випускався невеликими партіями і якщо в процесі з'являться якісь помилки, їх було б легко виправити.

Безперервне розгортання – його дуже часто плутають з безперервною доставкою, хоча між ними є декілька чітких відмінностей [6]. Сама головна відмінність, в тому що безперервна доставка забезпечує постійний потік оновлень користувачам і це проходить в ручну, а безперервне розгортання робить так, що весь новий функціонал після тестування відразу буде попадати в основний продукт автоматично на запланований час. Хоча інструменти будуть розглянуті пізніше, але для прикладу: є така технологія як Docker, котрий створений для безперервного розгортання. Розробники можуть обновляти контейнери і розгортати їх відразу в автоматичному режимі. Хоч ідея того, що все автоматично оновлюється після пройдених тестів і звучить казково, але це не завжди має зміст. В тих же банківських сферах він не використовується. Загалом треба завжди дивитися на наші потреби, та на логіку проекту.

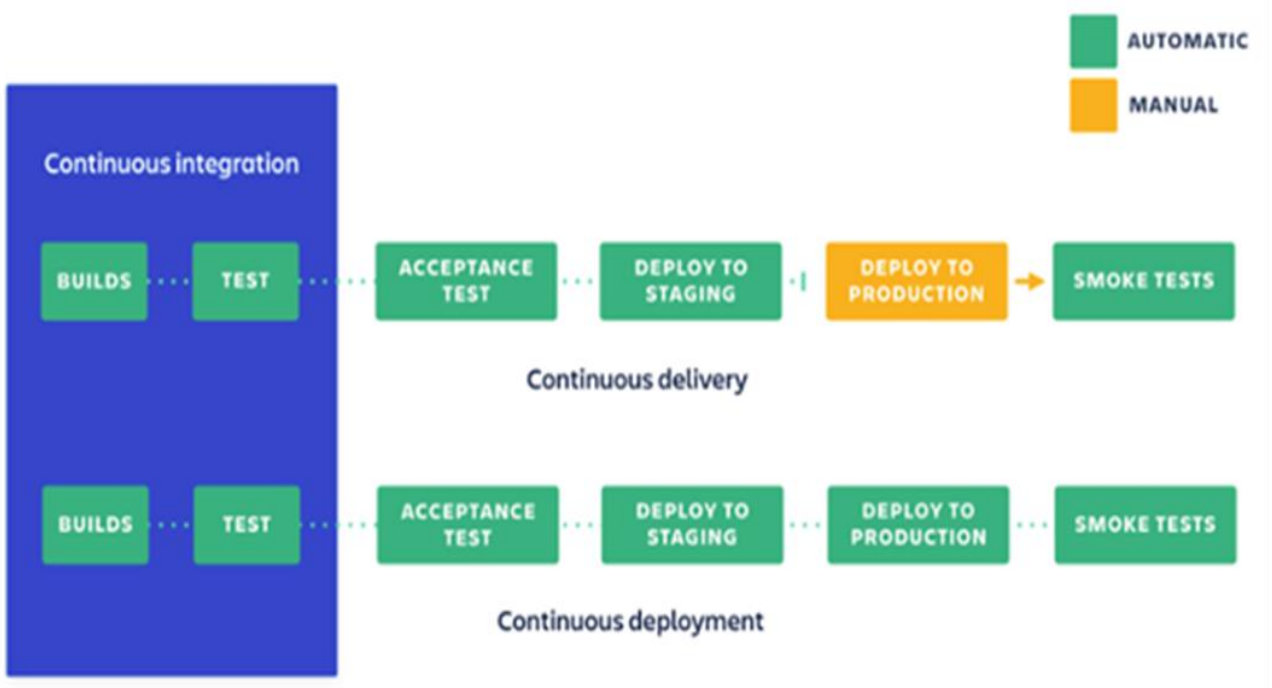

Рис. 1.3. CI/CD (CD)

В цілому, безперервна інтеграція є логічним провадженням як безперервної доставки, так і безперервного розгортання. А безперервне розгортання дуже схоже на безперервну доставку, за виключенням того, що нові релізи оновлень проходять автоматично.

1.1.2. Необхідність та переваги CI/CD

Безперервна інтеграція

Коли потрібно:

- команді потрібно писати автоматичні тести для кожної нової функції, для покращення чи виправлення помилок.
- потрібно спостерігати за основним репозиторієм і запускати тести автоматично для кожного нового коміта.
- розробникам потрібно як частіше вносити нові зміни, хоча б раз у день.

Що отримаємо:

- в кінці буде якомога менше помилок, бо вони фіксуються ще на ранній стадії авто-тестів;
- легко викласти новий реліз, бо всі проблеми з інтеграцією вирішено завчасно;
- розробники більш зосереджені на основних задачах, йде менше перемикання уваги, бо перш ніж перейти до нового завдання, вони максимально дороблюють попередні завдання, бо якщо там є якась помилка, то тести з цим допоможуть;
- час на тестування дуже скорочується, бо сервер CI може виконувати сотні тестів автоматично і дуже швидко.

Безперервна доставка

Коли потрібно:

- якщо є необхідність автоматизувати розгортання;
- якщо потрібен якийсь тригер, що виконується вручну і після якого вже піде автоматичне розгортання на автоматичному рівні;
- якщо вже є міцна основа безперервної інтеграції (ваш набір тестів охоплює значну частину вашого коду), після якої можна вдало використовувати безперервну доставку.

Що отримаємо:

- у команди буде набагато більше часу, так як вона звільняється від підготовки продукту до випуску. Складності з розгортанням продукту знімаються;
- можемо випускати оновлення частіше, що покращує зв’язок з клієнтом.

Безперервне розгортання

Коли потрібно:

- якщо команда хоче реалізувати процес тригеру в безперервній поставці автоматичним.
- Що отримаємо:
- бізнес буде розвиватися швидше, оскільки розробка не зупиняється для випуску;
- клієнти бачать безперервний потік покращень, а якість підвищується щодня (постійно).

Отже, якщо підвести підсумки. CI пакує код, робить тести та повідомляє розробникам про помилки. CD автоматизує доставку та розгортання продукту, та якщо потрібно робить додаткові тести.

CI/CD pipeline (конвеєри) потрібні, якщо потрібно часто вносити зміни, вкрай потрібна стабільність. Впровадження CI/CD дозволяє розробникам зосередитись на покращенні продукту, а не на його розгортанні. CI/CD як вже вияснилося є однією з DevOps практик, оскільки спрямована на боротьбу з протиріччям між командами розробників та експлуататорів. І один із самих головних пункті, про який хотілося обговорити детальніше трохи пізніше, це які інструменти команда розробників буде використовувати в своїх pipeline.

1.2. Історія розвитку культури DevOps

Тепер переходимо до теми хто такий DevOps, або що це, та як цей термін пов’язаний з CI/CD.

Хочеться почати з того, що кожна компанія, пов’язана з розробкою програмного забезпечення, хоче випускати оновлення швидше та бути як можна мобільнішою. Для цього потрібно, щоб розробники максимально залучалися в усі стадії розробки ПО. Стадій доволі багато і часто вони є неоднозначними. Виникає дуже багато питань, як все це спланувати, які процеси потрібно використовувати , а які ні, чи на якій платформі проект розташовувати, наскільки швидко повинні виходити оновлення. Давайте заглянемо у минуле, та розберемо як все це проходило раніше.

Коли все тільки починалося кожен розробник був сам собі бізнес-аналітик, архітектор, тестувальник тощо. В цьому було багато недоліків. Люди часто не розуміли один одного. Кожен робив індивідуально і якщо він не міг знайти помилку у себе, то звалював її на інших. З часом стало так, що все перейшло до масового розподілу. Люди розділилися на більш спеціалізовані групи, але та проблема яку ми розглянули вище, нікуди не зникла. Крім цієї проблеми з’явилася ще одна – аутсорс став домінувати. Це значить що код стали доставляти, як сировину, не замислювалися над кінцевим результатом, та про те, як і де він буде розташовуватися. Мали б змогу спостерігати за цим ще довго, як би не декілька наступних факторів.

Першим фактором стала поява низки продуктових контор, в яких замислюються не тільки над локальним вирішенням проблеми, а й над глобальним. Тут не пройде аби яке вирішення – та потім інші будуть розбиратися. Треба усвідомлювати, що далі доведеться жити з цим “швидким рішенням”, а тому треба вирішувати щось на рівні інфраструктури. Доводиться починати розробляти спираючись на те, де розміщуватиметься кінцевий продукт.

Якщо перший фактор звучить доволі спірно, до другий більш однозначний. Це широкий розвиток хмарних сервісів, відмова від хостингу на своїх серверах та підтримка своєї інфраструктури. Вибрана інфраструктура почала визначати, яка буде архітектура у програми. GCP, AWS, Azure, Digital Ocean, Heroku почали робити за нас нашу роботу. Тепер не треба вигадувати

варіанти написання балансеру або шардингу – це все доступно з коробки. Це знизило кількість витрачених зусиль та часів на це, але цей підхід вимагає знань інфраструктури сервісів та адаптації своїх продуктів під них. Крім того, мікросервісна архітектура внесла свій внесок у переосмислення розробниками інфраструктури програм. Тепер недостатньо “намазюкати” черговий модуль і запушити його в репозиторій, надавши деплоймент-інженерам вгадувати змінні конфігурування. Зараз треба думати, як це все взаємодіятиме один з одним, тому що ця конфігурація може не спрацювати, та відправить неперевірений код, а це вже погано.

Платформи почали визначати реалізацію додатків, тому розробник неспроможний написати хороший додаток без знань про платформи. Спеціалістів розробки почали залучати до операційної роботи. І тут з’явилася культура DevOps.

DevOps – це скорочення від Development Operations і це не назва професії. Ця культура виникла приблизно в 2008 і була покликана вирішувати проблеми, що накопилися в IT сфері.

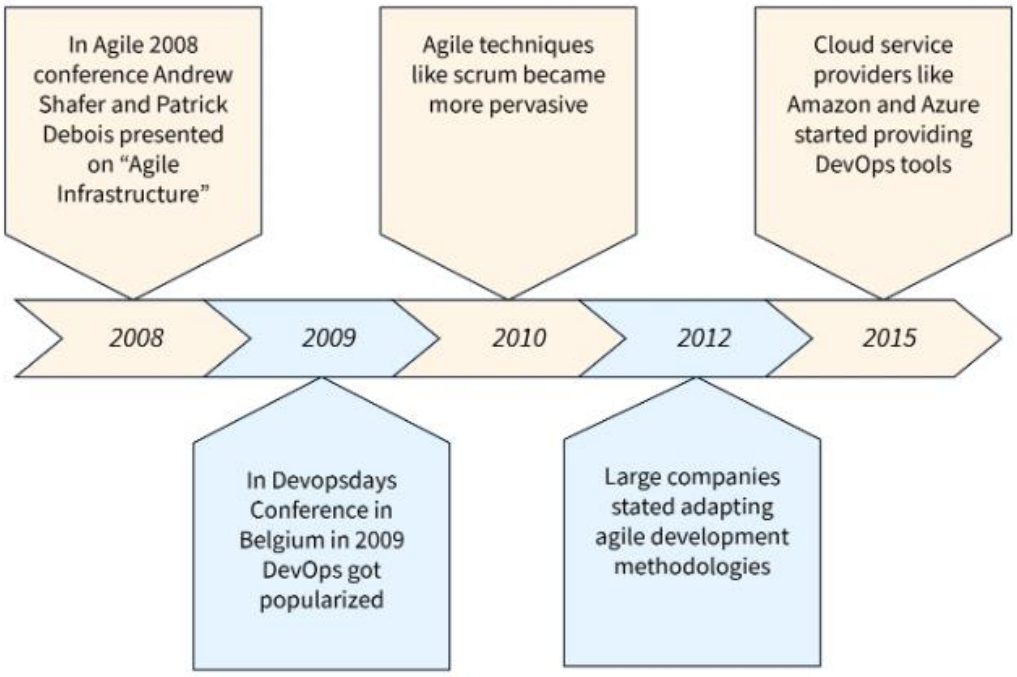

Рис. 1.4. History of DevOps

Може виникнути питання, чому культура, чому не роль? Тому що DevOps-практики, про які йтиметься нижче, повинні впроваджуватися на рівні компанії, а не на рівні групи або відділу. Люди в компанії повинні знати і про розробку програмного забезпечення, і про тонкощі використання інфраструктури. Дуже багато компаній бачили проблеми в відношеннях команд розробки та експлуатації. Розробники вважали, що якщо код запустився у них локально, то немає ніяких проблем – можна запускати у фінальний етап викатки продукту. Якщо проблеми все ж таки виникли, то з боку групи експлуатації звучало б: “Це проблема з кодом, нехай розробники з цим пораються”. Через такий підхід викатка релізів продуктів постійно затягувалася і часто страждала на якість. Сильним відбитком було й те, що за один реліз котилося дуже багато змін і було важко розібратися, що породило проблеми на продакшені. DevOps покликаний вирішувати ці проблеми, він повинен був стати об’єднанням між командою розробки та експлуатації [7].

Інженер DevOps представляє процеси, інструменти та методології, щоб збалансувати потреби протягом життєвого циклу розробки програмного забезпечення, від кодування та розгортання до обслуговування та оновлень.

У гнучкому середовищі розробники, системні адміністратори та програмісти можуть бути розділені, працюючи над одним продуктом, але не обмінюючись інформацією, необхідною для забезпечення цінності для користувача.

Деякі організації можуть наймати професіоналів для «виконання DevOps» у своїх робочих процесах, але оскільки успішне впровадження DevOps залежить від змін у культурі та процесах, це може лише поглибити розрив між розробниками та операційними командами.

Інженери DevOps створюють нові навички на основі свого поточного досвіду. Такі завдання, як керування даними та оновлення бібліотеки для нових випусків продуктів, поєднуються з потребами в лідерстві та співпраці між командами. Для інженерів DevOps важливо розуміти основи розробки та доставки програм[8].

1.3. Ролі та практики DevOps

У DevOps культурі можна виділити кілька ролей, які дуже добре співвідносяться з професіями:

- Build Engineer – це людина яка відповідає за збір коду. Підтягування залежностей, розбір конфліктів у коді, це все про нього;
- Release Engineer – відповідає за доставку коду розробників у реліз. Він вирішує такі питання: яка гілка коду піде на тестування, який білд продукту піде у реліз тощо;
- Automation Engineer – це інженер по автоматизації. Він автоматизує майже все. Автоматичний збір при розгортанні коду в гіт, прогін тестів, доставка на staging, розгортання в реліз – це його завдання. Головна роль в DevOps;
- Security Engineer – відповідає за прогін security-тестів та визначення вразливостей в використовуваних компонентах.

Зараз всі (або майже всі) ці ролі зазвичай окремо поєднує якась інша людина. Наприклад, роль build engineer можна віддати до рук розробника коду. Автоматизацію налаштування серверів зазвичай віддається системний адміністраторам. А DevOps інженерові залишається пропрацювати та автоматизувати процес складання та доставки коду від розробника в реліз [9].

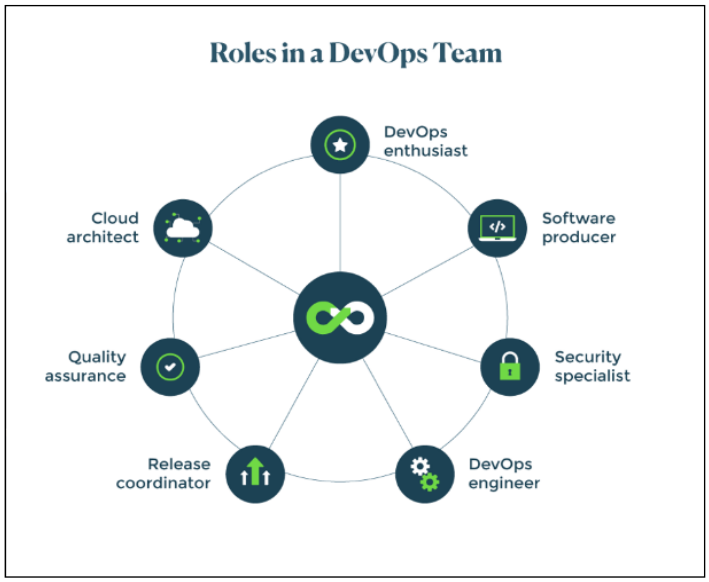

Рис. 1.5. Ролі в DevOps

Тепер розглянемо головні практики DevOps:

“Automate everything” – автоматизувати все, що можна. Зменшувати кількість ручної праці, якщо щось робиш два рази – придумай як це автоматизувати. Це прискорює всі процеси та зведе кількість помилок до мінімуму;

“Configuration management”. Docker допомагає в конфігурації, збереженні та менеджменті усього, що потрібно для вдалої праці. Оркестрацію контейнерів можна здійснювати за допомогою таких інструментів, як Kubernetes та Docker Swarm;

“Infrastructure as Code”. Мається на увазі, що підхід до конфігурування програм повинен бути таким самим, як і до коду. Якщо раніше в конфігурації системи через консоль не було нічого ганебного, то сьогодні стало вже поганим тоном не використовувати для цих цілей інструменти автоматизації, такі як Terraform, Chef, Puppet тощо. Ця практика дозволяє оптимізувати ресурси, а також значно прискорити час поставки;

“Continuous Deployment”. Автоматична викатка готових оновлень на робоче оточення. Ця практика також дозволяє оптимізувати ресурси, та зменшує участь людини у процесі постачання до мінімуму;

“Application monitoring”. Якщо раніше системи моніторингу являли собою різні способи “скидання” логів, то тепер це потужний інструмент для моніторингу стану програмного забезпечення. На аналіз метрик не треба витрачати дня та тижні, ви можете налаштуватись на ту чи іншу метрику та стежити за змінами в режимі реального часу. Мало того, крім суто технічних моментів, наприклад кількості запитів, завантаженні CPU, є можливість при допомозі таких систем, як Prometheus, збирати і внутрішні характеристики програми, важливі для бізнесу.

Розглянуто далеко не всі практики, які є в культурі DevOps. Однак, якщо звернути увагу, вони являють собою набір методів та інструментів, при використанні яких можна вирішити проблему швидкого та якісного постачання

ПЗ кінцевому споживачеві, а також якнайшвидше отримати від нього зворотній зв’язок у вигляді метрик [7]. Наведено фото метрик (рис. 1.6):

Рис. 1.6. Приклад метрик

1.4. Висновки до першого розділу

У першому розділі, було проведено аналіз понятійного апарату інтеграції та розгортання коду. Наведено різницю таких підходів як continuous integration/ continuous delivery/ continuous deployment та при яких умовах їх треба використовувати, також були перераховані переваги цих підходів.

Проаналізовано вплив DevOps методологій на сферу IT. Визначено, що DevOps практики впроваджуються на рівні компанії, а не на равні групи або відділу. DevOps вирішує проблеми між компаніями розробки та експлуатації. Впроваджується баланс потреб протягом життєвого циклу розробки програмного забезпечення, від кодування та розгортання до обслуговування та оновлень.

Розглянуто практики DevOps. Вони являють собою набір методів та інструментів, при використанні яких можна вирішити проблему швидкого та якісного постачання ПЗ кінцевому споживачеві, а також якнайшвидше отримати від нього зворотній зв’язок.

РОЗДІЛ 2
ДОСЛІДЖЕННЯ ПІДХОДІВ ТА ІНСТРУМЕНТІВ РОЗГОРТАННЯ КОДУ

2.1. Інструменти DevOps-розробки. Види CI/CD практик

2.1.1. Види інструментів, стратегій, підходів в Continuous integration/delivery

Як було зазначено вище, DevOps розробник та CI/CD процеси це одне ціле, а однією з головних задач DevOps інженера є вбудовування практик CI/CD в робочий процес бізнесу. Розглянемо інструменти, якими користується розробник, щоб побудувати ці CI/CD процеси. Зауважимо, що інструменти DevOps використовуються на всіх основних етапах життєвого циклу ПЗ. Вони розширюють можливості методик DevOps, сприяючи ефективній спільній роботі, дають можливість краще зосереджуватися на одній задачі, впроваджують автоматизацію та створюють можливості для спостереження та моніторингу.

При виборі пакету інструментів DevOps, організаціям необхідно знайти такі, які сприяють ефективній спільній роботі, дозволяють зменшити кількість перемикань контексту, автоматизувати процеси та використовувати можливості моніторингу та спостереження для прискореного постачання якісного програмного забезпечення.

