МІНІСТЕРСТВО ОБОРОНИ УКРАЇНИ

ВІЙСЬКОВИЙ ІНСТИТУТ ТЕЛЕКОМУНІКАЦІЙ ТА ІНФОРМАТИЗАЦІЇ

ІМЕНІ ГЕРОЇВ КРУТ

*Факультет Інформаційних технологій*

(назва факультету)

*Кафедра Автоматизованих систем управління*

(повна назва кафедри)

нетаємно

(гриф)

**ПОЯСНЮВАЛЬНА ЗАПИСКА**

до кваліфікаційної роботи

*перший (бакалаврський)*

(рівень вищої освіти)

на тему: *Модель використання CI/CD процесів під час розгортання інфраструктури на основі IaC*

Спеціальність: *126 інформаційні системи та технології*

Виконала: *курсант 20 курсу, 205 групи*

*Олена КСЕНДЗУК*

(прізвище та ініціали)

Керівник *Дмитро БАЛАН*

(прізвище та ініціали)

Рецензент  *Володимир ЛЕГКОБИТ*

(прізвище та ініціали)

КИЇВ – 2024

МІНІСТЕРСТВО ОБОРОНИ УКРАЇНИ

ВІЙСЬКОВИЙ ІНСТИТУТ ТЕЛЕКОМУНІКАЦІЙ ТА ІНФОРМАТИЗАЦІЇ

ІМЕНІ ГЕРОЇВ КРУТ

Факультет № 2 *Інформаційних технологій*

Кафедра № 21 *Автоматизованих систем управління*

Рівень вищої освіти *перший (бакалаврський)*

Спеціальність *126 – інформаційні системи та технології*

(шифр і назва)

**ЗАТВЕРДЖУЮ**

**Начальник кафедри № 21**

полковник Олександр ТРОЦЬКО

«\_\_\_» лютого 2024 року

**З А В Д А Н Н Я**

**НА КВАЛІФІКАЦІЙНУ РОБОТУ КУРСАНТУ**

*Ксендзук Олені Олегівні*

(прізвище, ім’я, по батькові)

1. Тема кваліфікаційної роботи: *Модель використання CI/CD процесів під час розгортання інфраструктури на основі IaC*

керівник кваліфікаційної роботи: *Балан Дмитро Дмитрович*

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

затверджені наказом начальника інституту від “05” лютого 2024 року № 20.

2. Строк подання курсантом кваліфікаційної роботи: “30” травня 2024 року.

3. Вихідні дані до кваліфікаційної роботи:

– *Концепція Continuous Delivery/ Continuous Delivery (Deployment);*

*– Infrastructure as a Code.*

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

1*. Аналіз поняття CI/CD та порівняння різних DevOps платформ;*

2. *Поняття та переваги використання інфраструктури як код та аналіз інструментів IaC;*

3. *Аналіз способів інтеграції IaC інструментів в поєднанні з CI/CD та їх практичних аспектів.*

5. Перелік графічного матеріалу (з точним зазначенням обов’язкових креслень):

1. *Пояснювальна записка (60* – *80 ст.);*

2. *Презентація PowerPoint (10* – *15 слайдів).*

6. Консультанти розділів роботи

|  |  |  |  |
| --- | --- | --- | --- |
| Розділ | Прізвище, ініціали та посада  консультанта | Підпис, дата | |
| завдання видав | завдання  прийняв |
| 1 | *п/п-к Кондрусь А.В., старший викладач кафедри № 21* |  |  |
| 2 | *п/п-к Кондрусь А.В., старший викладач кафедри № 21* |  |  |
| 3 | *п/п-к Кондрусь А.В., старший викладач кафедри № 21* |  |  |

7. Дата видачі завдання 05 лютого 2024 року

**КАЛЕНДАРНИЙ ПЛАН**

|  |  |  |  |
| --- | --- | --- | --- |
| №  з/п | Назва етапів кваліфікаційної роботи  (проекту) | Строк виконання етапів роботи (проекту) | Примітка |
| 1. | *Опрацювання необхідної літератури та методичних рекомендацій* | 05.03.2024 |  |
| 2. | *Написання та подання на перевірку 1-го розділу роботи* | 15.04.2024 |  |
| 3. | *Написання та подання на перевірку 2-го розділу роботи* | 24.04.2024 |  |
| 4. | *Написання та подання на перевірку 3-го розділу роботи* | 01.05.2024 |  |
| 5. | *Реалізація практичної частини та подання на перевірку* | 12.05.2024 |  |
| 6. | *Подання кваліфікаційної роботи на перевірку рівня оригінальності* | 19.05.2024 |  |
| 7. | *Оцінювання якості кваліфікаційної роботи (відгук керівника)* | 26.05.2024 |  |
| 8. | *Подання кваліфікаційної роботи на рецензію* | 30.05.2024 |  |

Курсант солдат Олена КСЕНДЗУК

( підпис ) (ім’я та прізвище)

Керівник роботи майор Дмитро БАЛАН

( підпис ) (ім’я та прізвище)

**ВІДГУК**

**керівника на кваліфікаційну роботу**

Курсанта *солдата Ксендзук Олену Олегівну*

(військове звання, прізвище, ім’я, по батькові)

Науковий керівник *викладач кафедри №21*

(посада, науковий ступінь, вчене звання)