Усі пакети інструментів DevOps поділяються на два типи: універсальні та відкриті. Універсальний пакет є комплексним рішенням; зазвичай його не можна інтегрувати зі сторонніми інструментами, тоді як відкритий пакет можна налаштовувати та доповнювати різними інструментами. У кожного з цих підходів є свої плюси та мінуси [9].

Незалежно обраного типу пакета, процес DevOps вимагатиме відповідних інструментів на ключових етапах циклу DevOps:

- системи керування версіями;
- інструменти безперервної інтеграції;

- quality gateway;
- планування;
- збірка;
- моніторинг;
- безперервний зворотний зв’язок;
- хмарні платформи;

Буде наведений перелік найпопулярніші інструменти. Через специфіку ринку список може скоро змінитися. Постачальники додають до інструментів нові можливості, які дозволяють працювати з додатковими етапами циклу DevOps.

1. Системи керування версіями

Хочеться почати саме з цього, з місця куди заливають весь код, звідки і починаються перші шаги автоматизації, та безперервної інтеграції. Тут буде розглянуто не тільки інструменти, які використовуються на даному етапі, а й стратегії, що допомагають більш якісно організувати роботу. Що ж, мати контроль над вихідним кодом дуже важливо. Інструменти керування вихідним кодом дозволяють зберігати код у різних ланцюжках. Так можна бачити всі зміни і легко працювати над ними всією командою. Замість тривалих зборів щодо підтвердження змін перед розгортанням у робочому середовищі, проводити перевірки у формі колег за допомогою запитів pull. За допомогою запиту pull повідомляється команда про зміни, внесені у гілку розробки в репозиторії. Перш ніж інтегрувати запропоновані зміни в головну гілку коду, команда зможе їх переглянути та обговорити. Застосування запитів pull підвищує якість програмного забезпечення, що призводить до зменшення кількості багів та інцидентів. В підсумку знижуються витрати на експлуатацію і прискорюється розробка. Самий популярний інструмент системи керування версіями це Git, в якому зберігаємо код. Хоч він і самий популярний, але він не єдиний (SVN, Mercurial). У них всіх свої особливості реалізації, Git вже закріпився, як самий популярний, з огляду на те що в ньому простіше всього створювати гілки та з’єднувати їх між собою. Більш детально про їх відмінності трохи далі.

Не менш важливими є branching strategy. Яка їхня мета? Це спосіб організації роботи розробників. Коли над проектом працює не одна людина і доволі тривалий час, то нерідко накопичується сміття (непрацюючий код або не видалені гілки), сам концепт CI веде нас до того, що всі гілки з часом з'єднаються в одну, і якщо не організовувати цей процес злиття, можуть статися інциденти. Також дуже важливим є feature гілки, які ми створюємо, щоб не заважати іншим. Перше що йде до голови це GitFlow (рис. 2.1), він був розроблений в 2010 році Вінсентом Дрісеном [10].

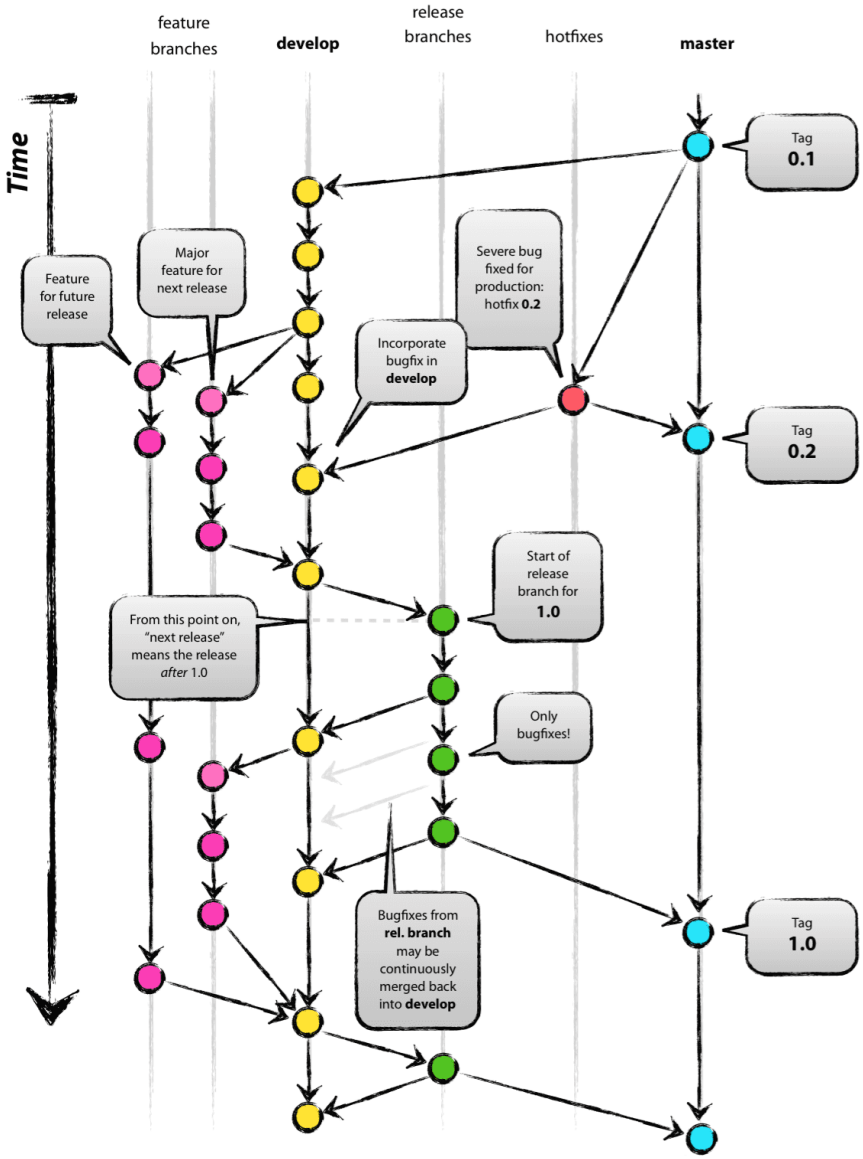

Рис. 2.1. GitFlow

Зауважимо, що ця стратегія була розроблена 13 років тому і є вже досить не новою. Сам автор казав, що він проектував її для продуктів, які будуть доставлятися у вигляді готового пакету, зараз це не сама оптимальна стратегія,

але все ще використовується. Вона доволі складна. Є нюанси в гілках, цих гілок купа і про них забувають. Надмірна стратегія, але доволі продумана. Сенс її в, тому що створюється декілька головних гілок (develop, release, maste, як показано на рис. 2.2), назви та кількість можуть варіюватися. Розробник робить свою feature гілку, в якій він працює над своє частиною, робить pull request (це запрос, щоб на його роботу подивилися, та дозволили їй зливатися з основним кодом), його код заливається в develop гілку, в вже потім в release та master.

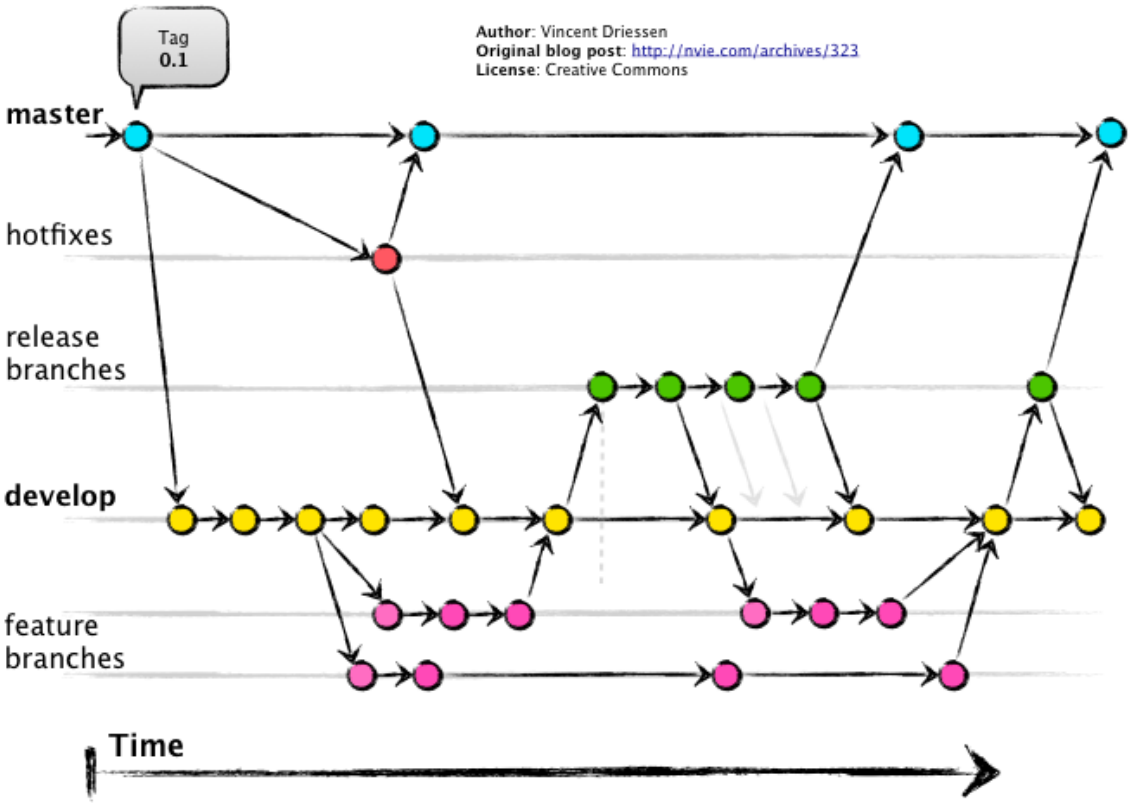

Рис. 2.2. Гілки GitFlow

Ось ще приклад пов’язаний з вичурністю гілок. В гілці release можуть з’являтися окремі гілки hotfix. Коли розробник вже має робочу версію продукту, але зустрічає якусь проблема, йому треба створити нову гілку hotfix, в якій він виправляє виявлений недолік, вже потім треба не забути поєднати цей код з гілкою develop, release та master. Дивлячись на це, розуміємо, що тут існує проблема в кількості з’єднань між гілками. Таким чином може статися, що із-за великої кількості гілок, про деякі забувають. Вони можуть жити в проекті дуже довго та коли вони помирають, вони вже не приносять користі. Тож GitFlow на даний час це не самий правильний варіант стратегій!

Наступний підхід до реалізації роботи в системах контроля версій є GitHub Flow (рис. 2.3) і це вже набагато популярніший підхід. Є думка, що цьому сприяла його назва, а саме GitHub, та менша кількість гілок [11]. Як це працює? Є основна гілка master, яка постійно знаходиться в робочому стані, тому що вона напряму пов’язана з готовим продуктом, а це значить, що немає можливості робити в ній помилки. А якщо там і з’являються якісь помилки або не співпадіння, то їх треба максимально швидко виправляти, створюючи нові гілки: bugfix, feature, тощо. Такий підхід може не всім підходити, банально не у всіх може бути можливість тримати головну гілку весь час в стабільності, в тому ж GitFlow є можливість зробити відкат і працювати в інших гілках не хвилюючись за основну.

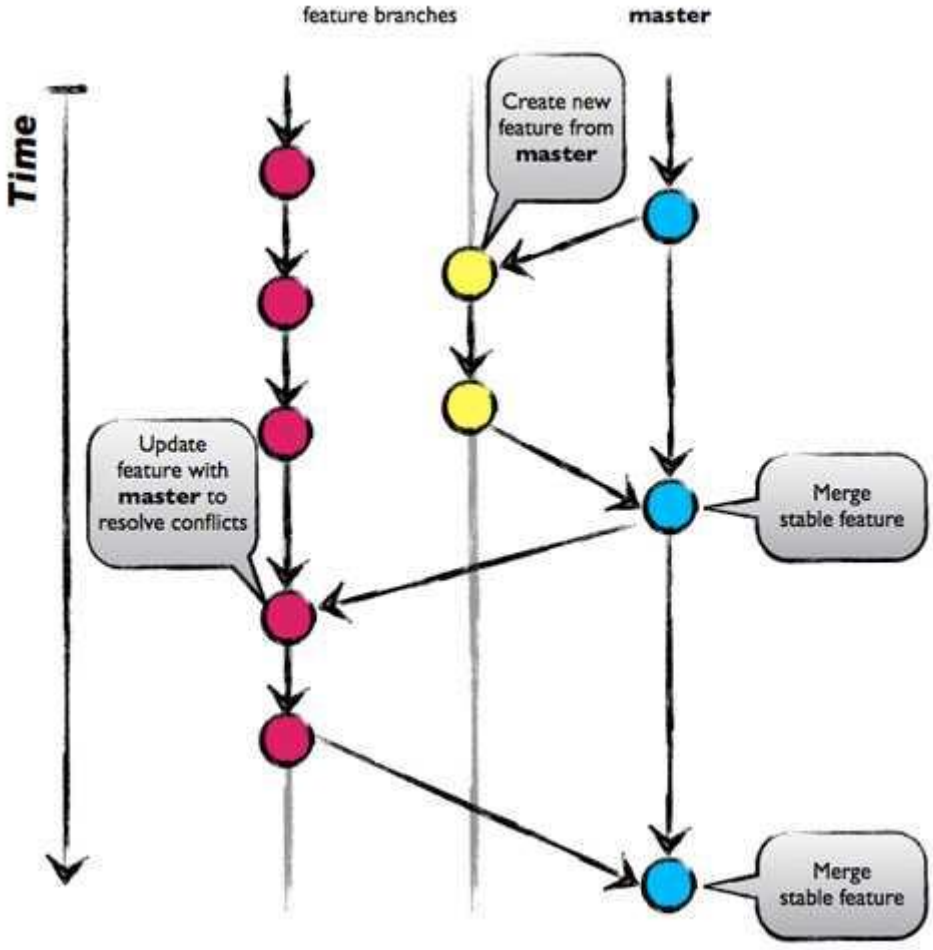

Рис. 2.3. GitHub Flow

І остання branching strategy, яка буде розглянута це Trunk-Based (рис. 2.4). Останім часом Trunk-Based все більше та більше набуває популярності. Тут схожий принцип з GitHub Flow де є одна постійно максимально стабільна гілка,

в яку заливаються оновлення або виправлені помилки. Різниця в тому, що тут розробники частіше вносять невеликі зміни, обмежуючи довгострокові гілки та уникаючи конфліктів злиття, оскільки всі розробники працюють над однією гілкою. Ця стратегія більше всього за своєю суттю підходе під концепт CI (Continuous Integration), тому що сенс CI, це інтеграція коду між гілками розробника і основною є дуже частими, можливо навіть декілька разів на день.

Така стратегія часто поєднується з прапорцями функцій. Оскільки trunk завжди готовий до випуску, позначки функцій допомагають відокремити розгортання від випуску, тож будь-які неготові зміни можна загорнути у прапор функції та залишити прихованими, тоді як готові функції можна без затримки надати кінцевим користувачам. Наприклад, була створена функція ще місяць назад, йде очікування якогось свята, типу чорної п’ятниці або іншої події і в цей день просто міняєте прапорець, роблячи цю функцію активною на проекті.

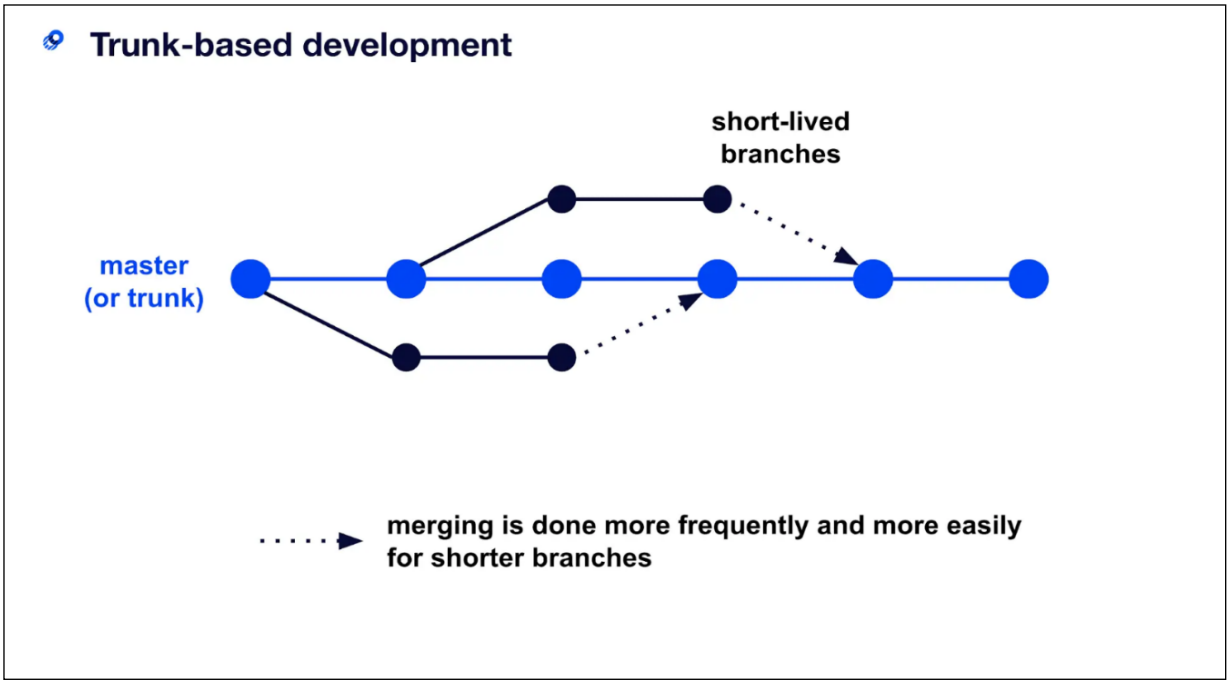

Рис. 2.4. Trunk-Based

По стратегіям це все, їх є ще декілька але цих вдосталь для загального розуміння теми гілкових стратегій. Існує необхідність зачепити таке поняття, як Code Review. Воно вже було згадано, це pull request, якими користуються, коли просять переглянути код для перевірки перед тим як заливати його до основної гілки. Підхід Code Review дуже важливий, він допомагає розвиватися молодшим

співробітникам. Коли новачки приєднуються до команди, їх код буде меншої якості ніж у більш досвідчений фахівців і Code Review це можливість навчатися, показуючи свій код іншій людині, вона може підказати або порекомендувати, як можна його покращити. Але є і мінус, Code Review може стати таким собі bottleneck (пляшкова шийка), адже щоб код увійшов до основної гілки, треба чекати підтвердження (approve). Це підтвердження може займати багато часу. В великих компаніях його можна очікувати аж до 1-2 днів, або навіть більше. Таким чином затягується життя гілки, порушуючи принципи безперервної інтеграції. Також approve не завжди може бути обґрунтованим, є ситуації, коли людина не до кінця занурюється у задачу або їй просто бракує на це часу. Все одно Code Review неможливо повністю замінити тестами, тож на даний час це незамінна практика.

Наступна тема, яка стосується систем контролю версій це – семантичне версіонування[12]. Існує таке поняття, як “dependency hell” (пекло), воно з’явилося із зростанням систем та інтеграцій в них великої кількості пакетів. Dependency hell – це ситуація, коли несумісність версій або їх блокування заважає легко і безпечно просувати проект вперед. У системах з багатьма залежностями випуск нових версій пакетів може швидко перетворитись на жах. Щоб запобігти цьому, в якості вирішення проблеми пропонується простий набір правил і вимог, які визначають те, як призначаються та збільшуються номери версій. Ці правила ґрунтуються (але цим не обмежуються) на існуючих поширених практиках, що використовуються як в закритому, так і у відкритому програмному забезпеченні. В середньому версії поділяються на 3 типи мажорна, мінорна, патч:

- мажорна версія, це коли ми робимо кардинальні зміни, несумісні API зміни. Яскравим прикладом є перехід Python з версії 2 на 3 (те що працювало в версії 2 не завжди працювали в 3, зараз 2 версією Python ніхто не користується). Тому ніколи не забувайте закріплювати версії ваших моделей в Terraform, або ставити останні версії в docker images,

бо коли мажорна версія зміниться, скоріше за все вони вже не будуть працювати!

- мінорна версія: якщо додана нова функціональність, то вона буде сумісною з попередньою версією.
- патч версія: якщо були зроблені виправлення багів та помилок, що не вплинуть на сумісність з попередньою версією.

Також після патчу є можливість добавляти номер білду або git hash того коміту з якого була зроблена збірка даного пакету. Це потрібно для того, щоб мати розуміння в якому місці та як швидко можна знайти помилку, якщо вона буде.

2. Інструменти безперервної інтеграції

Як вже було розглянуто, безперервна інтеграція - це методика завантаження коду в загальний репозиторій кілька разів на день із подальшою перевіркою. Вона дозволяє автоматично виявляти проблеми на ранніх етапах, коли їх найпростіше усунути і розгортати нові можливості для користувачів настільки швидко, наскільки це можливо.

Краще обирати ті інструменти, які автоматично тестують гілки розробки та дозволяють відправляти їх у головну гілку за відсутності помилок у зборці (рис. 2.5). Крім того, завдяки простій інтеграції можна отримувати безперервний зворотний зв’язок у вигляді оповіщень у чаті команди в режимі реального часу.

Рис. 2.5. Інструменти безперервної інтеграції

3. Quality gateway

Наступний підхід, який використовується в розглянутому CI/CD це quality gateway. В часи сучасності можна спостерігати, що в процесі розробки командам треба бути все більш гнучкими, тому що вимоги постійно змінюються.

І в таких ситуаціях дуже легко не помітити, як процес розробки та сам продукт перестають відповідати необхідним стандартам якості. Quality gateway - це заздалегідь визначені етапи, під час яких проект перевіряється на відповідність необхідним критеріям переходу до наступного етапу [13].

Мета – забезпечити дотримання набору певних правил та передових практик, щоб запобігти ризикам та збільшити шанси на успіх проекту. За допомогою якісних Quality Gates організації можуть гарантувати, що керівники проектів виконують свою роботу та не пропускають жодних важливих кроків.

Розглянемо на звичайних кроках CI, які інструменти та підходи quality gateway можна використати. Перший крок – Code Analysis, те що робимо на самому початку. Тут ми маємо можливість використати SonarQube (рис. 2.6), який аналізує код та пропонує такі варіанти як: кількість code smells на проекті (термін, що позначає код з ознаками проблем в системі), або туж кількість багів.

Рис. 2.6. Утиліта Sonarqube

У Sonarqube та схожих з ним інструментів є свої правила, на базі яких вони аналізують код. Наприклад, створюється функція з якоюсь логікою, але ця функція ніде не визиваєте, можливо в майбутньому вона десь буде використана, але на даний момент це буде не саме оптимальне рішення зберігати її у коді. Або випадково залитий до програми пароль, який був належним чином не збережений. Code Analysis допоможе виявити такі уразливі місця, та може запропонувати кращий варіант. Хоча цей етап Code Analysis може багато у кого не бути, зазвичай спочатку реалізують наступні 3 кроки, які будуть обговорені пізніше (Build, Unit Test, Artifacts), а вже потім добавляють початковий Code

Analysis. Тож це не 100% забов’язаність, щоб він був на проекті, а лише гарна практика, яка не завадить. Усе на покладається на розсуд та можливості.

Другий крок – Build. Це компіляція та збірка проекту, щоб впевнитися, чи все гарно працює. Після цього йде 3 етап – Unit tests. Модульне тестування пишуть розробник для кожного модуля коду програми. Модулем є найменша частина програми, яка може бути протестованою, наприклад функції. Це доволі прості тести, вони не повинні вимагати якоїсь складної інтеграції. Навіть є твердження, що unit tests не повинні нічого писати на файлову систему. Тобто, якщо тесту треба з’єднання з базою даних або з іншим сервісом, то це вже не є unit test, а інтеграційний тест. Unit tests - це незалежні тести. Для останнього етапу CI виділяється окремий підпункт.

4. Планування, збірка

Після того як код був вдало зібраний, проведений аналіз, проведено модульні тести, то наступним етапом йде створення та зберігання артефакту. Починається той час, коли треба зібраний артефакт відправляти в систему зберігання артефактів (Artifactory, Nexus) (рис. 2.7). Але зараз найчастіше все добавляється в docker образ і відправляється в docker registry, та потім використовується в kubernetes (інструменти збірки).

Рис. 2.7. Artifactory, Nexus

До речі, якщо серед популярних інструментів управління експлуатацією або ще кажуть, інструменти управління конфігурацією. Керування конфігурацією – це коли йде ідентифікація компонентів системи, визначення функціональних та фізичних характеристик системи. Яскравим прикладом таких рішень будуть інструменти Puppet та Chef, для створення ж окремих середовищ розробки використовуються інструменти з відкритим вихідним кодом, а саме Kubernetes та Docker (рис. 2.8).

Рис. 2.8. Інструменти збірки

Тільки но було розібрано 4 умовних кроки CI (безперервної інтеграції), як настав час поступово переходити до CD (безперервна доставка). CD базується на тому, що отримується від CI, тобто на попередньо отриманому артефактові. На базі цього артефакту продовжується тестування продукту та встановлення його на інші тестові середовища. Це може бути як develop, QA, UAT, staging середовища, на яких можуть бути різного рівня перевірки. Інколи ці середовища поєднані воєдино, бо не у всіх проектів є ресурси та можливість підтримувати таку їх кількість, але для того щоб поділити команду та не заважати друг другові, краще мати не один і не два. Гарним тоном вважається, коли environments визначені для окремих задач, а не робиться все в одному.

На етапі доставляння артефакту на інші середовища, в гру вступає планування інфраструктури (Infrastructure as code) (рис. 2.9). Цей підхід дозволяє створити та розгорнути інфраструктуру, зберігаючи її у вигляді коду. Код можна зберігати в системі контролю версій, тестувати, вбудовувати в процес безперервної інтеграції та оцінювати код колег. Коли накопичені знання зафіксовані у коді, зникає потреба у переліках процедур та внутрішньої документації. Процеси стають відтворюваними, а системи – надійними.

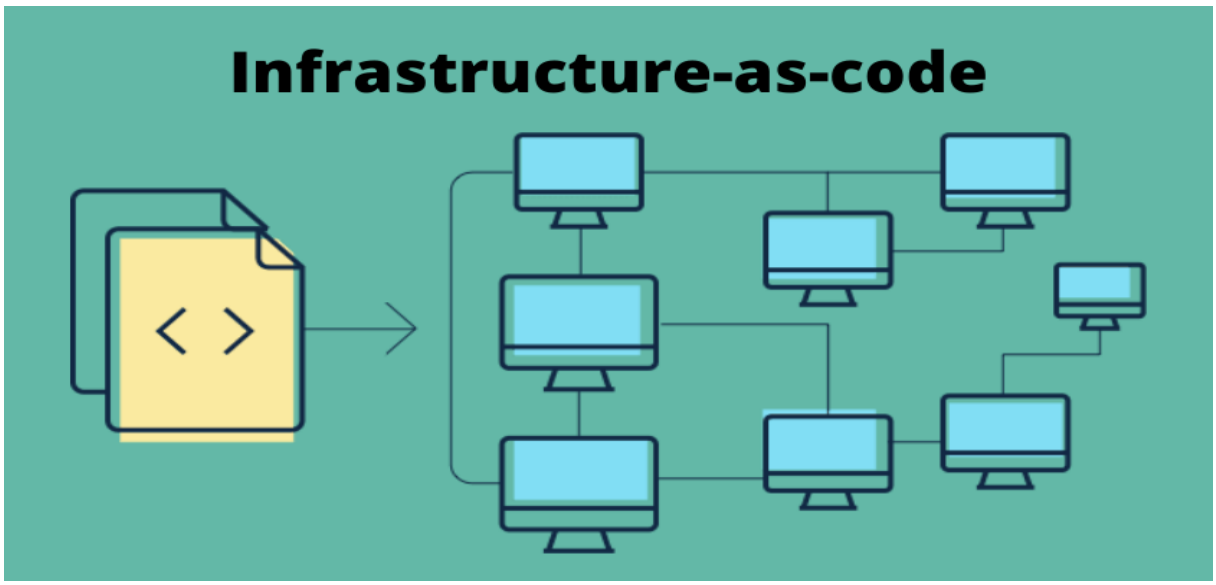

Рис. 2.9. Підхід “Інфраструктура як код”

Приклад декількох інструментів (рис. 2.10):

Рис. 2.10. Декілька інструментів “Інфраструктура як код”

Зауважимо, що саме на цьому етапі йде перехід з CI в CD. CD в цілому базується на переходах з одного середовища до іншого.

Перевагами continuous delivery є швидкість доставки нових функціональностей. Раніше починаємо помічати помилки, баги. Неодмінною перевагою є гнучкість, яку ми отримуємо. Дуже добре підхід безперервної доставки вливає на моральний стан команди, яка працює над проектом в якому використовується цей підхід. Мається на увазі, що команда стає більш вмотивованою за рахунок постійних результатів, які вона може бачити при розгортанні артефакту на середовищах, вже на ранніх середовищах можна

спостерігати роботу свого продукту. Чим більше ми робимо доставку, тим менше ми стресу отримуємо, бо процеси стануть більш “загостреними”.

Може скластися питання, чому коли вже існує артефакт і треба кудись його доставляти, тільки тоді починається створюватися інфраструктура. Так і є, у великої кількості проектів інфраструктура створюється раніше, але проте зв’язок між додатком та інфраструктурою повинен бути, бо є залежності або бібліотеки, які треба встановити. Звичайно, ті ж контейнери це невілюють, бо вони є ізольованим середовищем, але досі існують проекти в яких не все запускається в контейнерах і це є той момент, коли потрібно привести інфраструктуру в той стан, який потребує продукт. Як раз на цій стадії і є самий вдалий момент, коли можна запустити код інфраструктури та створити її.

На цьому кроці досі використовується принцип Quality gateway! Як зрозуміти, чи можна відправляти артефакт на інше середовище? Наприклад, коли вперше передається артефакт на develop середовище, запускаються тести, які перевіряють працездатність, запускається Quality gateway і припустимо, що тут артефакт отримує помилку або ще щось. Таким чином просто не добавляється тег до його версії, тим самим вказуючи, що далі на інші середовища він пройти не може. Зазвичай, коли процес доходе до фінального етапу, з’являється тег ready for production (готовий до випуску), який буде вказувати готову до випуску версію додатку. На кожному етапі Quality gateway йде перевірка, чи готовий артефакт йти далі, чи ні. Тож, чим більше Quality gateway в проекті, тим більше вірогідність, що помилка знайдеться на ранньому етапі.

5. Моніторинг

Вже на цьому етапі треба починати використовувати моніторинг. Усі нижчестоящі середовища повинні перевірятися. Діло в тому, що якщо develop або інша середа впаде, то це вплине на призупинення проекту. Тому моніторинг повинен бути налаштований з нижчих середовищ, а не перед самим випуском, щоб запобігти простою в процесі і виявляти помилки раніше.

Трохи більше про саме поняття моніторингу - Є два типи нагляду за проектом, які вимагають автоматизації: моніторинг серверів та моніторинг продуктивності додатків.

Доробка вручну та тестування API – відмінний варіант для вибіркової перевірки. Однак для розуміння тенденцій та загальної працездатності додатків та середовищ необхідне ПЗ, яке відстежує та зберігає дані цілодобово та без вихідних. Можливість безперервного спостереження є ключем до успіху команд DevOps.

Треба знайти інструменти (рис. 2.11), які будуть інтегруватися з клієнтом групового чату, щоб оповіщення доставлялися прямо в кімнату команди або окрему кімнату для інцидентів.

Рис. 2.11. Інструменти моніторингу

Слідкування за інцидентами, зміненням та помилками

Для розвитку взаємодії між командами DevOps необхідно розуміти, що усі співробітники повинні мати єдине та цілісне уявлення про поточну роботу. Що відбувається у разі виникнення інцидентів? Чи виконується їх прив'язка до проблем програмного забезпечення та чи відстежуються вони? Чи виконується прив'язка нових змін до релізів?

Найбільші перешкоди взаємодії між розробниками та спеціалістами з експлуатації виникають, коли відстеження інцидентів та проекти з розробки ведуться над єдиною, а не на різних системах. Обирайте інструменти, які зберігають інциденти, зміни, проблеми та програмні проекти на одній платформі, щоб виявлення та усунення проблем було оперативним.

6. Безперервний зворотній зв'язок

Далі про безперервний зворотний зв’язок. Безперервний зворотний зв’язок (рис. 2.12) – це культура та процеси регулярного отримання зворотного зв’язку, а також інструменти вилучення з нього аналітичних даних. Методики безперервного зворотного зв’язку включають збирання та аналіз даних NPS, опитувань про причини відтоку клієнтів, звітів про баги, заявок у службу підтримки і навіть твітів. Відповідно до принципів культури DevOps кожен учасник команди продукту повинен мати доступ до коментарів користувачів, оскільки вони допомагають керувати всім процесом: від планування релізів до сеансів глибокого тестування.

Спочатку може здатися, що аналіз зворотного зв’язку та коригування на його основні уповільнюють темпи розробки, однак у довгостроковій перспективі це ефективніше, ніж випуск нових, але нікому не потрібних можливостей [14].

Рис. 2.12. Інструменти безперервного зворотного зв’язку

7. Хмарні платформи

Треба також згадати і про хмарні платформи, без яких підхід Infrastructure as code рідко де використовується. На даний час методологія DevOps неможлива без використання хмарних платформ (рис. 2.13), бо в такому випадку вона не зможе забезпечити необхідну ефективність CI/CD-процесу.

Хмарні обчислення можна охарактеризувати як систему, яка забезпечує та підтримує надання інфраструктури. Таким чином, можна описати її як код або шаблон, який полегшує створення повторюваних процесів. Це також основний принцип DevOps, що максимально автоматизує процеси/завдання в життєвому циклі розробки програмного забезпечення.

Однією з найбільших переваг використання хмари для CI/CD є динамічний характер хмарної інфраструктури. Хмарні ресурси можуть автоматично масштабуватися у напрямку зростання та спадання на основі навантажень в CI/CD. Це забезпечує величезну економічну вигоду, адже підприємствам не потрібно мати власні сервери для обслуговування.

Рис. 2.13. Хмарні платформи

2.1.2. Додаткові знання для DevOps

Було проаналізовано та розібрано інструменти, підходи та стратегії в етапах Continuous integration/delivery розробки, хоч інструменти в набагато меншому обсязі, що буде виправлено в наступному параграфі. Та перед тим, як почати розбирати і порівнювати інструменти більш детально, хотілося б зауважити, що є ще декілька знань без яких DevOps розробнику буде важко впроваджувати свої методології. Наведено перелік:

- System Administrator.

Ідея у тому, що DevOps розробнику бажано розбиратися на гарному рівні в середовищі, де будуть працювати його програми. Як вони запускаються, що роблять, як працювати з помилками, чим користуватись, а чим ні. Всі ці знання сильно знадобляться, коли він буде запускати справжні проекти.

- Networking – CCNA.

Інтернет мережа – надто важлива річ, хоча цим нехтують багато розробників. Буде дуже важко писати онлайн-сервіси, не розуміючи, як вони працюють. Не говоримо про те, що потрібно завчати всі рівні моделі OSI, але Вам точно знадобляться знання, як працюють IP, TCP/UDP і звісно, протокол рівня програми – наприклад, HTTP, HTTP/2.

- Developer.

DevOps бажано розуміти, як пишеться код, що таке ООП, потоки та ще багато різних речей. Загалом, чим знань у цьому пункті, тим легше буде збирати та виконувати програму.

- DBA.

База даних – це організоване забрання даних, завдяки якому можна легко отримати доступ та керування до них. Оскільки всі теперішні додатки пишуться з використанням баз даних, DevOps розробнику необхідно розбиратися в синтаксисі SQL [15].

2.2. Порівняння інструментів

Розглянуто інструменти, які обрані безпосередньо для реалізації демонстративної практичної частини, а також розглянуто ті інструменти, які не були використані в ході практики. Порівнюємо інструменти.

Перш за все, розглянуто інструменти керування версіями. Обрано Git в противагу CVS та SVN у табл. 2.1 та табл. 2.2 [16].

Таблиця 2.1

Порівняння Git та CVS

Характеристики	Git	CVS
Просте та швидке розгалуження	+	-
Підтримує автономну роботу, локальні уявлення можуть бути надіслані на сервер пізніше	+	-
Коміти є автомарними та застосовними для всього проекту	+	-
Кожне робоче дерево містить репозиторій з повною історією проекту	+	-

Таблиця 2.2

Порівняння Git та SVN

Характеристики	Git	SVN
Працює у розподіленому режимі	+	-
Зберігає дані у вигляді метаданих	+	-
Кожна гілка є особливою	+	-
Є глобальний номер версії	-	+
Цілісність вмісту	+	-

Інша група інструментів – контейнеризація і тут є фаворит – docker. У нього також є свої конкуренти в сфері контейнеризації такі як: Podman, Buildah, Buildkit, Kaniko, Skopeo та інші (рис. 2.13), але робити таблицю і порівнювати їх між собою не є доцільним, оскільки Docker ще довго буде займати лідируючі місця [17].

Рис. 2.14. Інструменти контейнеризації

Також необхідно порівняти Docker та Віртуальні машини (VM), так як ці “інструменти” поєднані таким поняттям, як віртуалізація, хоча під демонстраційний проект підходе саме Docker (рис. 2.15). VM дозволяє запускати віртуальні машини на будь-якій машині, а Docker дозволяє запускати додаток або код в ізольованому процесі на будь-якій машині. Більш детально про їх відмінності наведено у табл. 2.3.

Таблиця 2.3

Порівняння Docker та VM

Docker	VM
Контейнери Docker, навпаки, розміщуються на одному фізичному сервері з операційною системою хоста, яка розділяє їх між собою. Спільне використання ОС хоста між контейнерами робить їх легкими та збільшує час завантаження.	Віртуальні машини мають як хостову ОС так і гостьову всередині кожної віртуальної машини.
Контейнерна технологія має доступ до підсистем ядра; в результаті один заражений додаток здатний зламати всю хост-систему.	Віртуальні машини автономні зі своїм ядром та функціями безпеки. Тому програми, які потребують додаткових привілеїв та безпеки, працюють на віртуальних машинах.
Docker-контейнери є автономними і можуть запускати програми в будь-якому середовищі, і оскільки їм не потрібна гостьова ОС, вони легко переносяться на різні платформи	Віртуальні машини є ізольовані від їхньої ОС, тому вони не можуть переноситися на декілька платформ.
Полегшена архітектура контейнерів є менш ресурсомісткою, ніж віртуальні машини.	VM є більш ресурсомісткими, ніж контейнери Docker, оскільки віртуальні машини повинні завантажувати всю ОС для запуску

Хоча як технології віртуалізації, в них є деякі збіжності. І контейнери Docker і віртуальні машини створюються з образів. Кожен образ служить кресленням віртуалізованого середовища. Образи дозволяють користувачам створювати узгоджені середовища та спільно використовувати їх без необхідності щоразу налаштовувати їх.

У образі вказані всі необхідні системні ресурси запуску додатків. Наприклад, образ віртуальної машини створює резервні копії операційної системи, а образ контейнера Docker – резервну копію середовища програми.

Також версіями образів як контейнерів Docker, так і віртуальних машин можна керувати для відстеження змін конфігурації середовища з часом. Управління версіями в Docker свідчить про можливість відстежувати зміни образів Docker та керувати ними з часом. Це дозволяє розробникам відстежувати різні версії додатків, при необхідності повертатися до попередніх версій і одночасно розгортати різні версії.

Аналогічним чином управління версіями у віртуальних машинах значить процес відстеження змін образу віртуальної машини та управління ними з часом. Керування версіями віртуальних машин відстежує зміни (наприклад, оновлення та виправлення) у конфігурації віртуального обладнання або операційної системи.