*майор Балан Дмитро Дмитрович*

(військове звання, прізвище, ім’я, по батькові)

Викладач кафедри № 21

майор Дмитро БАЛАН

**РЕЦЕНЗІЯ**

**на кваліфікаційну роботу**

Курсанта Ксендзук Олену Олегівну

(прізвище та ініціали)

Рецензент старший викладач кафедри №22

(посада, науковий ступінь, вчене звання)

підполковник Легкобит Володимир Сергійович

(військове звання, прізвище, ім’я, по батькові)

Старший викладач кафедри № 22

підполковник Володимир ЛЕГКОБИТ

# АНОТАЦІЯ

наукової роботи на тему:

„ Модель використання процесів CI/CD під час розгортання інфраструктури на основі IaC”

Актуальність дослідження пов’язана з необхідністю впровадження методів автоматизації для розгортання та управління інфраструктурою у Збройних силах України.

За останні роки інфраструктура в Збройних силах України стала значно складніша та розподілена і тенденція нарощування зберігатиметься та лише зростатиме. Військова діяльність вимагає високої стабільності і надійності інформаційних систем та мереж. Окрім того, під час служби нерідко виникає потреба у швидкому масштабуванні інфраструктури для забезпечення додаткових можливостей. CI/CD та IaC дозволяють автоматизовано масштабувати інфраструктуру з мінімальними затратами.

Процеси CI/CD допомагають автоматизувати тестування якості та забезпечує швидке виявлення та виправлення помилок збірки, що підвищує загальну надійність систем, тим самим забезпечуючи безпеку.

Мета роботи – підвищення ефективності, швидкості та надійності управління та розгортанням інформаційних систем та технологічних рішень, що використовуються в Збройних силах України.

***Ключові слова:*** *DevOps, безперевна інтеграція, безперевна доставка/розгортання, автоматизація, конвеєр, система контролю версій, Git, Jenkins, GitLab, інфраструктура як код, playbook, Ansible, Terraform, контейнеризація, застосунок.*

Зміст

АНОТАЦІЯ 6

Перелік умовних позначень 9

ВCТУП 10

РОЗДІЛ 1 12

Концепція *Continuous* *integration* (CI) та *Continuous* *Delivery* (CD) 12

1.1 Поняття CI/CD 12

1.2. Безперевна інтеграція CI 13

1.3 Безперервна доставка/розгортання CD (*Delivery*/*Deployment*) 15

1.4 Порівняння різних *DevOps* платформ CI/CD 18

1.4.1 *Jenkins* 18

1.4.2 *GitLab* CI/CD 19

1.4.3 *Azure DevOps* 20

1.4.4 *AWS CodePipeline* 21

1.4.5 *CircleCI* 22

1.4.6 *Buildbot* 23

1.4.7 Інші цікаві рішення 25

Висновки до розділу 1 27

Розділ 2 29

Інфраструктура як код (IaC) 29

2.1 Поняття про IaC 29

2.2 Важливість та переваги впровадження IaC в контексті ведення мережево-центричної війни 31

2.3 *Terraform* 33

2.3.1 Провайдери *Terraform* 35

2.3.2 Каталоги та модулі 37

2.3.3 Основні команди *Terraform* 39

2.4 *Ansible* 39

Висновок до розділу 2 41

Розділ 3 Аналіз моделі використання CI/CD під час розгортання інфраструктури на основі IaC 43

3.1 Аналіз способів інтеграції CI/CD з інструментами інфраструктури як коду, такими як *Ansible*, *Terraform* 43

3.2 Практичні аспекти інтеграції CI/CD з IaC інструментами 44

3.2.1 Репозиторій коду та система контролю версій 46

3.2.2 Структура репозиторію та організація коду 48

3.2.3 Розгортання тимчасових середовищ 49

3.2.4 Універсальні вхідні параметри та секрети 52

3.2.5 Тестові набори та перевірка IaC коду 53

3.2.6 Артефакти конфігурацій 56

3.2.7. Схвалення змін та безпечне розгортання 58

3.2.8. Моніторинг та журналювання 61

3.2.9 Зворотні засоби (*rollback*, *rollforward*) 62

3.2.10 Гібридні середовища та різні хмарні провайдери 63

Висновки до розділу 3 65

Висновки 67

Список використаних джерел 68

Додатки 69

Додаток 1 Спадна спіраль 69

Додаток 2 Основи *Lean Thinking* 70

Додаток 3 Досвід команди *Google Web Server* 71

# Перелік умовних позначень

CI/CD – *Continuous Integration / Continuous Delivery, Deployment* (безперевна інтеграція/безперервна доставка)

IaC – *Infrastructure as Code* (інфраструктура як код)

DevOps – *Development and Operation*

API – *Application Programming Interface* (прикладний програмний інтерфейс)

SLA – *Service Level Agreement* (угода про рівень послуг)

HCL – *HashiCorp Configuration Language*

AWS – *Amazon Web Services*

RDS – *Relational Database Service*

GCP – *Google Cloud Platform*

SSL – *Secure Sockets Layer* (криптографічний протокол безпеки)

DNS – *Domain Name System* (система доменних імен)

SSH – *Secure Shell* (мережевий протокол для віддаленого з’єднання та тунелювання TCP-з’єднань)

TCP – *Transmission Control Protocol* (протокол передавання даних у комп’ютерних мережах)

GWS – *Google Web Server*

IRC – *Internet Relay Chat* (технологія багатокористувацьких конференцій в текстовому режимі через мережу Інтернет)

ПЗ – Програмне забезпечення

SaaS – *Software as a service* (програмне забезпечення як послуга)

PaaS – *Platform as a Service* (платформа як послуга)

# ВCТУП

**Актуальність роботи.** Враховуючи стрімкий розвиток технологій та необхідність надійного функціонування інформаційних систем, дослідження цієї теми має важливе значення для подальшого розвитку для застосування у Збройних силах України.

У сучасних умовах безпеки та оборони України, ефективне використання передових технологій та методів стає критично важливим для забезпечення обороноздатності. Ця робота спрямована на дослідження практичного вдосконалення процесів розгортання інфраструктури та забезпечення безпеки інформаційних систем. Впровадження передових методів CI/CD та використання підходів IaC забезпечуватиме швидкість, ефективність і безпеку при впровадженні нових технологій та інформаційних систем.

Впровадження CI/CD процесів під час розгортання інфраструктури на основі IaC полягає у наступних перевагах:

1. Скорочення часу на розгортання та оновлення інфраструктури, підвищення швидкості доставки змін.

2. Підвищення стабільності та надійності інфраструктури за рахунок автоматизованого тестування змін.

3. Зниження ризиків помилок при ручному розгортанні за рахунок автоматизації процесів.

4. Покращення відслідковування змін та можливість повернення до попередніх конфігурацій.

5. Узгодженість конфігурацій в різних середовищах (*dev, staging, prod*).

6. Спрощення процесу управління гібридними та мультихмарними інфраструктурами.

7. Підвищення ефективності командної роботи та співпраці над інфраструктурним кодом.

Впровадження CI/CD процесів під час розгортання інфраструктури на основі IaC полягає у наступних перевагах:

1. Скорочення часу на розгортання та оновлення інфраструктури, підвищення швидкості доставки змін.

2. Підвищення стабільності та надійності інфраструктури за рахунок автоматизованого тестування змін.

3. Зниження ризиків помилок при ручному розгортанні за рахунок автоматизації процесів.

4. Покращення відслідковування змін та можливість повернення до попередніх конфігурацій.

5. Узгодженість конфігурацій в різних середовищах (*dev, staging, prod*).

6. Спрощення процесу управління гібридними та мультихмарними інфраструктурами.

7. Підвищення ефективності командної роботи та співпраці над інфраструктурним кодом.

**Метою дослідження** є опис оптимальної моделі використання процесів *Continuous Integration* (CI) та *Continuous Delivery* (CD) під час розгортання інфраструктури на основі *Infrastructure as Code*  (IaC) з метою підвищення ефективності, швидкості та надійності управління та розгортанням інформаційних систем у Збройних силах України

**Об’єктом дослідження** є процеси розгортання інфраструктури на основі *Infrastructure as Code*  (IaC)

**Предметом дослідження** є модель використання *Continuous Integration* (CI) та *Continuous Delivery* (CD) процесів під час розгортання інфраструктури на основі *Infrastructure as Code*  (IaC).

# РОЗДІЛ 1

# Концепція *Continuous* *integration* (CI) та *Continuous* *Delivery* (CD)

## 1.1 Поняття CI/CD

CI/CD – це неперервна інтеграція та неперервна доставка (або неперервне розгортання – *Continuous Integration/ Continuous Delivery* або *Deployment*), – це практика розробки певного продукту, яка включає такі ключові елементи:

1. Неперервна інтеграція: часте об’єднання змін коду у спільний репозиторій. Після кожної інтеграції автоматично запускаються тести для забезпечення якості коду та виявлення можливих проблем.

2. Неперервна доставка:  на основі неперервної інтеграції автоматизує процес підготовки коду до розгортання, щоб код завжди був готовий до випуску.

3. Неперервне розгортання: робить ще один крок, автоматично розгортаючи зміни коду у продакшн (тобто випуск) після успішного проходження тестів та перевірок якості.

Ці практики дозволяють командам розробників швидше, надійніше та з меншою кількістю помилок доставляти програмне забезпечення. Вони сприяють оптимізованому робочому процесу, що покращує якість коду та прискорює випуск оновлень. Інструменти CI/CD допомагають розробникам швидко збирати, тестувати та розгортати оновлення застосунків через автоматизовані конвеєри. Замість ізольованих ручних процесів, CI/CD зв’язує всі етапи – від коміту до продакшну – в єдиний плавний робочий процес.

*DevOps* зосереджений на швидшій доставці ПЗ за рахунок інтеграції розробки та операцій. Майже у будь-якій IT-компанії існує постійний конфлікт між розробкою та IT-експлуатацією, що створює спадну спіраль *(див додаток 1)* і призводить до постійного збільшення часу, необхідного для випуску нових продуктів або нових функціональностей, зменшення якості тощо, проте насправді вирішення проблем цієї самої низхідної спіралі упираються у принципи *DevOps*.

CI/CD – це важлива практика *DevOps,* яка автоматизує інтеграцію та доставку змін коду. *Continuous Integration* (CI) та *Continuous Delivery/Deployment* (CD) – це практики, які дозволяють автоматизувати процес збирання, тестування та випуску програмного забезпечення. Вони є ключовими елементами підходу *DevOps* та сприяють частішому та більш надійному релізу програмних продуктів.