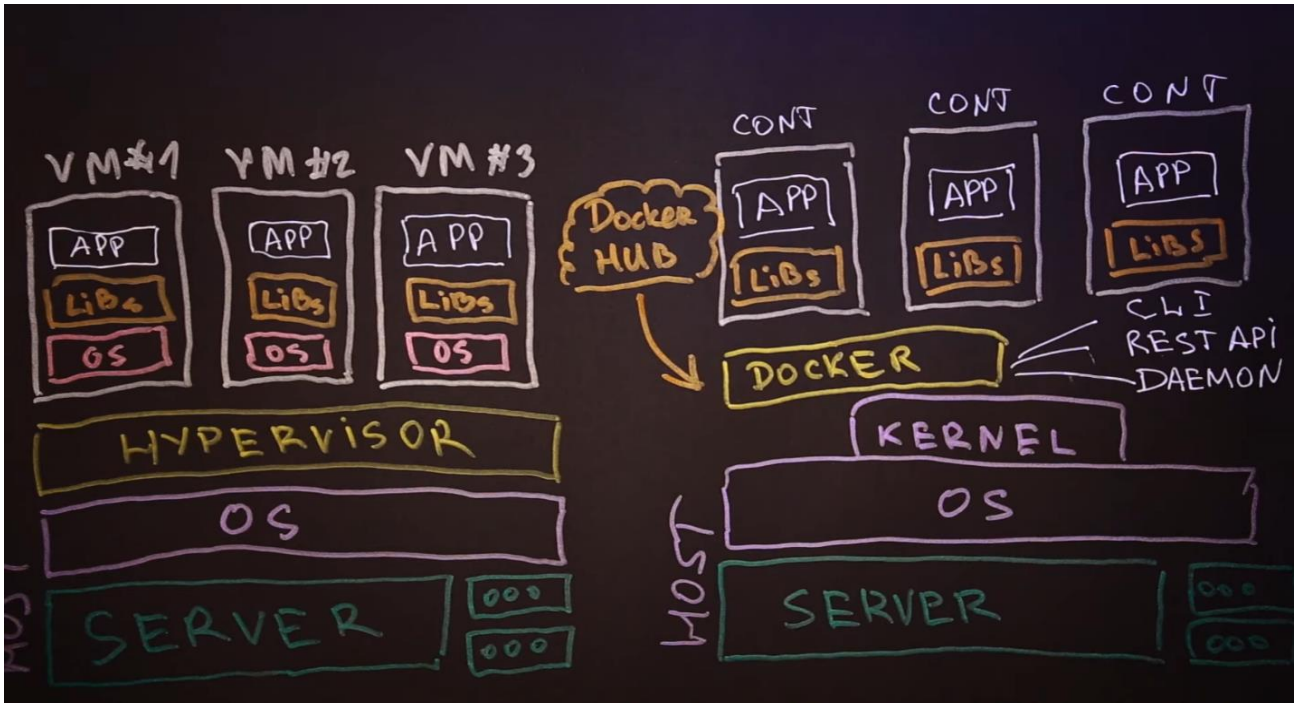

Рис. 2.15. Відмінності Docker та VM на рівні віртуалізації

Як висновок, контейнери Docker не конфліктують з віртуальними машинами, вони обидва є додатковими інструментами для різного робочого навантаження та використання [18, 19].

Наступний вид інструментів – “інфраструктура як код”. Ця практика передбачає, що код інфраструктури буде накопичуватися і потім буде можливість його автоматизувати. Також зникне потреба у переліках процедур та внутрішньої документації. Процеси стануть відтворюваними, а системи – надійними. В даній роботі було використано Terraform в противагу схожому інструменту Ansible. Хоча ці два інструменти можна використовувати разом, тим самим створюючи кращий досвід для команд розробників і команд експлуатації. Різниця між ними наведена у табл. 2.4.

Таблиця 2.4

Порівняння Terraform та Ansible

Terraform	Ansible
Обидва інструменти заплановують, конфігурують та курують інфраструктурою	Обидва інструменти заплановують, конфігурують та курують інфраструктурою
Terraform краще працює з запланування̆м	Ansible краще працює з конфігуруванням
Може розгортати програми	Також може розгортати програми, але гірше ніж Terraform
Краще працює з оркестрацією	Поганий в оркестрації

Наступний інструмент, що було використано – GitLab CI. Він допомагає налаштовувати процес безперервної інтеграції та розгортання. В GitLab CI є багато альтернатив і кожен з них є гарним інструментом. Ідея GitLab CI, як інструменту для безперервної інтеграції є доволі простою: створюємо файл з назвою .gitlab-ci.yml безпосередньо у репозиторію проекту та налаштовуємо додаткові сервери Runner, тоді кожне нове залиття та з’єднання гілки буде запускати pipeline.

Наведено основні переваги:

- Зручна система з чудовою підтримкою Docker/Kubernetes;

- Система контролю версій та система безперервної інтеграції в одному інструменті;
- Легко налаштувати та масштабувати сервер збірок, тобто GitLab-Runner;
- Безкоштовна система з відкритим кодом;
- Паралельне виконання Jobs;
- Легко вирішувати конфлікти;
- Підтримка хмарного рішення і сервера;
- Код і Pipeline в одному місці. Зручне оформлення Pipeline.

Недоліки:

- Майже неможливо перевірити стан проєкту до того, як безпосередньо відбудеться злиття гілки.
- Артефакти треба визначати та завантажувати окремо у кожному випадку;

Обрано GitLab CI за рахунок гарного вибору для роботи з Git та Docker-контейнерами. На рисунку нижче наведено приклад альтернативних інструментів (рис. 2.16).

Рис. 2.16. Найпопулярніші інструменти CI/CD

І останнє, але не менш важливе – це хмарні платформи. Переведення локальної інфраструктури в хмарне середовище дозволяє зменшити капітальні витрати, також витрати на адміністрування, допомагає вирішувати питання захисту та резервування інформаційних ресурсів. У поєднанні з впровадженням DevOps-технологій, хмарна інфраструктура може стати потужним засобом автоматизації та оптимізації ресурсів. Використано хмарну платформу GCP – Google Cloud Platform, оскільки вона не потребує окремого серверу та коштує дешевше. На думку більшості, функціоналом вони майже не відрізняються, GCP по декількох пунктах навіть відстає (рис. 2.17), але він дає користувачу 91 день безкоштовного користування майже всіма його сервісами, в той час, як Azure та AWS дають набагато менше. Також Azure та AWS мають під собою багато підводних каменів. Перевагою GCP є його більш сприятливий та зручний інтерфейс (user friendly), на відмінну від його конкурентів.

Рис. 2.17. Різниця функціоналу різних хмарних платформ

2.3. Аналіз підходів розгортання коду (Continuous deployment)

Було розібрано інструменти, підходи та стратегії в етапах Continuous integration/delivery розробки, тепер прийшов час розбору стратегій або підходів

Continuous deployment. В чому ж різниця між delivery та deployment, це питання вже розбиралося в першому розділі, тож коротко для повторення. По-перше в deployment йде автоматичне розгортання наших змін на більш високі середовища автоматично. В continuous delivery може бути налаштований трігер на якесь дійство (наприклад approve). Тобто, якщо все добре з артефактом і він вдало пройшов тести, то є можливість перекинути його на наступну середу. В deployment перехід між середовищами йде автоматично. По-друге є можливість безпечно розгортати додаток у фінальне виробництво без впливу на клієнтів, для цього використовують deployment strategies про які буде йти мова далі. Та третє, deployment дозволяє зменшити час з моменту, як було придумано якусь нову функціональність до моменту її реалізації, коли вже буде можливість нею користуватися (lead time), та зменшити кількість помилок.

Щоб deployment пройшов вдало та не нашкодив користувачам, для такого існують deployment strategies. Їх ще інколи називають zero downtime deployment strategies. Процес безперервного розгортання не може бути створеним без цих стратегій, бо він реалізується на них, continuous deployment не можу існувати без застосування стратегій. Важливо розібратися, яка саме стратегія розгортання буде нам підходити, бо кожна із них має свою ціль.

- Recreate (повторне створення)

Сама проста стратегія, зносимо стару версію повністю замінюючи на нову (рис. 2.18). Не завжди вона є автоматизована і вона не дозволяє нам зробити zero downtime в нашому розгортанні. Zero downtime – це коли клієнт користується додатком і йде розгортання нової версії додатку але клієнт все ще може ним користуватися. Коли ж на проекті немає цього процесу, при розгортанні нової версії додатку клієнт не зможе ним користуватися, програма покаже помилку, що сервер не відповідає або щось типу того. Запобігти цього можна тільки якщо це не проводиться дуже швидко, або ж робиться в часи, коли клієнти не користуються додатком. Наприклад оновлення офісного ПЗ, коли люди підуть з роботи додому, навряд чи ним хтось буде користуватися. Також можна висилати повідомлення в який час будуть проводиться роботи над додатком.

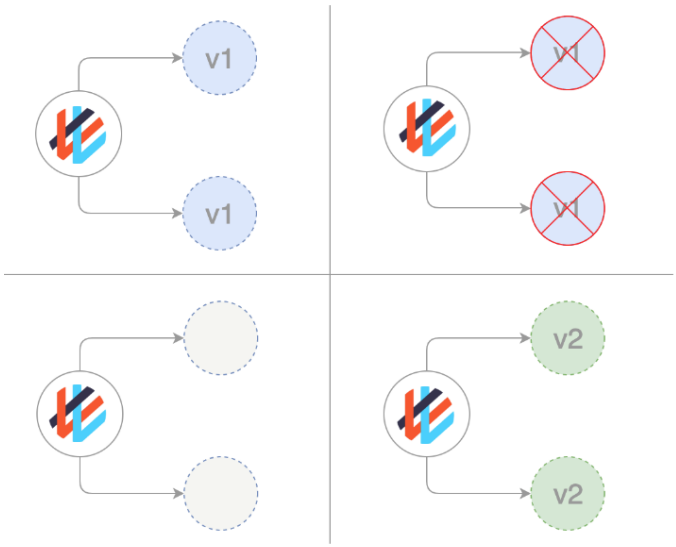

Рис. 2.18. Recreate

- Ramped (поступовий)

Це стандартна стратегія розгортання. Поступово, один за одним, замінюємо старі “версії” на нові – без простою кластера. Чекаємо коли нові версії будуть готові до роботи (перевіряємо їх на авто-тести), перш ніж почнемо закривати старі. Якщо виникає якась похибка, то оновлення можна зупинити, не зупиняючи усього кластеру (рис. 2.19).

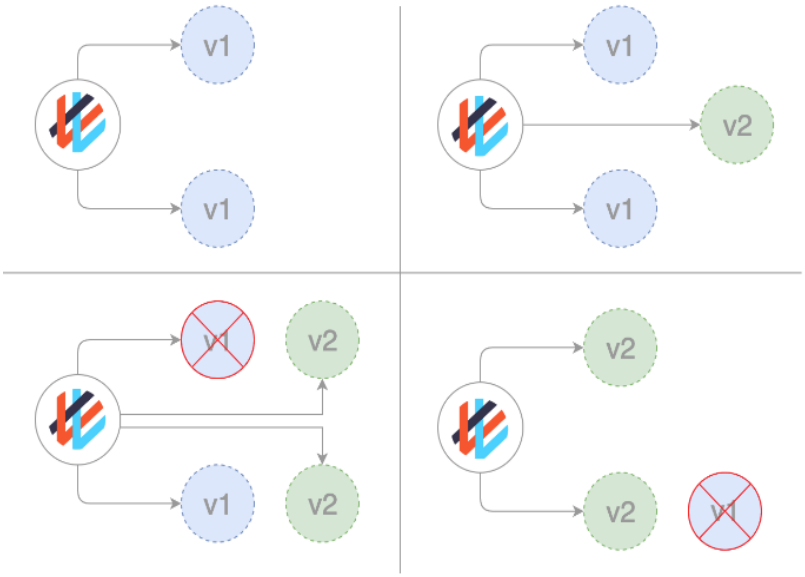

Рис. 2.19. Ramped або Rolling

- Blue/Green

Сама популярна стратегія та найчастіше використовується. Це стратегія синьо-зеленого розгортання, інколи її ще називають red/black (червоно-чорна),

передбачає одночасне розгортання старої (зеленої) та нової (синьої) версій програми. Після розміщення обох версій звичайні користувачі отримують доступ до зеленої, в той час як синя доступна для тест-команди для автоматизації тестів через окремий сервіс або пряме прокидання портів. Після того, як синя (вже нова) версія була протестована та була сприятливо сприйнята для релізу, сервіс переключається на неї, а зелена (стара) звертається (рис. 2.20). Єдиним мінусом цієї стратегії можна назвати її вартість, бо треба підтримувати два середовища одночасно.

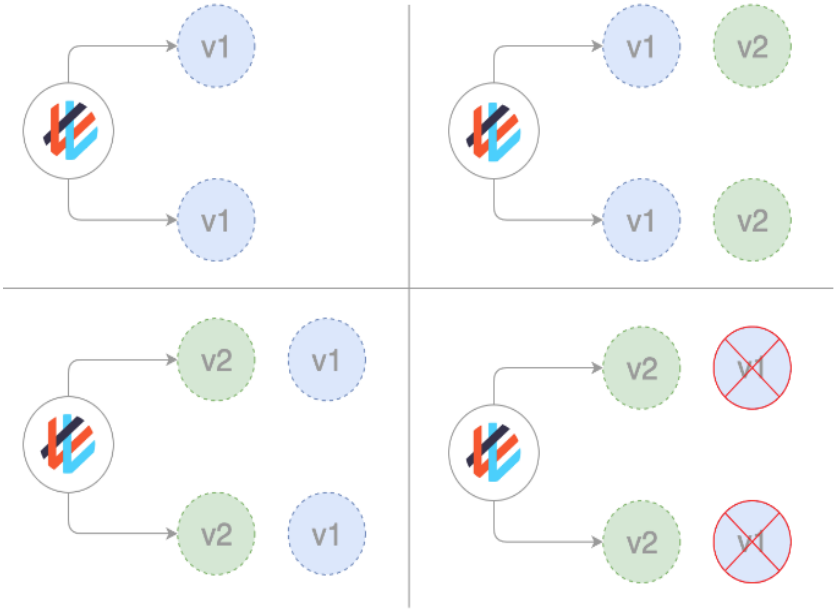

Рис. 2.20. Blue/Green

- Canary

Canary розгортання дуже схоже на синьо-зелене, але краще управляється та використовується в прогресивних поетапних підходах. До таких типів підходів є декілька різних типів стратегій, включаючи “приховані” запуски та A/B-тестування.

Цей підхід застосовується, коли необхідно випробувати якусь нову функціональність, як правило, у бекенді програми. Сенс підходу в тому, щоб створити два практично однакові сервери: один обслуговує майже всіх користувачів, а інший, з новими функціями, обслуговує лише невелику підгрупу користувачів, але користувачі вибираються випадково, після чого результати їхньої роботи порівнюються. Якщо все відбувається без помилок, нова версія

поступово викочується на всю інфраструктуру (рис. 2.21) [20]. Також ця стратегія допомагає нам зрозуміти, як наш додаток реагує на реальний трафік від наших користувачів

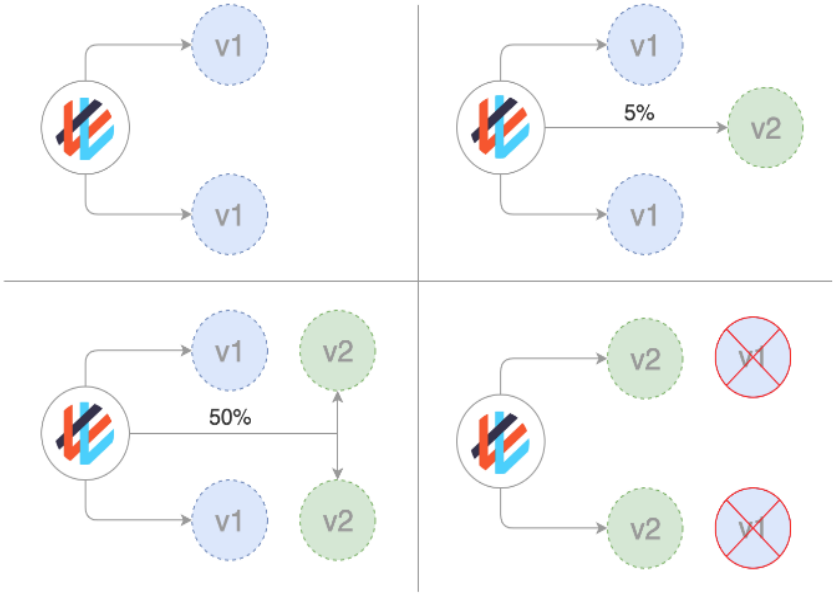

Рис. 2.21. Canary

Є доволі цікави факт пов’язаний з назвою цієї стратегії. Сама по собі назва є найменням такої птиці, як канарка. Колись цю птицю використовували у вугільних шахтах, для того, щоб зрозуміти який там рівень газу. Запускали птицю і дивилися скільки часу вона там могла протриматися, якщо вона залишалася живою протягом довгого часу, то в цій шахті можна працювати.

- A/B testing

Остання стратегія розгортання це A/B testing (рис. 2.22). Вона дуже схожа на Canary, але тут обираються певні користувачі для серверу з новим функціоналом. Таким чином ми можемо побачити, як вибрані для тесту користувачі будуть реагувати на певні оновлення. Це сама складна стратегія, бо тут потрібна реалізація своєї певної логіки, яких користувачів ми будемо обирати і для чого.

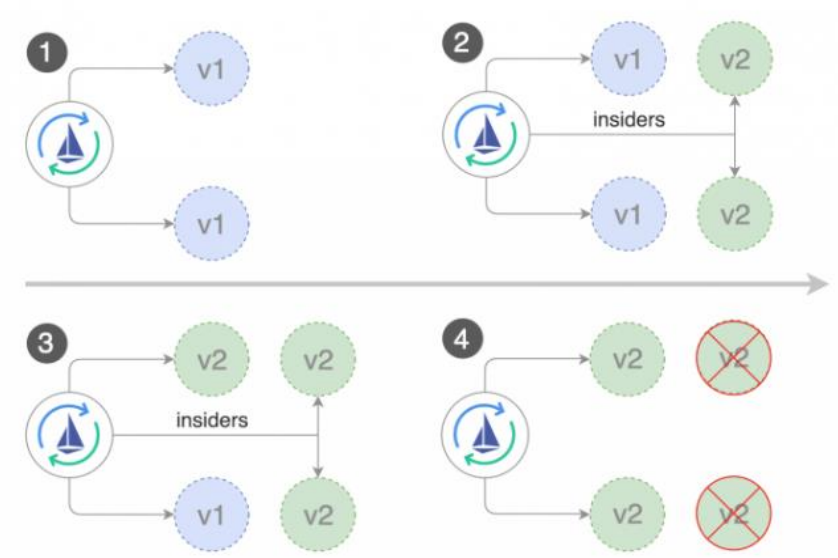

Рис. 2.22. Dark або A/B – розгортання

Мабуть самими популярними будуть Canary, Blue/Green та Ramped стратегії. Recreate є самою звичайною та простою, а A/B testing доволі складна в реалізації (рис. 2.23).

Strategy	ZERO DOWNTIME	REAL TRAFFIC TESTING	TARGETED USERS	CLOUD COST	ROLLBACK DURATION	NEGATIVE IMPACT ON USER	COMPLEXITY OF SETUP
RECREATE version A is terminated then version B is rolled out	✗	✗	✗	■□□	■■■	■■■	□□□
RAMPED version B is slowly rolled out and replacing version A	✓	✗	✗	■□□	■■■	■□□	■□□
BLUE/GREEN version B is released alongside version A, then the traffic is switched to version B	✓	✗	✗	■■■	□□□	■■□	■■□
CANARY version B is released to a subset of users, then proceed to a full rollout	✓	✓	✗	■□□	■□□	■□□	■■□
A/B TESTING version B is released to a subset of users under specific condition	✓	✓	✓	■□□	■□□	■□□	■■■

Рис. 2.23. Стратегії розгортання

Підводячи підсумки, було майже повністю розглянуті етапи безперервної інтеграції, доставки та розгортання коду, розібрано велику кількість інструментів та стратегій у цих підходах. Також ще є безліч технологій та підходів, але розглянутий загальний план є доволі гарним типовим прикладом

CI/CD процесу. Маючи ці знання, залишається вибрати ті стратегії та інструменти, які відповідатимуть проекту і почати реалізовувати їх. Існує бажання ще раз, але швидко підвести висновки по підходам Continuous integration/delivery та deployment. Все починається з безперервної інтеграції - це постійні, систематичні інтеграції коду з основною гілкою розробки. Не обов'язково треба тільки заливати код, час від часу код з головної гілки треба стягувати, щоб не забути, що там взагалі діється і не було майбутніх не співпадінь. Основні гілки знаходяться в окремих репозиторіях, постійно підтримуємо код в робочому стані, перевіряємо його через Code Review. Для усієї роботи з кодом використовується одне робоче середовище, усе тестується та збирається на одному build server. Не повинно такого бути, що на машині розробника воно працює, а на машині іншого ні, усе в одному місці. Після отриманого артефакту йде перехід до безперервної доставки. Тобто базуємося на тому, що отримано від безперервної інтеграції, плюс автоматично доставляємо додаток на інші створені середовища. Кожен артефакт, це потенціальний кандидат на постановку до фінального користувача, але щоб остаточно викласти його, для цього потрібна людина, яка буде керувати цим процесом. Щоб автоматизувати це, в гру вступає безперервне розгортання, яке базується на стратегіях, розглянутих раніше.