Ця автоматизація прискорює цикли випуску, покращує якість за рахунок раннього виявлення дефектів та дозволяє надійні розгортання “одним натисканням”. Впроваджуючи CI/CD, команди *DevOps* можуть часто та надійно випускати оновлення.

## 1.2. Безперевна інтеграція CI

Безперервна інтеграція (*Continuous Integration*, CI) є ключовою практикою в процесі розробки програмного забезпечення, що допомагає підвищити якість коду та прискорити цикл релізів (випусків). Розгляньмо на прикладі класичної ситуації, коли розробники об’єднують зміни коду в спільний репозиторій протягом дня. Кожна зміна автоматично збирається і тестується, щоб швидко виявляти проблеми. Такий приклад насправді актуальний не лише для цивільної сфери, як може здаватися на перший погляд, оскільки для Збройних сил України також розробляються різні інформаційні системи, програмні застосунки та інші рішення, котрі після їх релізу (тобто випуску) потребують постійної підтримки, вдосконалення програмного коду, випуск оновлень, та особливою впровадження безпекових рішень для цих продуктів.

Тобто *Continuous Integration* (CI) стосується практики регулярного збирання та тестування коду програми після кожного злиття у центральний репозиторій коду. Це допомагає виявляти конфлікти та помилки на ранніх стадіях циклу розробки. CI здійснюється через автоматизовані збірки та запуск тестових наборів, що дозволяє постійно перевіряти цілісність системи.

Отже, інтеграція це та частина роботи команди розробників котра стосується до збірки готового продукту (перед випуском у продакшн). Концепція CI полягає в регулярному (наприклад, після кожного коміту, тобто застосованих змін, до системи контролю версій) збиранні та автоматичному тестуванні всього проекту для раннього виявлення та виправлення помилок інтеграції. Основні кроки в процесі безперервної інтеграції:

1. Збирання коду (*compile/build*) програмний код збирається в деяку виконувану форму (двійкові файли, артефакти тощо).

2. Запуск юніт-тестів - виконуються автоматизовані тести низького рівня, що перевіряють окремі модулі системи.

3. Аналіз якості коду - статичний аналізатор коду перевіряє код на дотримання стандартів, відсутність вразливостей та антипатернів.

4. Збирання артефакту - створюється пакунок або образ, придатний для розгортання.

5. Запуск інтеграційних/функціональних тестів - більш комплексні тести перевіряють взаємодію компонентів системи.

6. Розгортання в тестове середовище - збірка, що пройшла всі перевірки, розгортається для додаткових ручних або автоматизованих тестів.

Усі ці моменти більш детально описуються у розділі 3.

Весь цей процес ініціюється автоматично при новому коміті і зазвичай виконується на окремому віртуальному або фізичному сервері CI. Популярні рішення: *Jenkins, GitLab CI/CD, Azure DevOps, CircleCI* та інші.

Основні переваги безперервної інтеграції: раннє виявлення та виправлення дефектів, підвищення якості програмного забезпечення, зменшення ризиків і витрат на інтеграцію, можливість частіших релізів, гнучкість та економія часу за рахунок автоматизації.

CI є невід'ємною практикою сучасних гнучких методологій розробки програмного забезпечення, таких як *Agile, DevOps* тощо.

## 1.3 Безперервна доставка/розгортання CD (*Delivery*/*Deployment*)

Безперервна доставка (*Continuous Delivery*):

CD (*Delivery*) розширює концепцію CI, автоматизуючи процес релізу програмного забезпечення за допомогою безперервного розгортання до тестових або проміжних середовищ. Основні кроки CD (*Delivery*):

1. Виконання процесу CI (збирання, тестування коду).

2. Створення пакунку або образу для розгортання.

3. Розгортання збірки у тестове або проміжне середовище.

4. Запуск додаткових тестів (навантажувальне тестування, тести користувацького інтерфейсу тощо).

5. Потенційне ручне схвалення для розгортання у виробниче середовище.

Ключова відмінність від CI - можливість легко вручну або автоматично перенести успішну збірку виробничого середовища після додаткових перевірок.

Безперервне розгортання (*Continuous Deployment*):

CD (*Deployment*) є менш обмежувальною версією безперервної доставки. Тут кожна збірка, що успішно пройшла через конвеєр CI/CD, автоматично розгортається у виробниче середовище без ручного схвалення. Це забезпечує максимальну швидкість релізів. Однак, як зазначають DevOps інженери, при такому підході потрібно «тримати руку на серці» іноді.

Конвеєр (*pipeline*) у контексті безперервної інтеграції та безперервної доставки - це автоматизована послідовність кроків, які виконуються з кодом з моменту внесення змін до моменту розгортання у виробниче середовище.

Конвеєр CI/CD описує весь робочий процес створення та релізу програмного забезпечення і може включати такі основні етапи:

1. Ініціація конвеєра - коміт коду в репозиторій або інша подія, що запускає конвеєр.

2. Збирання коду (*build*) - компіляція або пакування коду в придатну для розгортання форму.

3. Юніт-тестування - запуск автоматизованих модульних тестів.

4. Аналіз якості коду - статичний аналіз, перевірка стандартів, безпеки ін.