2.4. Нові виклики для DevOps

Розібрано безліч позитивних аспектів, які впроваджує методологія DevOps та етапи CI/CD підходу. Хоч DevOps і приніс ковток свіжого повітря у розробку ПЗ, але вже пройшло немало часу, близько 13 років. Тож є необхідність розповісти про можливі сучасні недоліки методології, та запропонувати рішення які могли б допомогти з цим. Здебільшого це пов'язано з новими непередбаченими викликами, які виникли у розробці та в експлуатації. Спільною проблемою для всіх є [21]:

- Швидші цикли випуску змушують обидві команди постійно бути напоготові
- Технічний борг, утворений внаслідок скорочення термінів, за який треба платити вже в найближчий день
- Жодна сторона не може зосередитися на своїй основній компетенції

Ну і саме головне, це кількість інструментів[22]. Використовуючи інструменти DevOps без наскрізної платформи, команди ускладнили свою роботу, збільшили витрати та головний біль. Перехід на справжню платформу Devops — це спосіб вийти з-під усього цього й отримати контроль над проектами, зруйнувати розриви та розвивати співпрацю.

Сучасні компанії все частіше звертаються до DevOps для більш ефективного та безпечного створення програмного забезпечення. Однак не всі з них прийняли єдину платформу DevOps, натомість вирішили об’єднати безліч інструментів для обробки всього життєвого циклу розробки програмного забезпечення – від планування до доставки та релізу. Звичайно, інструменти DevOps корисні, але і хороших речей може бути забагато.

Сьогодні спроба використання великої суміші інструментів змушує членів команди постійно перемикатися між кількома інтерфейсами, паролями та способами роботи. Це також створює хаотичне середовище, яке потрібно нескінченно оновлювати та утримувати. І використовуючи безліч різноманітних інструментів, ніхто не отримує загального уявлення про проекти, над якими вони працюють.

Є дуже цікавий підхід, як self-serve DevOps, чи допоможе він? Вже існують проблеми, які заважають швидкому й ефективному розвитку.

Щоб спробувати це виправити, деякі компанії дійшли до підходу «самообслуговування DevOps». Self-serve DevOps обіцяє зняти рутинні вимоги з плечей операційних команд. Яким чином? Розробникам просто надаватимуть усі необхідні інструменти, інфраструктуру та послуги за допомогою простих ефективних запитів. Це усуває діри та спрощує робочий процес розробки, дозволяючи їм повернутися до кодування, не чекаючи участі Ops. Сьогодні існує

кілька способів впровадження DevOps із самообслуговуванням, принаймні певною мірою:

- внутрішня платформа розробника;
- засоби автоматизації робочого процесу;
- сервісні каталоги;
- торгові майданчики API;

На жаль, цей підхід теж не є ідеальним. Зазвичай він просто додає ще один рівень складності, тим самим віддаляючи команди від їхніх основних завдань. Та і багато цих рішень покладаються на залучення “людини посередині” і кількість інструментів не зникла. Тобто, майже нічого не змінилося.

Має бути єдиний і оптимізований спосіб доступу до всього робочого процесу DevOps. Щоб розробникам та DevOps-інженерам не треба було освоювати нові інструменти та постійно отримувати доступи. Типу хмарна інфраструктура, CI/CD та постачальники ідентифікаційних даних. Немає нових систем для вивчення — немає перемикання контексту. Лише один простий інтерфейс, щоб керувати усім.

Існує доволі новий підхід який називається headless. Термін «headless» з’явився у світі вебсайтів (CMS) та електронної комерції. Він означає, що інформація (контент, записи в каталозі, ціни тощо) незалежні від презентації. Цей підхід допомагає під іншим кутом подивитися на DevOps.

Якщо ми прагнемо зробити свої команди DevOps більш продуктивними. Яку б систему ми не обирали, будь то Slack або Teams (або навіть CLI), нам потрібно перетворити її на універсальний інтерфейс користувача DevOps. Для чого це:

- Розробники виграють, тому що вони отримають доступ до необхідної хмарної інфраструктури. Зможуть запускати конвеєри тощо. І це все без досвіду в цих сферах. Крім того, вони можуть робити це безпечно завдяки вбудованому контролю.
- Команди Ops виграють, тому що вони звільняться від повсякденної рутини, зосередившись на більш загальних проблемах.

Ключовою перевагою впровадження “безголового” є відокремлений характер підходу. Це дає змогу підтримувати декілька платформ, таких як веб-сайти, мобільні програми, пристрої IoT, голосові помічники, AR та віртуальну реальність (VR) з одного джерела [23].

Також додатковими перевагами впровадження headless, такі як підвищення продуктивності, швидший час виходу на ринок і швидке впровадження інновацій. Ці позитивні властивості мають сенс, тому що за допомогою headless програми запитують лише ті дані, які їм потрібні. Таким чином, headless, безсумнівно, може сприяти більш динамічним додаткам і персоналізованим користувачам.

Словом, з використанням нових підходів команди будуть працювати ефективно, як ніколи раніше.

2.5. Висновок до другого розділу

У другому розділі було проведено дослідження інструментів, стратегій та підходів в continuous integration, continuous delivery та continuous deployment. Виділено та досліджено ключові етапи циклу DevOps: quality gateway, планування, збірка, моніторинг безперервний зворотній зв’язок та хмарні платформи. В кожному з цих етапів був проведений аналіз стратегій та технологій, в яких випадках їх треба використовувати, наведені їх переваги та недоліки.

Також розглянуті ті додаткові знання, в яких DevOps розробнику необхідно розібратися, щоб вважати себе кваліфікованим спеціалістом. Без цих знань розробнику буде дуже важко впроваджувати методології в команію.

Був проведений аналіз підходів в continuous deployment. Такі підходи інколи називають zero downtime deployment strategies. Вони використовуються для прискорення життєвого циклу розробки програмного забезпечення, впроваджуючи нульові простої в розгортанні прогам. Розгортання без простоїв дають змогу команді вносити зміни під час роботи програми.

З’ясували типові мінуси DevOps методологій, які виникають у команд розробки та експлуатації. Для їх вирішення були запропоновані сучасні підходи self-serve Dev-Ops та headless. Завдячуючи цим підходам, вдається подивитися на DevOps під іншим кутом. Ці підходи, а особливо headless, дають змогу одразу декілька платформ, прогами починають запитувати лише ті дані, які їм потрібні та зменшується кількість інструментів, які використовуються в CI/CD підходах.

РОЗДІЛ 3
РОЗРОБКА CI/CD ПРОЦЕСУ

3.1. Розробка демонстраційного CI/CD pipeline

Розглянемо процес побудови власного CI/CD. По-перше, треба розробити план: які інструменти будуть використовуватися, яку стратегію розгортання коду буде обрано, яку хмарну платформу вибрано та інше. Створена заготовка, яка буде використана в демонстративному проектові (рис. 3.1).

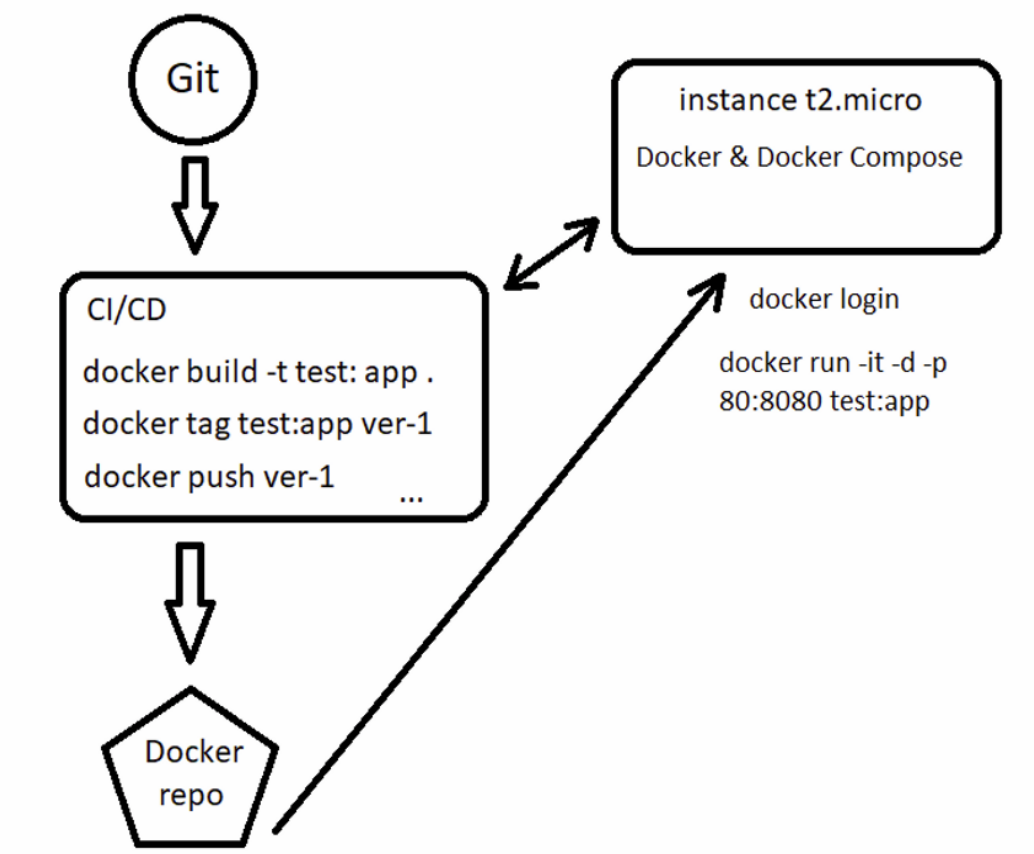

Рис. 3.1. CI/CD pipeline

Розглянуто рис. 3.1. Додатоком, над яким буде розгортатися pipeline, є типовий сайт зі своєю базою даних. Після створення сайту та бази даних до нього, наступним пунктом йде контейнеризація через Docker.

Docker дає змогу упакувати додаток в контейнер, так щоб кожна людина могла без проблем його скачати та користуватися ним (рис. 3.2).

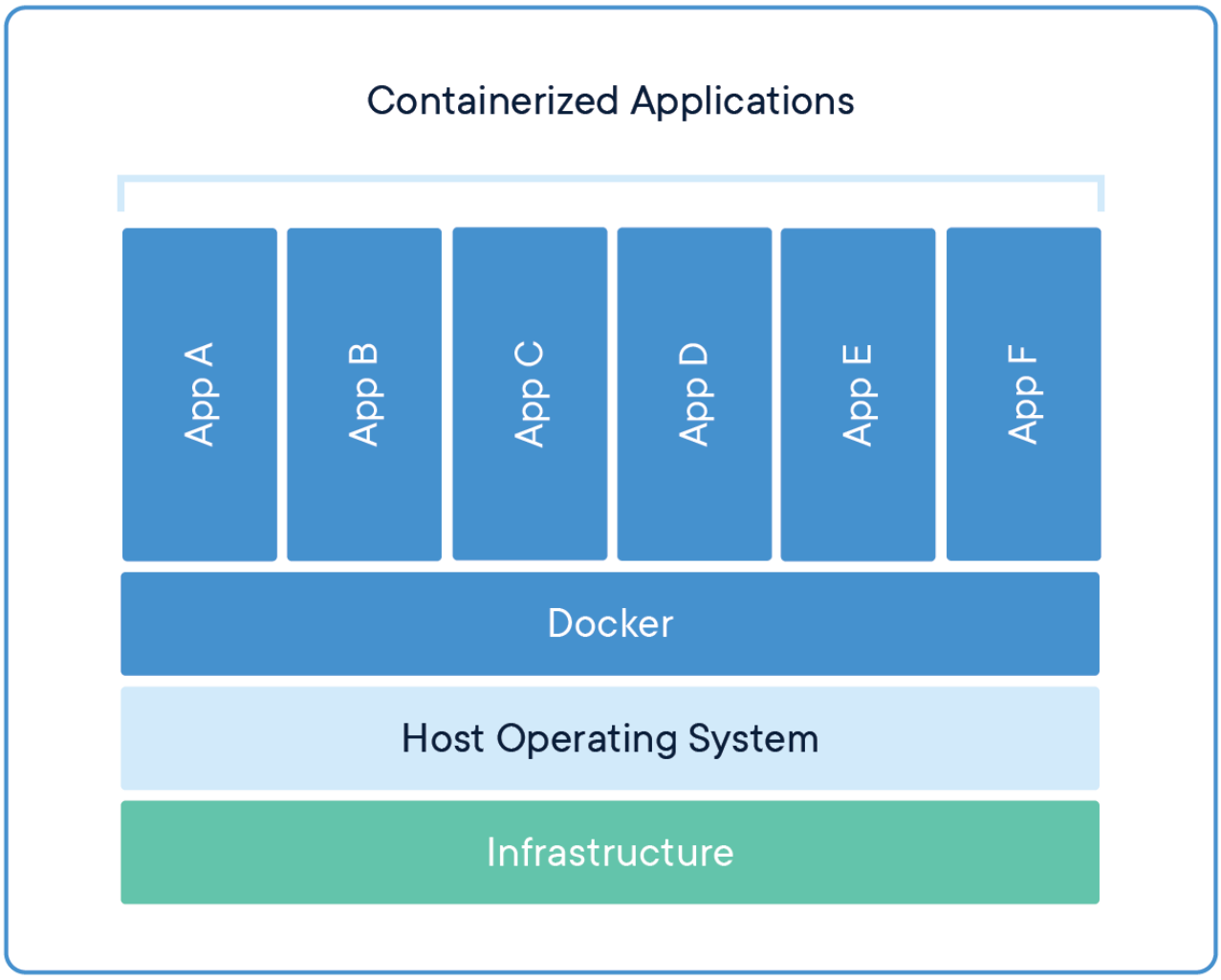

Рис. 3.2 Приклад docker контейнеру, як ізольованого процесу

Контейнером є спосіб упакування нашого додатку з усіма необхідними для нього залежностями та конфігурацією в середину контейнера. Створюється свого роду артефакт, яким легко ділитися та користуватися на різних операційних системах. Таким чином, роблячи розробку набагато ефективнішою.

Важливим питання є, де зберігати docker контейнер, адже контейнеру треба десь зберігатися, щоб була можливість ділитися ним з іншими? Контейнери зберігаються в спеціалізованих репозиторіях створених саме для цього (container repository) (рис. 3.3). Багато компаній мають свої приватні репозиторії, де вони зберігають свої контейнери. Також є публічні репозиторії, де можна знайти майже любий налаштований контейнер під конкретні додатки з необхідними залежностями. Самим популярним репозиторієм таких контейнерів є DockerHub (рис. 3.4).

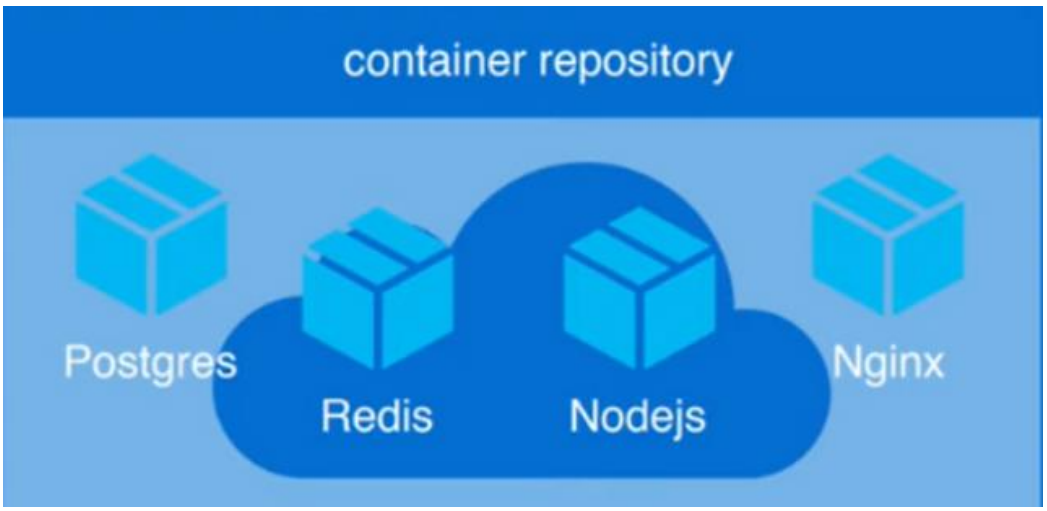

Рис. 3.3. Container repositories

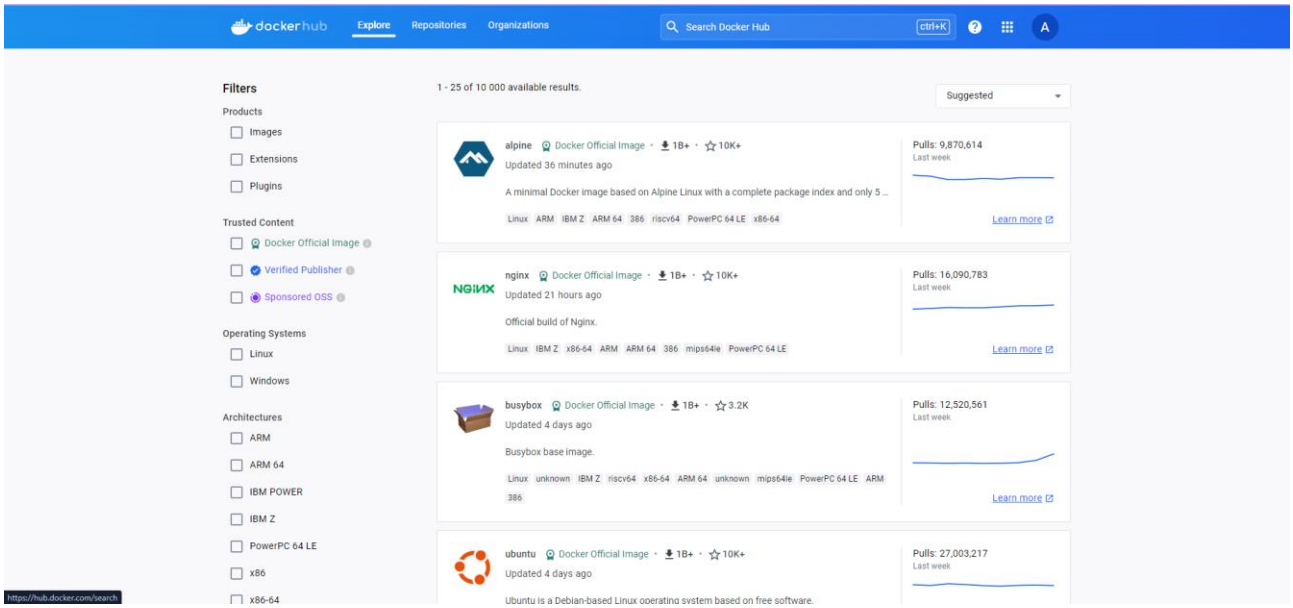

Рис. 3.4. Головна сторінка DockerHub

Якби користувач хотів установити створюваний продукт собі локально на комп’ютер, йому би знадобилося встановлювати усі додаткові залежності, щоб просто запустити сайт (локально). З Docker цих проблем не буде, так як при контейнеризації усі залежності також туди заносяться, це свого роду архів. Також разом з Docker треба настроїти Docker-compose (рис. 3.5). Docker-Compose це інструмент оркестрації контейнерів, який існує для того, щоб людина могла запускати відразу кілька контейнерів, так як не зручно було б запускати кожен контейнер окремо через команди Docker. Цей інструмент добре підходить для середовищ розробки, тестування та постановки.