5. Створення артефактів - створення пакунків, образів для розгортання.

6. Тестування безпеки - сканування на вразливості, перевірки відповідності політиці безпеки.

7. Інші типи тестування - інтеграційне, регресійне, навантажувальне тощо.

8. Розгортання у тестове середовище - для подальших тестів та перевірок.

9. Затвердження для релізу - ручне або автоматичне схвалення переходу до продакшн.

10. Розгортання у виробниче середовище.

11. Моніторинг - спостереження за продуктивністю та станом розгорнутого застосунку.

Конвеєри зазвичай конфігуруються як код в спеціальних файлах із визначенням усіх етапів, середовищ, інструментів. Популярні інструменти: *Jenkins, GitLab CI/CD, Azure Pipelines, AWS CodePipeline* тощо.

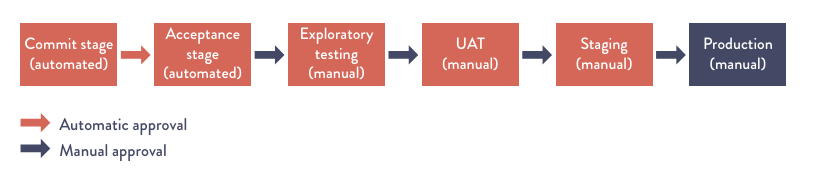
Конвеєр розгортання після будь-якої зміни перевіряє, що код успішно інтегрується у середовище, близьке до виробничого.

Рис. 1.3.1 Конвеєр розгортання

Ключові переваги конвеєрів: автоматизація процесу доставки, виявлення помилок на ранніх стадіях, підвищення швидкості та надійності релізів, відтворюваність процесу, прозорість та аудит процесу, скорочення витрат на розгортання.

Отже, ключові відмінності CD (*Delivery*) від *Continuous Deployment*: при CD (*Delivery*) потрібне ручне схвалення для розгортання у виробниче середовище, при *Continuous* *Deployment* розгортання у виробниче середовище відбувається автоматично.

Обидві практики CD і *Continuous Deployment* мають на меті прискорити цикл релізів, але CD додає додатковий контрольний захід безпеки у вигляді ручного затвердження перед виробничим розгортанням.

Ключові переваги CD (*Delivery*) та *Continuous Deployment*:

- швидший цикл релізів та доставка;

- зниження ризиків через автоматизацію та безперервне тестування;

- зменшення витрат на релізи програмного забезпечення;

- можливість невеликих, але частих випусків замість великих релізів.

Обидві практики є складовими *DevOps* підходу і повинні впроваджуватися разом з відповідними культурними та організаційними змінами.

У випадку створення і подальшого підтримання нового інформаційного продукту ми опиняємося в ситуації, коли розгортання займає кілька місяців. Особливо якщо це дуже складна структура, що використовує тісно пов’язані один з одним монолітні застосунки, котрі зазвичай погано інтегровані у середовище для тестування, мають значну тривалість тестування і тривалий час розгортання в робочому середовищі, високу залежність від тестування вручну і необхідність схвалення численними інстанціями в компанії.

За наявності великої тривалості часу розгортання практично на кожній стадії потоку створення цінності потрібні надзусилля. Може статися і так, що на завершальному етапі проєкту, коли всі результати праці інженерів буде зібрано докупи, все перестане працювати, код не буде збиратися правильно або не витримуватиме тестування. Виправлення проблеми разом із виявленням того, хто саме “заламав” код і як це виправити, вимагатиме кількох днів або навіть тижнів, а в результаті віддача для клієнтів виявиться невисокою.

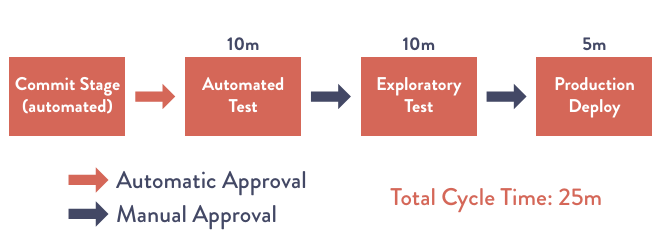
В ідеальному випадку при роботі з *DevOps* розробники швидко і регулярно отримують зворотний зв’язок. Це надає їм можливість швидко й незалежно впроваджувати, інтегрувати і валідувати код, а також забезпечувати його розгортання у виробничому середовищі.

Ключовий компонент для створення швидкого та рівномірного потоку – виконання завдань невеликими порціями. Концепція ощадливого виробництва доводить те, що вона ефективніша за традиційну стратегію «великих партій» *(додаток 2)*.

Це досягається шляхом постійної перевірки змін у коді, зроблених у репозиторій системи контролю версій, автоматичного і передвиробничого тестування змін, а потім розгортання у реальному виробничому середовищі. Що дозволяє нам бути твердо впевненим: зроблені зміни після розгортання будуть функціонувати так, як задумано, і будь-яку проблему, що виникне, буде швидко виявлено і виправлено.

Цього найпростіше досягти, коли архітектура є модульною, добре інкапсульованою й у ній відсутні тісні зв’язки між компонентами. Завдяки цьому невеликі групи розробників отримують можливість працювати з високим ступенем автономності, а можливі збої зазвичай мають обмежені наслідки, що не викликають глобальних порушень працездатності системи.

За такого сценарію час розгортання вимірюється хвилинами або, у гіршому разі, годинами, а карта потоку цінності має виглядати приблизно так, як показано на рисунку нижче.

Рис 1.3.2 Технологічний потік створення цінності з розгортання за хвилини

## 1.4 Порівняння різних *DevOps* платформ CI/CD

## 1.4.1 *Jenkins*

Розглянемо деякі з найпопулярніших систем CI/CD:

*Jenkins* – це одна з найстаріших і найпопулярніших систем CI/CD з відкритим кодом, підтримує величезну кількість плагінів для інтеграції з різними інструментами та середовищами. Це сервер із відкритим кодом на основі *Java*, який виконує безперервний процес інтеграції. Конфігурування може відбуватися через вебінтерфейс або за допомогою кодування конвеєрів.

Переваги *Jenkins*:

- Велика і активна спільнота користувачів та розробників.

- Гнучкість завдяки підтримці величезної кількості плагінів.

- Безкоштовний інструмент з відкритим кодом.

- Простий у встановленні та використанні.

- Кросплатформеність, можливість розгортання на різних операційних системах.

Недоліки *Jenkins*:

- Вебінтерфейс може бути не завжди зручним та інтуїтивно зрозумілим.

- Вимагає додаткового налаштування та обслуговування, особливо при використанні великої кількості плагінів.

- Може мати проблеми з продуктивністю при обробці великої кількості одночасних збірок або великих журналів.

- Масштабування може бути складним процесом у складних середовищах.

- Обмежена гнучкість порівняно з деякими альтернативами, такими як *Buildbot*.

Загалом, *Jenkins* залишається одним із найпопулярніших і потужних інструментів для безперервної інтеграції та безперервного розгортання, пропонуючи багату екосистему плагінів, простоту встановлення та активну спільноту користувачів.

## 1.4.2 *GitLab* CI/CD

*GitLab* CI/CD – позиціонує себе як повноцінна *DevOps* платформа, це вбудована система CI/CD від *GitLab* для інтеграції з їх хмарною або локальною *Git*-системою. Її особливості: конфігурація через YAML-файли в репозиторії, має підтримку контейнерів *Docker* і *Kubernetes*.

Переваги *GitLab* CI/CD:

- Тісна інтеграція з *GitLab*, що забезпечує безшовний робочий процес від коду до розгортання.

- Простота налаштування завдяки використанню деклараційних YAML-файлів у репозиторіях.

- Візуалізація конвеєрів у вбудованому графічному інтерфейсі *GitLab*.

- Розширена підтримка *Docker* і *Kubernetes* для контейнеризації та оркестрації.

- Безкоштовний для відкритих та внутрішніх проектів.

Недоліки *GitLab* CI/CD:

- Менша гнучкість та розширюваність порівняно з *Jenkins* через відсутність великої екосистеми плагінів.

- Обмежена можливість налаштування складних сценаріїв CI/CD через деклараційний підхід YAML-файлів.

- Може мати обмеження продуктивності при обробці великої кількості паралельних конвеєрів або проектів.

- Вимагає використання *GitLab* як системи управління вихідним кодом, що може бути перешкодою для деяких команд.

Загалом, *GitLab* CI/CD є потужною та зручною вбудованою платформою CI/CD, яка чудово інтегрується з *GitLab* та підтримує найсучасніші технології, такі як *Docker* та *Kubernetes*. Однак, її можливості можуть бути обмеженими порівняно з більш гнучкими та розширюваними інструментами CI/CD, такими як *Jenkins* або *Buildbot*, для складних сценаріїв чи вимог до налаштування.

## 1.4.3 *Azure DevOps*

*Azure DevOps* - рішення від *Microsoft* для CI/CD та управління життєвим циклом додатків, включає *Git*-репозиторії, системи будівництва та релізів, тестові інструменти. *Azure DevOps* є хмарною службою, але також можливий локальний сервер.

Переваги *Azure DevOps*:

- Тісна інтеграція з екосистемою *Microsoft* та хмарною платформою *Azure*.

- Хмарний сервіс, що забезпечує легке масштабування та відсутність необхідності в обслуговуванні інфраструктури.

- Безкоштовний для невеликих команд (до 5 користувачів) з необмеженою кількістю приватних репозиторіїв.

- Вбудована підтримка різних мов програмування та фреймворків *Microsoft*.

- Інтеграція з іншими інструментами Microsoft, такими як *Visual Studio* та *Office*.

Недоліки *Azure DevOps*:

- Не є повністю відкритим джерельним програмним забезпеченням, що може бути проблемою для деяких рішень.

- Деякі розширені функції, такі як паралельні задачі та додаткові хмарні ресурси, є платними.

- Обмежена гнучкість порівняно з деякими відкритими інструментами CI/CD, такими як *Jenkins* або *GitLab* CI/CD.

- Тісний зв'язок з екосистемою *Microsoft* може бути перешкодою для команд, які використовують інші технології.

Загалом, *Azure DevOps* є потужним і зручним рішенням для CI/CD та управління життєвим циклом додатків, особливо для команд, які вже використовують інструменти та хмарну платформу *Microsoft*. Однак його залежність від екосистеми *Microsoft* та наявність деяких платних функцій може бути недоліком для деяких організацій, які шукають більш відкриті та гнучкі альтернативи.