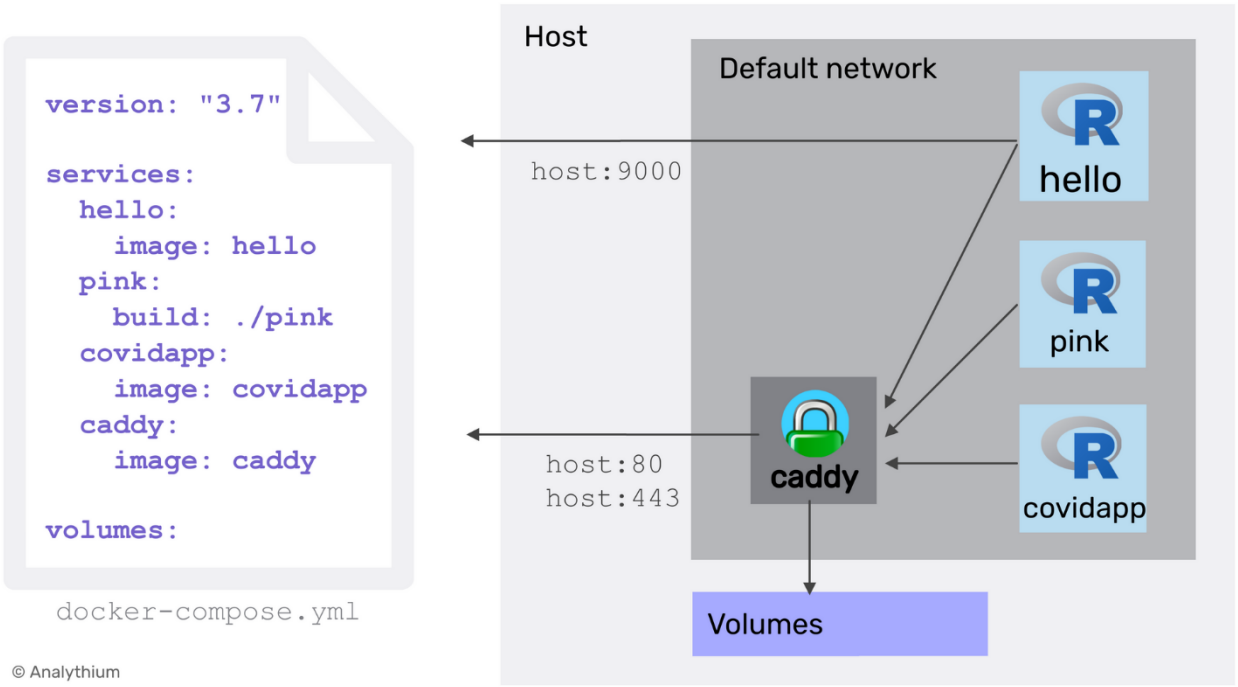

Рис. 3.5. Docker-compose

Існує ще одна технологія оркестратор – Kubernetes (рис. 3.6). Це більш потужний та універсальний інструмент. Він є галузевим стандартом оркестровки контейнерів і використовується компаніями будь-якого розміру. Kubernetes також було прийнято багатьма хмарними провайдерами, включаючи Amazon, Microsoft і Google. Kubernetes має такі переваги як: неймовірну масштабованість, високу надійність, гнучкість, вбудовані можливості самовідновлення, підтримку хмарних платформ, тощо. Але для демонстративного типового pipeline обрано Docker-compose, за його менший поріг входу знань для його користування та його зручність для маленьких проектів.

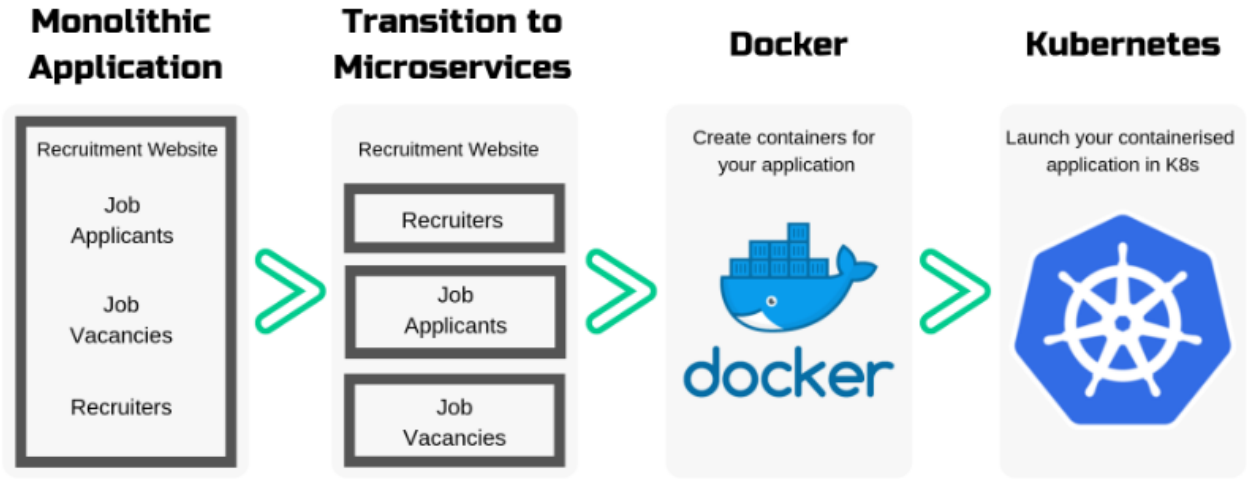

Рис. 3.6. Перехід до Kubernetes

Далі увесь код та Docker контейнери треба залити до GitLab, користуючись інструментом керування версіями – Git. GitLab чудовою DevOps платформою, платформою на якій можна будувати власний робочий процес DevOps (рис. 3.7).

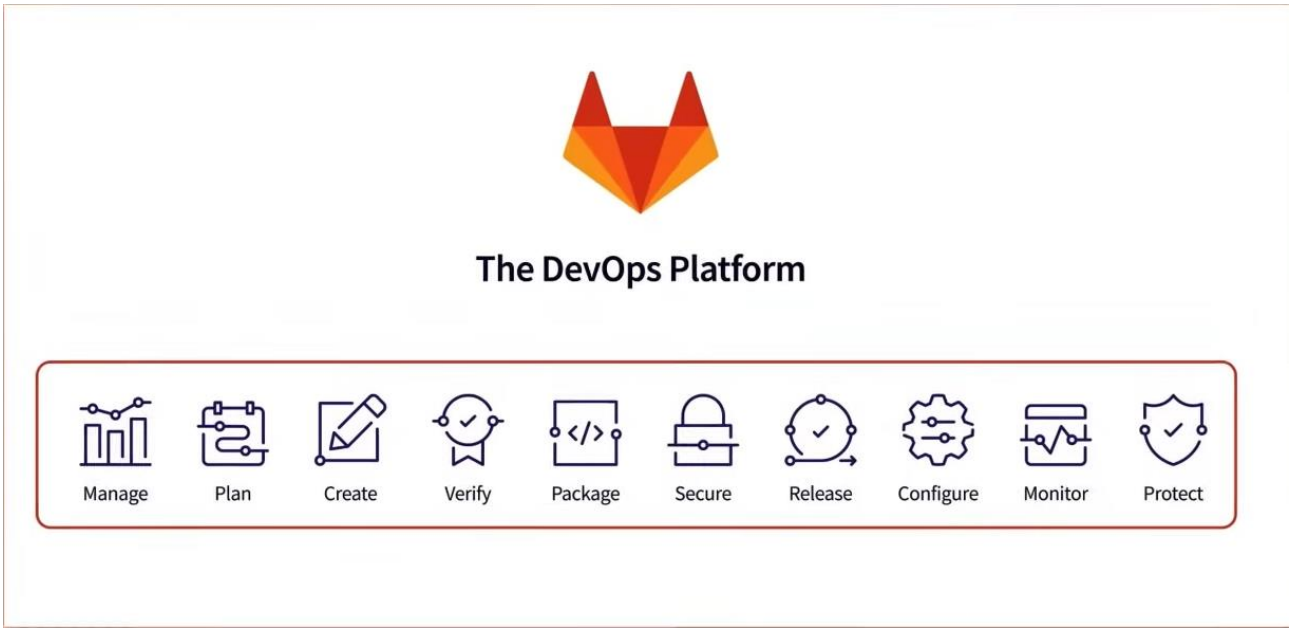

Рис. 3.7. The DevOps Platform on GitLab

Після вдалого відправлення коду до GitLab, скористаємося влаштованим в платформу (GitLab) розширенням GitLab CI. Це свого роду CI/CD інструмент для безперервної інтеграції та розгортання коду. В ньому ми повинні прописати команди для автоматичного розгортання. Сенсом цих команд буде збірка наших Docker контейнерів та відправлення їх до Docker registry. Docker registry – це місце де зберігаються images (docker контейнери), щоб можна було їх в любий час звідти дістати та розгорнути для використання, це свого роду версії. Наступні команди повинні будуть з'єднатися з віддаленими машинами, скоріш за все через ssh (протокол рівня програм, що дозволяє проводити віддалене управління комп'ютером), які будуть створені Terraform-ом на базі хмарної платформи Google Cloud Platform, на яких і буде розгортатися демонстративний сайт. Команди буде виконувати GitLab Runner (рис. 3.8). Сенс цього інструмента – зняти навантаження з головного серверу GitLab CI. Ще GitLab Runner можна називати, як агент. Цей агент, як вже промовлялося, буде знімати навантаження з головного серверу, виконуючи команди CI/CD pipeline.

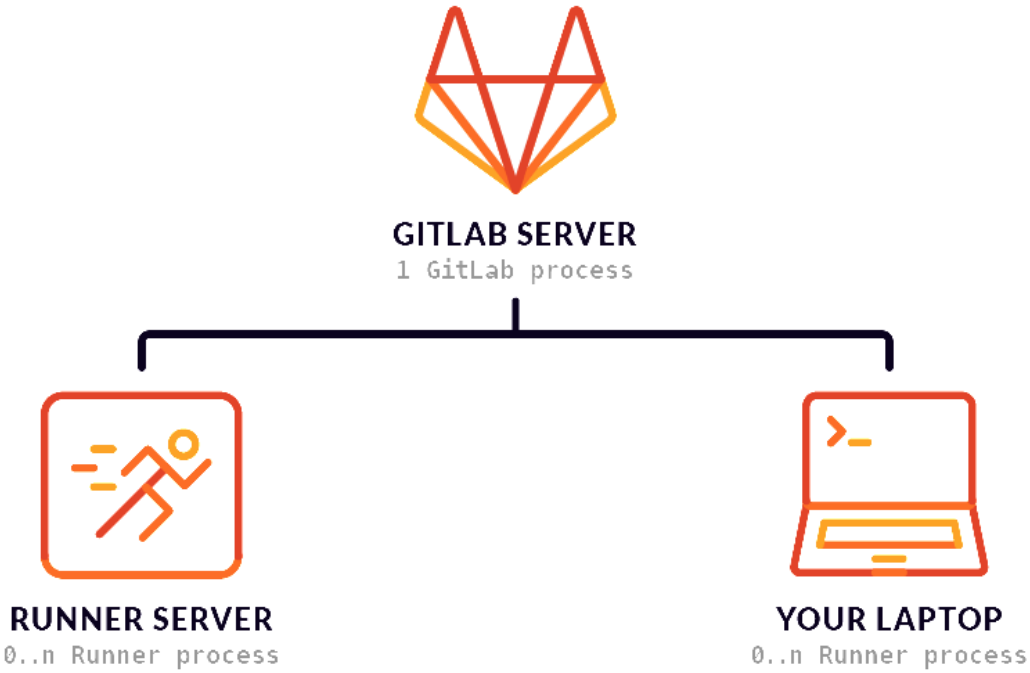

Рис. 3.8. GitLab Server and Runner

Ну і нарешті, після того як команди відпрацюють, отримуємо налаштований , а головне – працюючий сайт, який буде постійно оновлюватися при нових відправленнях або з'єднаннях коду до GitLab і буде постійно працювати на серверах в GCP. Також, можна додати які-небудь системи моніторингу типу Splunk, щоб відслідковувати стан серверів та додатку.

3.2. Етапи побудови демонстративного CI/CD pipeline

Продукт, який будt розгортатися, є типовий веб-сайт з маленькою базою даних. Ця робота написана на HTML, CSS, JS, NODE.JS (структура на рис. 3.9).

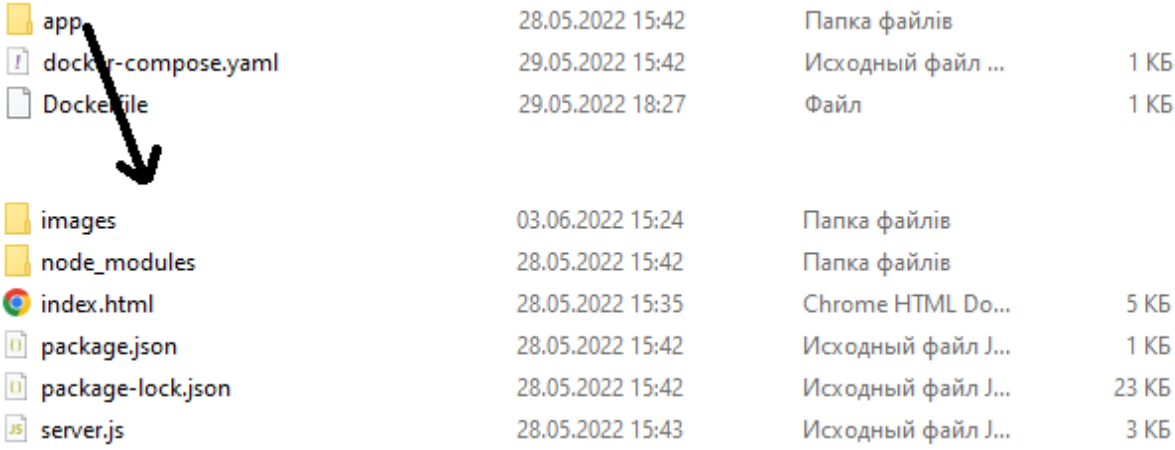

Рис. 3.9. Структурування проекту на локальному комп'ютері

Він складається з однієї сторінки, має css стилі, трохи коду мови програмування JavaScript та свою невелику базу даних, яка генерується. Базою даних є MongoDB – нереляційна база, яка є легкою у застосуванні. Це база даних з відкритим кодом, яка зберігає свої дані у форматі збору, та легко маштабується. Також, використано додаток Mongo-express, щоб можна було легко з'єднуватися з базою даних, та мати гарний графічний інтерфейс, який би дозволяв передивлятися зміни в MongoDB та впливати на них. Щоб з'єднатися з Mongo-express, в коді NodeJS прописується рядок з модулем MongoClient, який потребує посилання на файл коду бази даних MongoDB (рис. 3.10).

```
MongoClient.connect('mongodb://admin:password@localhost:27017', function (err, client) {
```

Рис. 3.10. Дані які потрібні для підключення бази до Mongo-express

Також ці дані будуть встановлені як змінні середовища, коли буде створюватися docker mongodb контейнер.

Запускається сервер через команду *node server.js*. Інтерфейс сайту представлено на рис. 3.11 та рис. 3.12. Сайт запускається на порту 3000.

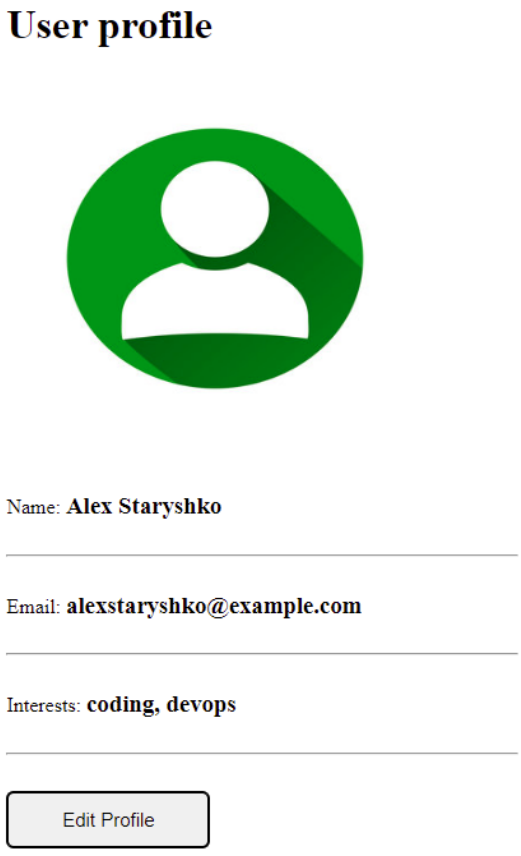

Рис. 3.11. Профіль сайту

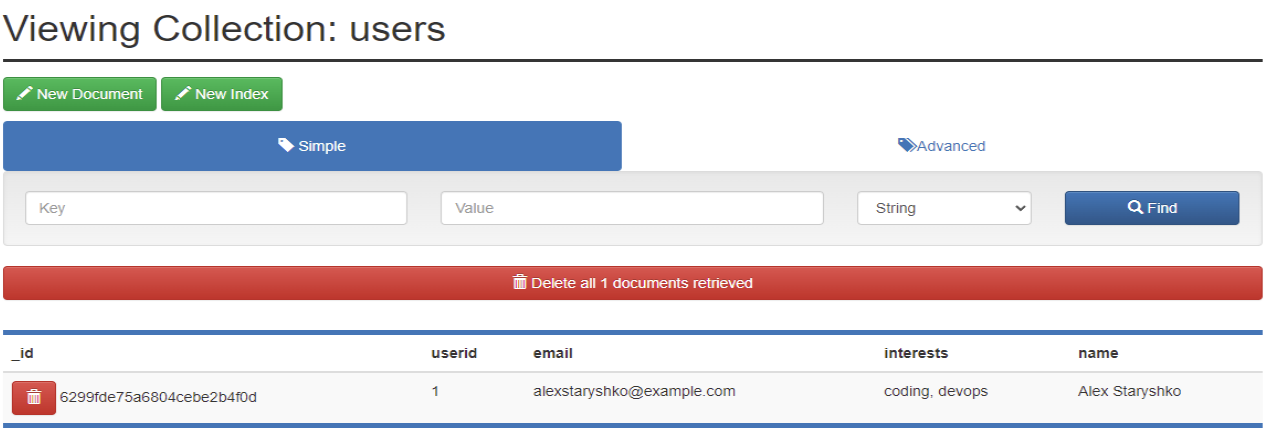

Рис. 3.12. Зв’язок Mongo-express та MongoDB

Після перевірки працездатності переходимо до Docker. Початок йде з того, що Docker за замовченням не існує на операційних системах, тож його треба встановити. Буде продемонстровано процес встановлення Docker на Windows. Перед початком інсталювання треба впевнитися, що операційна система має потрібні налаштування типу: включена віртуалізація, працює Hyper-V, чи підходе версія операційної системи (можливо вона застаріла), тощо. Потім треба скачати інсталятор докеру та запустити його, якщо ж все пройшло добре, то запускаємо докер, бо автоматично на початку він не запуститься.

Переходимо на DockerHub, щоб скачати вже готове зображення (images) бази даних mongo та mongo-express (рис. 3.13).

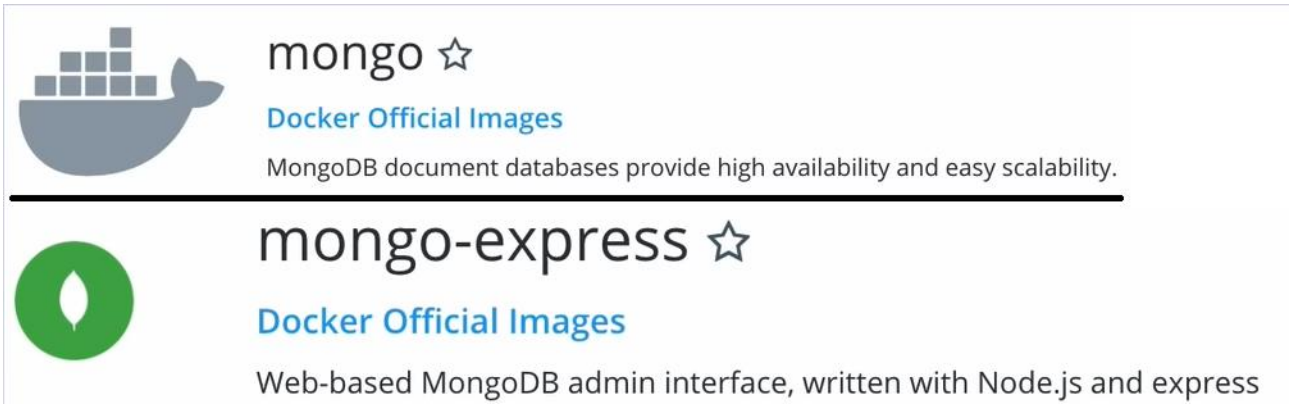

Рис. 3.13. Images of mongo and mongo-expess

Виконуємо команди *docker pull mongo* та *docker pull mongo-express*, щоб завантажити собі ці зображення.

Перед тим, як продовжити далі працювати над кодом контейнерів, треба об’єднати їх в одну мережу. Docker має свою ізольовану мережу в якій працюють

контейнери. Якщо покласти контейнери mongo та mongo-express в одну мережу, то вони зможуть спілкуватися між собою завдяки своїм іменам, не треба додатково використовувати порти та щось подібне. Сервер сайту, який працює на NodeJs знаходиться за мережею докер, тому він зможе спілкуватися з mongo та mongo-express, тільки вказавши свій ip та номер порту. Щоб обійти це, треба html, css та js файл покласти в докер контейнер, в одну мережу з іншими. Наостанок, браузер буде з'єднуватися з мережею через ip та номер порту 3000 (рис. 3.14).

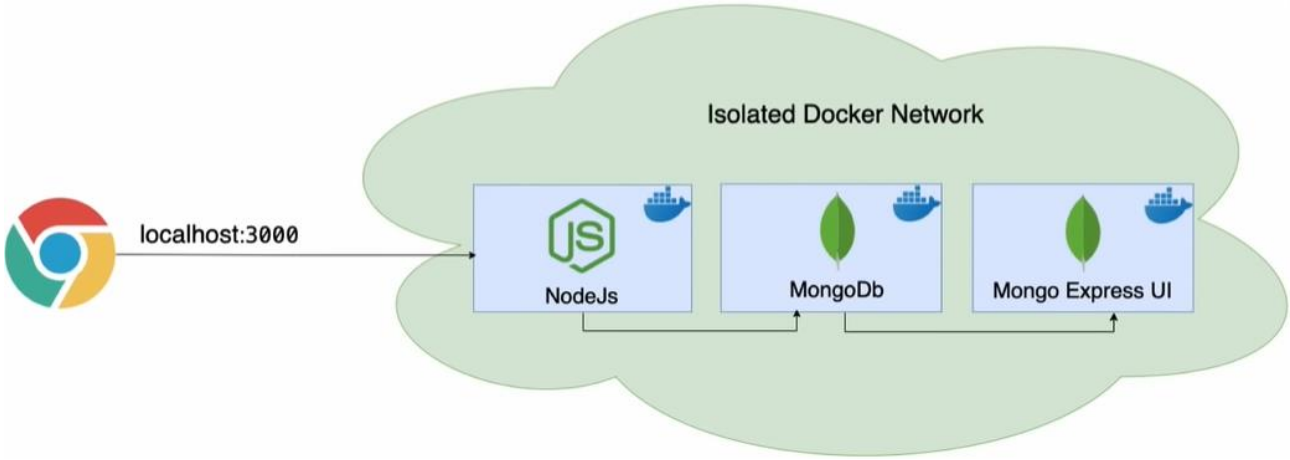

Рис. 3.14. Приклад мережі в Docker

Створюється мережа командою *docker network create*, потім йде назва мережі. Далі створюються контейнери mongo та mongo-express з зображень які були завантажені до цього. При створені в командах прописуються змінні середи, в яких вказується необхідна інформація для запуску та з'єднання двох контейнерів між собою (рис. 3.14 та рис. 3.15)

```
[[simple-js-app]$ docker run -d \
[> -p 27017:27017 \
[> -e MONGO_INITDB_ROOT_USERNAME=admin \
[> -e MONGO_INITDB_ROOT_PASSWORD=password \
[> --name mongodb \
[> --net mongo-network \
[> mongo
```

Рис. 3.14. Команда для створення контейнеру mongo

```
[simple-js-app]$ docker run -d \
> -p 8081:8081 \
> -e ME_CONFIG_MONGODB_ADMINUSERNAME=admin \
> -e ME_CONFIG_MONGODB_ADMINPASSWORD=password \
> --net mongo-network \
> --name mongo-express \
> -e ME_CONFIG_MONGODB_SERVER=mongodb \
> mongo-express
```

Рис. 3.15. Команда для створення контейнеру mongo-express

Для того, щоб не прописувати команди кожного разу, якщо раптом треба буде пере-створити контейнери заново, і не створювати їх окремо, ще й з перезапуском мережі. В такому випадку доцільно використовувати Docker-compose. Він дозволе створювати контейнери запуском одного файлу, а не писати команди кожен раз в командний рядок. Створимо файл mongo.yaml (рис. 3.16).

```yaml
version: '3'
services:
  mongodb:
    image: mongo
    ports:
     - 27017:27017
    environment:
     - MONGO_INITDB_ROOT_USERNAME=admin
     - MONGO_INITDB_ROOT_PASSWORD=password
  mongo-express:
    image: mongo-express
    ports:
     - 8080:8081
    environment:
     - ME_CONFIG_MONGODB_ADMINUSERNAME=admin
     - ME_CONFIG_MONGODB_ADMINPASSWORD=password
     - ME_CONFIG_MONGODB_SERVER=mongodb
```

Рис. 3.16. Mongo.yaml

Тепер якщо запустити команди *node server.js* та *docker-compose -f docker-compose.yaml up*, то можна буде побачити вдалий запуск та з'єднання (рис. 3.17).

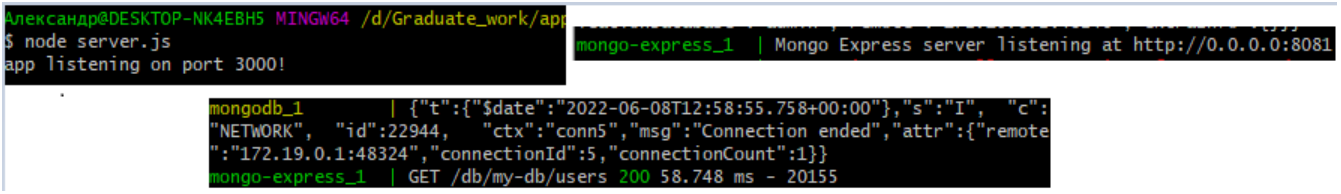

Рис. 3.17. Зв'язок між сайтом та базою даною

Переходимо до коду веб-сайту. Щоб сайт та усі необхідні файли опинилися в докер контейнері, для цього треба буде створити Dockerfile та прописати в ньому конфігураційні команди для створення необхідного середовища з усіма залежностями. Створюємо Dockerfile в корені нашої папки та записуємо в нього код (рис. 3.18).

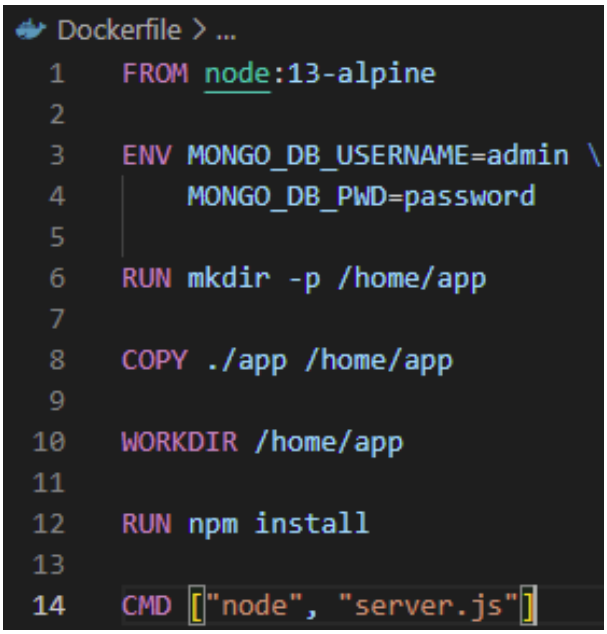

Рис. 3.18. Dockerfile

Якщо прописати команду docker build -t test:app, то Docker з файлу Dockerfile збере images (оболонку), яку можна буде скачати та використовувати.

Далі ініціалізуємо папку командою *git init* та відправляємо код з локальної папки до GitLab репозиторію, користуючись такими командами як: *git commit, git add, git push*. Перевіряємо чи залився код до репозиторію та йдемо далі.

Наступний етап це docker registry та налаштування інфраструктури хмарної платформи через Terraform. Docker registry буде зберігати images контейнерів. Щоб запушити images треба дати йому тег командою *docker tag*

test:app ver-1 та прописати команду відправлення *docker push ver-1*. Важливо, щоб Docker знав куди йому розміщати images і для цього треба його зареєструвати командою *docker login*. Images будуть зберігатися в GCP, там для цього є виділений сервіс (рис. 3.19)

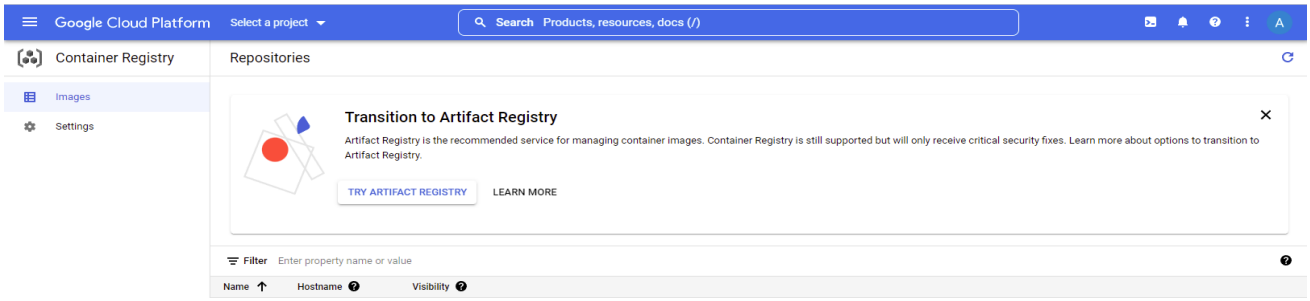

Рис. 3.19. Container Registry в GCP

Щоб через Terraform створити машини, на яких будуть розташовуватися GitLab CI, Runner (для розгортання коду сайту) та веб-сайт, треба прописати конфіг файли. Нижче представлений приклад тестового main.tf який після відпрацювання на GCP створить машину (рис. 3.20).

```
resource "google_compute_instance" "vm_instance" {
  name         = "terraform-instance"
  machine_type = "f1-micro"

  boot_disk {
    initialize_params {
      image = "debian-cloud/debian-9"
    }
  }

  network_interface {
    network = google_compute_network.vpc_network.name
    access_config {
    }
  }
}
```

Рис. 3.20. Тестовий Terraform файл main.tf

Після створення трьох машин, на першу встановлюється GitLab CI через команди bash (рис. 3.21), на другу встановлюється GitLab Runner (рис. 3.22) та через команду *gitlab-runner registry* реєструється docker runner (це для того щоб він міг працювати з командами docker), а на останню машину встановлюється

docker та docker-compose (на цій машині буде працювати сайт). Перевіряємо чи все встановилося та йдемо далі (рис. 3.23, рис. 3.14.).

```
1.sudo apt update

2.sudo apt-get install -y curl openssh-server ca-certificates perl

3.sudo apt-get install -y postfix
Select 'Internet Site' and press enter.

4.curl -sS https://packages.gitlab.com/install/repositories/gitlab/gitlab-ce/script.deb.sh | sudo bash

5.sudo EXTERNAL_URL="http://gitlab.whatproject.local" apt-get install gitlab-ce
gitlab.example.com - your DNS or IP
```

Рис. 3.21. Інсталювання GitLab CI (server)

```
1.sudo apt update

2.sudo apt-get install -y curl openssh-server ca-certificates perl

3.sudo apt-get install -y postfix
Select 'Internet Site' and press enter.

4.curl -sS https://packages.gitlab.com/install/repositories/gitlab/gitlab-ce/script.deb.sh | sudo bash

5.sudo EXTERNAL_URL="http://gitlab.whatproject.local" apt-get install gitlab-ce
gitlab.example.com - your DNS or IP
```

Рис. 3.22. Інсталювання Runner

Рис. 3.23. Вдале інсталювання GitLab CI (server)

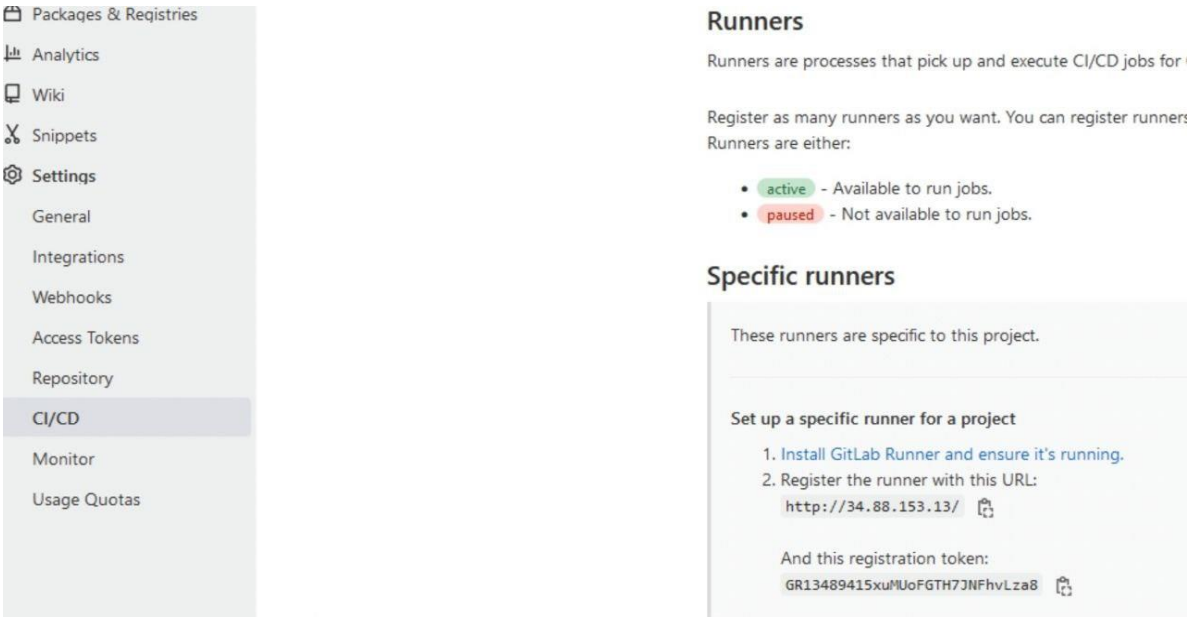

Рис. 3.24. Налаштований Runner

Тепер один з головних моментів, необхідно для GitLab CI створити файл gitlab-ci.yml, щоб прописати в ньому код для розгортання. Важливо слідкувати за відступами, бо файли типу yml дуже строго до цього відносяться. Нижче наведено приклад схожого коду (рис. 3.25).

master .gitlab-ci.yml

```
variables:
  MAVEN_CLI_OPTS: "-s .m2/settings.xml --batch-mode"
  MAVEN_OPTS: "-Dmaven.repo.local=.m2/repository"

cache:
  paths:
    - .m2/repository/
    - target/

build:
  stage: build
  tags:
    - ang_build
  script:
    - echo "Build disk folder with front end files"
    - npm install
    - ls -l
    - ng build
    - ls -l
    - docker build -t $NEXUS_ACC/$DOCKER_REPO:$IMG_TAG .
    - docker tag $NEXUS_ACC/$DOCKER_REPO:$IMG_TAG $NEXUS_IP:8083/$NEXUS_ACC/$DOCKER_REPO:$IMG_TAG
    - docker login -u $DOCKER_ACC -p $DOCKER_PASS http://$NEXUS_IP:8083
    - docker push $NEXUS_IP:8083/$NEXUS_ACC/$DOCKER_REPO:$IMG_TAG
    - echo "Ansible playbook is running now to deploy code on QA server"
    - ansible-playbook deploy.yml
```

Рис. 3.25. Приклад gitlab-ci.yml

Так як автоматизація всього процесу CI/CD за допомогою GitLab CI не потребує складних конфігурацій та багато часу, дуже легко побачити виграш у використанні цієї концепції. Як було сказано у попередніх розділах, процес

розробки ПЗ завдяки автоматизації значно прискорюється, зменшує кількість людського фактору (помилки). CI/CD в демонстративному проекті дозволяє мати клієнтам нову версію додатку кожен раз, як буде випускатися нова версія. Без використання цих підходів, буде тяжко кожного разу вручну оновлювати додаток, тож домогтися тих результатів які давав би CI/CD, було б просто неможливим. Процес розробки виявлення помилок під час написання коду та знаходження несправностей за рахунок автоматизації у декілька разів прискорює процес розробки додатку на всіх етапах життєвого процесу, тож переваги використання CI/CD очевидні.

І нарешті, якщо все вдало відпрацьовує (про це нам скаже gitlab ci) (рис. 3.26. та рис. 3.27).

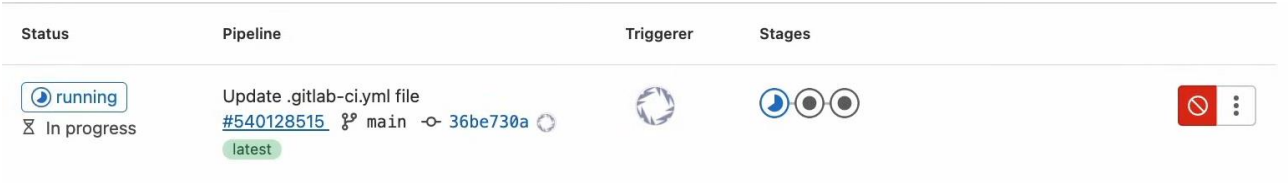

Рис. 3.26. GitLab pipeline

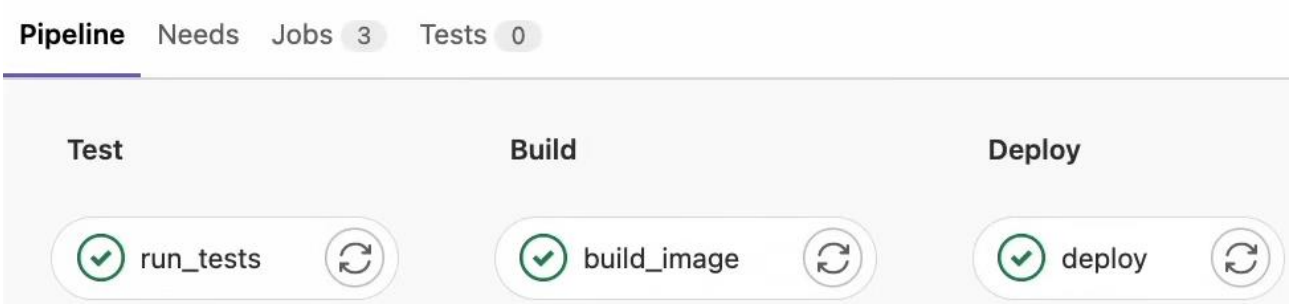

Рис. 2.27. Приклад виконаного GitLab pipeline

На трьох встановлених машинах в GCP, на порту 3000 буде працювати сайт, і якщо прописати ip машини в гуглі та порт 3000, то буде можливість його побачити. Тепер при новому відсиланні коду до GitLab CI, Runner буде автоматично виконувати прописані команди, витягувати новий images з docker registry або ж GCP registry та автоматично розгортувати його в GCP.