## 1.4.4 *AWS CodePipeline*

AWS *CodePipeline* - сервіс безперервної доставки від AWS для автоматизації релізів, інтегрується з іншими сервісами AWS (*CodeBuild, CodeDeploy, ECR, ECS* тощо), налаштування конвеєрів через веб-консоль або AWS CLI.

Переваги *AWS CodePipeline*:

- Використання переваг екосистеми AWS, тісна інтеграція з іншими сервісами для повного циклу CI/CD.

- Масштабованість і відмовостійкість завдяки хмарній природі сервісу.

- Автоматичні оновлення та усунення помилок безпеки AWS.

- Візуалізація конвеєрів та простий моніторинг їх статусу.

- Підтримка різних джерел коду, таких як AWS *CodeCommit, GitHub, Bitbucket* тощо.

Недоліки AWS *CodePipeline*:

- Не має вбудованих інструментів для збирання та тестування коду, для цього потрібно використовувати додаткові сервіси AWS, такі як *CodeBuild*.

- Платна послуга з урахуванням вартості інших задіяних сервісів AWS.

- Може бути менш гнучким порівняно з відкритими інструментами CI/CD для специфічних потреб.

- Залежність від екосистеми AWS, що може бути проблемою для команд, які використовують інші хмарні платформи або локальні рішення.

Загалом, AWS *CodePipeline* є зручним і масштабованим сервісом безперервної доставки для команд, які вже використовують екосистему AWS. Він забезпечує тісну інтеграцію з іншими сервісами AWS, візуалізацію конвеєрів та автоматичні оновлення. Однак його платні послуги, відсутність вбудованих інструментів збирання/тестування та залежність від AWS можуть бути недоліками для деяких організацій.

## 1.4.5 *CircleCI*

*CircleCI* - хмарна платформа CI/CD з акцентом на швидкість та продуктивність, підтримує *Docker*, інтеграцію з хмарними провайдерами, її конфігурація конвеєрів через YAML-файли.

Переваги *CircleCI*:

- Проста у налаштуванні завдяки використанню декларативних YAML-файлів для визначення конвеєрів.

- Масштабована архітектура, яка дозволяє легко розширювати потужності за потребою.

- Гнучка підтримка різних мов програмування, фреймворків та інструментів.

- Паралельне виконання завдань для прискорення процесів збирання та тестування.

- Інтеграція з популярними хмарними провайдерами, такими як AWS, GCP та *Azure*.

Недоліки *CircleCI*:

- Повністю хмарний сервіс без можливості локального розгортання.

- Платні тарифні плани, які можуть бути дорогими для великих команд або проектів.

- Обмежена гнучкість порівняно з самообслуговуваними інструментами CI/CD для складних сценаріїв.

- Залежність від зовнішнього постачальника послуг, що може створювати проблеми з безпекою та конфіденційністю для деяких організацій.

Загалом, *CircleCI* є потужною та швидкою хмарною платформою CI/CD, яка пропонує просте налаштування, масштабованість та інтеграцію з популярними хмарними провайдерами. Вона є чудовим вибором для команд, які шукають зручне рішення CI/CD з акцентом на швидкість та продуктивність. Однак її повністю хмарна архітектура та платні тарифи можуть бути недоліками для деяких організацій з обмеженими бюджетами або особливими вимогами до безпеки та конфіденційності.

## 1.4.6 *Buildbot*

*Buildbot* – є потужним інструментом для автоматизації збірки, тестування та розгортання програмного забезпечення*. Buildbot* – це платформа CI на основі *Python*, яка допомагає автоматизувати цикли компіляції та тестування, щоб перевіряти зміни коду, а потім автоматично створювати та тестувати після кожної зміни. Ця програма допомагає розгортати програму та керувати складним процесом випуску програмного забезпечення.

Переваги: підтримує широкий спектр систем збірки, таких як *Make, Ant, Maven, Gradle,* і може бути інтегрований з різними системами управління версіями, такими як *Git, Subversion, Mercurial, Buildbot* дозволяє легко масштабувати процеси збірки шляхом розподілу навантаження між декількома віддаленими робітниками (*workers*); завдяки можливості налаштування за допомогою *Python, Buildbot* надзвичайно гнучкий і може бути адаптований до різноманітних потреб та робочих процесів, має зручний вебінтерфейс, який дозволяє відстежувати стан збірок, переглядати журнали та керувати конфігурацією, *Buildbot* може надсилати повідомлення про стан збірок через різні канали, такі як електронна пошта, IRC, *Slack* тощо.

Недоліки: хоча *Buildbot* є гнучким, його налаштування може бути складним, особливо для новачків, оскільки воно вимагає знання *Python* та специфічної конфігурації *Buildbot;* засвоєння Buildbot може бути складним через його складність і відсутність всеосяжної документації; у деяких випадках Buildbot може мати проблеми з продуктивністю, особливо при обробці великої кількості збірок або працюючи з великими журналами; Оскільки *Buildbot* написаний на *Python*, він залежить від наявності правильно налаштованого середовища *Python* на кожному вузлі, де він працює.

Загалом, Buildbot є потужним інструментом для автоматизації збірки та розгортання, але він може мати складну крива освоєння та вимагати значних зусиль для налаштування та підтримки. Вибір Buildbot або альтернативного інструменту залежить від конкретних вимог проекту, наявних ресурсів та досвіду команди.