CI/CD налаштований! Оскільки йде показ типового CI/CD, зі стратегій continuous deployment використоно звичайний recreate – процес CI/CD просто видаляє стару версію сайту та заміняє її на нову (рис. 2.18). Якщо необхідно

налаштувати моніторинг машин, щоб не заходити вручну кожного разу та перевіряти метрики через файл var/log (рис. 2.28).

```
alexander@DESKTOP-LKRGG9N:/$ cd var/log
alexander@DESKTOP-LKRGG9N:/var/log$ ls
alternatives.log  bootstrap.log  dist-upgrade  faillog  lastlog  ubuntu-advantage.log  upgrade-policy-changed.log
apt               btmp           dpkg.log      journal  private  unattended-upgrades   wtmp
```

Рис. 2.28. Каталог var/log

Можна встановити на одну з машину такі інструменти, як: Splunk або ELK Stack, а на інші машини з яких будуть стягуватися та переглядатися дані (їх логи, або файли) – встановимо forwarders (маленьки програми, які дозволять передавати дані на головний моніторинг сервер). Налаштування Splunk, forwarders, а також приклади робіт з інших проектів наведено на рис. 3.29, рис. 3.30, рис. 3.31, рис. 3.32.

```
1. Download the .deb file;
wget -O splunk-8.2.6-a6felee8894b-linux-2.6-amd64.deb "http

2. Install Splunk;
sudo dpkg -i splunk-8.2.6-a6felee8894b-linux-2.6-amd64.deb

3. Start Splunk and accept license
cd /opt/splunk/bin
sudo ./splunk start --accept-license

4. Enter an administrator username;

5. Enter and confirm a password;

6. Start Splunk at boot;
sudo ./splunk enable boot-start
```

Рис. 3.29. Встановлення Splunk Enterprise

```
wget -O splunkforwarder-8.2.6-a6felee8894b-linux-2.6-amd64.deb "https:

sudo dpkg -i splunkforwarder-8.2.6-a6felee8894b-linux-2.6-amd64.deb

cd /opt/splunkforwarder/bin

sudo ./splunk start --accept-license

sudo ./splunk enable boot-start

sudo ./splunk add forward-server 10.154.0.2:9997

sudo ./splunk add monitor /var/log

cd /opt/splunkforwarder/etc/system/local/
```

Рис. 3.30. Встановлення Splunk forwarders

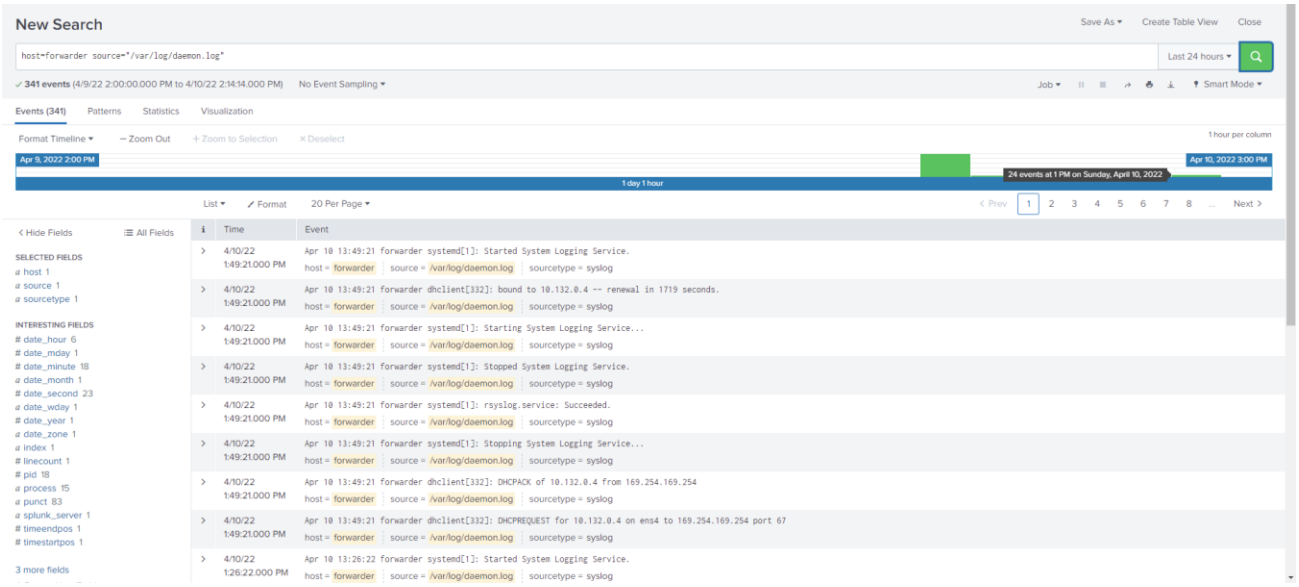

Рис. 3.31. Приклад роботи Splunk Enterprise 1

Рис. 3.32. Приклад роботи Splunk Enterprise 2

3.3 Висновки до третього розділу

У третьому розділі було створено демонстративний CI/CD pipeline, який розгортає типовий веб-сайт зі своєю базою даних. Було продемонстровано нескладний blueprint, який можна використати на простому проекті, щоб впровадити автоматизацію та деякі методології DevOps у розробку. Також продемонстровано, як та яким чином працює безперервна інтеграція, безперервна доставка та безперервне розгортання коду на проекті. Детально

показано якими інструментами та підходами користувалися, щоб створити демонстративний CI/CD pipeline.

Спостерігаючи за налаштованою автоматизацією демонстративного процесу CI/CD, робимо висновки, що такий підхід дає велику перевагу у часі, якщо порівнювати з повністю мануальними проектами. часу, дуже легко побачити виграш у використанні цієї концепції. Як було показано у третьому розділі, процес розробки ПЗ завдяки автоматизації значно прискорюється та зменшується помилок, які б могли статися із-за людського фактору. CI/CD в демонстративному проекті дозволяє мати клієнтам нову версію додатку кожен раз, як буде випускатися нова версія. Без використання цих підходів, буде тяжко кожного разу вручну оновлювати додаток, тож домогтися тих результатів які давав би CI/CD, було б просто неможливим. Процес розробки виявлення помилок під час написання коду та знаходження несправностей за рахунок автоматизації у декілька разів прискорює процес розробки додатку на всіх етапах життєвого процесу, тож переваги використання CI/CD на демонстративному прикладі очевидні.

ВИСНОВКИ

У ході виконання роботи розглянуто такі поняття, як CI/CD та DevOps. Дана розгорнута відповідь наскільки є важливим в наш час автоматизовувати процеси розробки та як це автоматизування пов’язано з підходами CI/CD. Дізналися як DevOps розробник, вміло користуючись концепціями безперервної інтеграції та розгортання, може позитивно впливати на бізнес проекти і на роботу процесу розробки в цілому. Також було розглянуто наскільки важливими є знання інструментів та інших тем для DevOps розробника.

Під час кваліфікаційної роботи переглянуто та розібрано безліч інструментів. Сформовані висновки в яких періодах розробки треба їх використовувати.

Детально описано процес CI/CD в практичній частині, вибрано оптимальні інструменти та стратегії для демонстративного прикладу.

Досліджено позитивний вплив підходів continuous integration, continuous delivery та continuous deployment.

Зауважимо, що який би проект не був: будь-то велика фірма або стартап із декількох людей, на якій би сфері він не базувався (веб, ігри, мобільні додатки та інші), в сучасних реаліях впровадження CI/CD процесів у розробку необхідні. Від впровадження цих підходів очікується як позитивний соціальний вплив, так і економічний. Культура DevOps в компанії сприяє кращому спілкуванню та співпраці команд, оскільки увага розробників буде зосереджена на продуктивності. В економічному плані методології DevOps сприяють позитивному впливу, завдяки налагодженому та якісному процесу оновлення програми, контролю продуктивності через постійний моніторинг у реальному часі, та економічно правильно побудованій інфраструктурі проекту.

Досліджено типові проблеми методологій DevOps. Мінусами стандартних підходів є: швидкі цикли випуску змушують команди постійно бути наготові, технічний борг, велика кількість інструментів, тощо. Запропонували та розібрали нові типи підходів, які повинні допомагати налаштовувати кращий

процес побудови DevOps у команді. Такими підходами є self-serve DevOps та headless. Це нові підходи, користь яких досі досліджується, але вже зараз можна стверджувати їх позитивний вплив на розвиток сфери DevOps в цілому.

ОФОРМЛЕННЯ СПИСКУ ВИКОРИСТАННИХ ДЖЕРЕЛ

1. CI/CD [Електронний ресурс] - Режим доступу: https://en.wikipedia.org/wiki/CI/CD

2. Що таке CI/CD? Розбираємося з безперервною інтеграцією та безперервним постачанням. [Електронний ресурс] - Режим доступу: https://habr.com/ru/company/otus/blog/515078/

3. Безперервна інтеграція та доставка (CI/CD pipeline). [Електронний ресурс] - Режим доступу: https://devops-courses.zone3000.net/ci-cd-pipeline/

4. Що таке DevOps Pipeline? [Електронний ресурс] - Режим доступу: https://dinarys.com/ru/blog/devops-pipeline#contents-4

5. Безперервна інтеграція, постачання або розгортання. [Електронний ресурс] – Режим доступу: https://www.atlassian.com/ru/continuous-delivery/principles/continuous-integration-vs-delivery-vs-deployment

6. Яка різниця між Continuous Delivery, Continuous Deployment та Continuous Integration [Електронний ресурс] – Режим доступу: https://qaat.ru/kakaya-raznica-mezhdu-continuous-delivery-continuous-deployment-i-continuous-integration/

7. Введення в культуру DevOps: про практики та роль DevOps інженера [Електронний ресурс] – Режим доступу: https://dou.ua/lenta/articles/devops-culture-0/

8. Who is a DevOps engineer? [Електронний ресурс] – Режим доступу: https://www.redhat.com/en/topics/devops/devops-engineer

9. DevOps: усунення розбіжностей між розробниками та операторами [Електронний ресурс] – Режим доступу: https://www.atlassian.com/ru/devops

10. Gitflow workflow [Електронний ресурс] - https://www.atlassian.com/git/tutorials/comparing-workflows/gitflow-workflow

11. Git Branching Strategies: GitFlow, Github Flow, Trunk Based [Електронний ресурс] - https://www.abtasty.com/blog/git-branching-strategies/

12. Семантичне Версіонування 2.0.0 [Електронний ресурс] - https://semver.org/lang/uk/

13. Quality gates [Електронний ресурс] - https://docs.sonarsource.com/sonarqube/9.8/user-guide/quality-gates/

14. Інструменти DevOps [Електронний ресурс] – Режим доступу: https://www.atlassian.com/ru/devops/devops-tools

15. Хто такий DevOps та як ним стати: план навчання [Електронний ресурс] – Режим доступу: https://tproger.ru/curriculum/devops/

16. Інструменти керування версіями [Електронний ресурс] – Режим доступу: https://russianblogs.com/article/7402163439/

17. 8 безкоштовних альтернатив Docker на 2022 рік [Електронний ресурс] – Режим доступу: https://highload.today/8-besplatnyh-alternativ-docker-na-2022-god/

18. VM або Docker [Електронний ресурс] – Режим доступу: https://habr.com/ru/post/474068/

19. Docker проти віртуальних машин: відмінності, про які Ви не знали [Електронний ресурс] – Режим доступу: https://andreyex.ru/stati/docker-protiv-virtualnyh-mashin-razlichiya-o-kotoryh-vy-dolzhny-znat/

20. Стратегії деплою в Kubernetes: rolling, recreate, blue/green, canary, dark (A/B-тестування) [Електронний ресурс] – Режим доступу: https://habr.com/ru/company/flant/blog/471620/

21. Проблеми DevOps: чи все добре з методологією? [Електронний ресурс] – Режим доступу: https://blog.iteducenter.ua/articles/devops-is-broken/

22. Ditch toolchain problems with a DevOps platform [Електронний ресурс] – Режим доступу: https://about.gitlab.com/blog/2022/08/24/too-many-toolchains-a-devops-platform-migration-is-the-answer/

23. More and More Enterprises Go Headless [Електронний ресурс] – Режим доступу: https://devops.com/more-and-more-enterprises-go-headless/

Додаток А

ЛІСТИНГ ПРОГРАМИ

UNIT1.cs // файл "index.html"

```
<html lang="en">
<style>
    .container {
        margin: 40px auto;
        width: 80%;
    }
    .button {
        width: 160px;
        height: 45px;
        border-radius: 6px;
        font-size: 15px;
        margin-top: 20px;
    }
    img {
        width: 328px;
        height: 287px;
        display: block;
        margin-bottom: 20px;
    }
    hr {
        width: 400px;
        margin-left: 0;
    }
    h3 {
        display: inline-block;
    }
    #container {
        display: none;
    }
    #container-edit {
        display: none;
    }
    #container-edit input {
        height: 32px;
    }
    #container-edit hr {
        margin: 25px 0;
    }
    #container-edit input {
        width: 195px;
        font-size: 15px;
    }
</style>
<script>
    (async function init() {
        const response = await fetch('http://localhost:3000/get-profile');
        console.log("response", response);
        const user = await response.json();
        console.log(JSON.stringify(user));

document.getElementById('name').textContent = user.name ? user.name :
'Anna Smith';
document.getElementById('email').textContent = user.email ? user.email :
'anna.smith@example.com';
```

```
      document.getElementById('interests').textContent = user.interests ?
user.interests : 'coding';

              const cont = document.getElementById('container');
              cont.style.display = 'block';
          })();

          async function handleUpdateProfileRequest() {
              const contEdit = document.getElementById('container-edit');
              const cont = document.getElementById('container');

              const payload = {
                  name: document.getElementById('input-name').value,
                  email: document.getElementById('input-email').value,
                  interests: document.getElementById('input-interests').value
              };

              const response = await fetch('http://localhost:3000/update-
profile', {
                  method: "POST",
                  headers: {
                    'Accept': 'application/json',
                    'Content-Type': 'application/json'
                  },
                  body: JSON.stringify(payload)
              });
              const jsonResponse = await response.json();

              document.getElementById('name').textContent = jsonResponse.name;
              document.getElementById('email').textContent = jsonResponse.email;
      document.getElementById('interests').textContent= jsonResponse.interests;

              cont.style.display = 'block';
              contEdit.style.display = 'none';
          }

          function updateProfile() {
              const contEdit = document.getElementById('container-edit');
              const cont = document.getElementById('container');

              document.getElementById('input-name').value =
document.getElementById('name').textContent;
              document.getElementById('input-email').value =
document.getElementById('email').textContent;
              document.getElementById('input-interests').value =
document.getElementById('interests').textContent;

              cont.style.display = 'none';
              contEdit.style.display = 'block';
          }
      </script>
      <body>
          <div class='container' id='container'>
              <h1>User profile</h1>
              <img src='profile-picture' alt="user-profile">
              <span>Name: </span><h3 id='name'>Anna Smith</h3>
              <hr />
              <span>Email: </span><h3 id='email'>anna.smith@example.com</h3>
              <hr />
              <span>Interests: </span><h3 id='interests'>coding</h3>
              <hr />
              <button class='button' onclick="updateProfile()">Edit
Profile</button>
          </div>
```

```
          <div class='container' id='container-edit'>
              <h1>User profile</h1>
              <img src='profile-picture' alt="user-profile">
              <span>Name: </span><label for='input-name'></label><input
type="text" id='input-name' value='Anna Smith' />
              <hr />
              <span>Email: </span><label for='input-email'></label><input
type="email" id='input-email' value='anna.smith@example.com' />
              <hr />
              <span>Interests: </span><label for='input-
interests'></label><input type="text" id='input-interests' value='coding' />
              <hr />
              <button class='button'
onclick="handleUpdateProfileRequest()">Update Profile</button>
          </div>
      </body>
      </html>
```

UNIT2.cs // файл "server.js"

```
let express = require('express');
let path = require('path');
let fs = require('fs');
let MongoClient = require('mongodb').MongoClient;
let bodyParser = require('body-parser');
let app = express();

app.use(bodyParser.urlencoded({
  extended: true
}));
app.use(bodyParser.json());

app.get('/', function (req, res) {
    res.sendFile(path.join(__dirname, "index.html"));
  });

app.get('/profile-picture', function (req, res) {
  let img = fs.readFileSync(path.join(__dirname, "images/profile-1.jpg"));
  res.writeHead(200, {'Content-Type': 'image/jpg' });
  res.end(img, 'binary');
});

// use when starting application locally
let mongoUrlLocal = "mongodb://admin:password@localhost:27017";

// use when starting application as docker container
let mongoUrlDocker = "mongodb://admin:password@mongodb";

// pass these options to mongo client connect request to avoid
DeprecationWarning for current Server Discovery and Monitoring engine
let mongoClientOptions = { useNewUrlParser: true, useUnifiedTopology: true
};

// "user-account" in demo with docker. "my-db" in demo with docker-compose
let databaseName = "my-db";

app.post('/update-profile', function (req, res) {
  let userObj = req.body;

  MongoClient.connect(mongoUrlLocal, mongoClientOptions, function (err,
client) {
    if (err) throw err;

    let db = client.db(databaseName);
```

```
        userObj['userid'] = 1;

        let myquery = { userid: 1 };
        let newvalues = { $set: userObj };

        db.collection("users").updateOne(myquery, newvalues, {upsert: true},
function(err, res) {
          if (err) throw err;
          client.close();
        });

      });
      // Send response
      res.send(userObj);
    });

    app.get('/get-profile', function (req, res) {
      let response = {};
      // Connect to the db
      MongoClient.connect(mongoUrlLocal, mongoClientOptions, function (err,
client) {
        if (err) throw err;

        let db = client.db(databaseName);

        let myquery = { userid: 1 };

        db.collection("users").findOne(myquery, function (err, result) {
          if (err) throw err;
          response = result;
          client.close();

          // Send response
          res.send(response ? response : {});
        });
      });
    });

    app.listen(3000, function () {
      console.log("app listening on port 3000!");
    });
```

UNIT3.cs // Dockerfile

```
FROM node:13-alpine

ENV MONGO_DB_USERNAME=admin \
    MONGO_DB_PWD=password

RUN mkdir -p /home/app

COPY ./app /home/app

WORKDIR /home/app

RUN npm install

CMD ["node", "server.js"]
```

UNIT4.cs // файл “docker-compose.yaml”

```
version: '3'
services:
  mongodb:
    image: mongo:4.4.6
    ports:
```

```
      - 27017:27017
    environment:
      - MONGO_INITDB_ROOT_USERNAME=admin
      - MONGO_INITDB_ROOT_PASSWORD=password
  mongo-express:
    image: mongo-express
    restart: always # fixes MongoNetworkError when mongodb is not ready when
mongo-express starts
    ports:
      - 8080:8081
    environment:
      - ME_CONFIG_MONGODB_ADMINUSERNAME=admin
      - ME_CONFIG_MONGODB_ADMINPASSWORD=password
      - ME_CONFIG_MONGODB_SERVER=mongodb
```

UNIT5.cs // файл "main.tf"

```
resource "google_compute_instance" "vm_instance" {
  name         = "try-instance"
  machine_type = "f1-micro"

  boot_disk {
    initialize_params {
      image = "debian-cloud/debian-10"
    }
  }

  network_interface {
    network = google_compute_network.vpc_network.name
    access_config {
    }
  }
}
```

UNIT6.cs // команди для встановлення GitLab Server та GitLab Runner

```
sudo apt-get install -y curl openssh-server ca-certificates perl

sudo apt-get install -y postfix

curl   -sS   https://packages.gitlab.com/install/repositories/gitlab/gitlab-
ce/script.deb.sh | sudo bash

sudo EXTERNAL_URL="http://10.128.0.2" apt-get install gitlab-ce

.

curl   -L   "https://packages.gitlab.com/install/repositories/runner/gitlab-
runner/script.deb.sh" | sudo bash

sudo apt-get install gitlab-runner -y

sudo gitlab-runner register
```

UNIT7.cs // команди для встановлення моніторинг інструменту Splunk та forwarders

```
wget -O splunk-8.2.6-a6fe1ee8894b-linux-2.6-amd64.deb
"https://download.splunk.com/products/splunk/releases/8.2.6/linux/splunk-8.2.6-
a6fe1ee8894b-linux-2.6-amd64.deb"

sudo dpkg -i splunk-8.2.6-a6fe1ee8894b-linux-2.6-amd64.deb

cd /opt/splunk/bin
```

```
sudo ./splunk start --accept-license

sudo ./splunk enable boot-start
.

wget -O splunkforwarder-8.2.6-a6fe1ee8894b-linux-2.6-amd64.deb
"https://download.splunk.com/products/universalforwarder/releases/8.2.6/linux/sp
lunkforwarder-8.2.6-a6fe1ee8894b-linux-2.6-amd64.

sudo dpkg -i splunkforwarder-8.2.6-a6fe1ee8894b-linux-2.6-amd64.deb

cd /opt/splunkforwarder/bin

sudo ./splunk start --accept-license

sudo ./splunk enable boot-start

sudo ./splunk add forward-server 10.128.0.4:9997

sudo ./splunk add monitor /var/log

cd /opt/splunkforwarder/etc/system/local/
```

Додаток Б

ПЕРЕЛІК ДОКУМЕНТІВ НА ОПТИЧНОМУ НОСІЇ

Ім’я файла	**Опис**
Пояснювальні документи	
Диплом_Старишко.doc	Пояснювальна записка роботи. Документ Word.
Диплом_Старишко.pdf	Пояснювальна записка роботи в форматі PDF
Програма	
Program.rar	Архів. Містить коди програми і откомпільовану програму
Презентація	
Презентація_Старишко.ppt	Презентація роботи