## 1.4.7 Інші цікаві рішення

*Nevercode* - це хмарна платформа CI/CD, призначена для тестування та розгортання програмних проектів. Вона інтегрується з *GitHub* і дозволяє автоматизувати будь-який конвеєр безперервної доставки. *Nevercode* також підтримує тестування та розгортання застосунків на основі *Docker*.

*Semaphore* - також хмарне рішення CI/CD для тестування та розгортання програмних проектів. Воно інтегроване з *GitHub* і здатне автоматизувати будь-який конвеєр безперервної доставки, включаючи підтримку тестування та розгортання застосунків, що базуються на *Docker*.

*GoCD* - це інструмент з відкритим кодом для створення та випуску програмного забезпечення, який підтримує сучасну інфраструктуру CI/CD. *GoCD* пропонує просте налаштування для швидкого зворотного зв'язку та розгортання за вимогою. Він забезпечує контроль над повним робочим процесом і відстежує зміни від комітів до розгортання.

*Wercker* - це хмарна платформа CI/CD, яка є чудовим вибором для проектів, орієнтованих на *Docker*. *Wercker* забезпечує інтеграцію з *Git*, включаючи *GitHub*, *Bitbucket* та *GitLab*, а також контроль версій. Він також пропонує можливість імітувати середовище *SaaS* для налагодження та тестування конвеєрів перед їх остаточним розгортанням.

*Codeship* - це хмарне рішення, яке сприяє ранньому та автоматизованому випуску програмного забезпечення. Воно допомагає розробникам швидше створювати кращі продукти, оптимізуючи процеси тестування та релізів. *Codeship* інтегрується з будь-якими інструментами, сервісами та хмарними середовищами.

*Travis* CI - це інструмент безперервної інтеграції, який допомагає командам розробників створювати та тестувати проєкти. *Travis* CI автоматично виявляє нові коміти у *GitHub*, збирає проєкти та запускає тести при кожному коміті. Він простий у налаштуванні та має вбудовану підтримку баз даних.

*TeamCity* - це інструмент CI/CD з відкритим кодом від *JetBrains*. Заснований на *Java, TeamCity* добре інтегрується з проєктами на базі *Docker* та *Kubernetes*. Він забезпечує гнучкий робочий процес, придатний для різних методологій розробки. *TeamCity* відрізняється простотою встановлення, налаштування та використання.

*Bamboo* - це інструмент безперервної інтеграції від *Atlassian*, який автоматизує керування розгортанням застосунків та створення конвеєрів безперервного розгортання. *Bamboo* автоматично виявляє нові гілки в *Git, Mercurial* та SVN-репозиторіях і застосовує до них практики CI без ручних команд. Він також здатний створювати образи та відправляти їх до реєстру контейнерів.

*Spinnaker* - багатохмарна платформа безперервного розгортання з відкритим вихідним кодом, яка підтримує релізи та розгортання в різних хмарних провайдерів. *Spinnaker* створює конвеєри для розгортання, що виконують інтеграцію, тестування системи та контролюють процес розгортання. Він також запускає конвеєри через події *Git, Jenkins* тощо.

*BuildKite* - швидка, безпечна і масштабована платформа CI/CD, яка допомагає створювати та тестувати конвеєри для процесів безперервної інтеграції. Якщо потрібно налаштувати веб-сайт відповідно до власної інфраструктури, *BuildKite* може стати чудовим рішенням.

*Buddy* - сервер безперервної інтеграції та безперервного розгортання, який допомагає створювати, тестувати та розгортати веб-сайти разом з кодом через *GitHub, Bitbucket* та *GitLab*. Налаштування образів *Docker* на *Buddy* є простим, а система автоматичного виявлення змін полегшує роботу.

*CodeFresh* - це платформа CI/CD від компанії *Argo*, яку можна використовувати для розгортання додатків на *Azure* та AWS. Побудована на *Kubernetes*, вона має вбудовані бібліотеки, що допомагають швидко додавати та видаляти потрібні функції та аспекти.

*Tekton* - інтуїтивна платформа з відкритим вихідним кодом, яка забезпечує гнучкий і потужний робочий процес для систем CI/CD. *Tekton* дозволяє створювати, тестувати та розгортати коди на будь-яких хмарних серверах та системах. Він вважається одним з найкращих інструментів для користувачів *Kubernetes.*

Загалом, кожна система має свої сильні та слабкі сторони. Вибір залежить від потреб проєкту, наявного досвіду команди, вимог до масштабованості, бюджету та інтеграції з іншими інструментами *DevOps.*

## Висновки до розділу 1

CI та CD є ключовими практиками *DevOps*, які дозволяють автоматизувати процес збирання, тестування та доставки програмного забезпечення, забезпечуючи більш часті та надійні релізи.

Безперервна інтеграція зосереджена на регулярному збиранні та тестуванні коду після кожного коміту. Це допомагає раннє виявляти та виправляти помилки та конфлікти на ранніх етапах циклу розробки.

Безперервна доставка розширює концепцію CI, автоматизуючи процес релізу програмного забезпечення шляхом розгортання успішних збірок у тестові або проміжні середовища.

Безперервне розгортання є крайньою формою CD, де кожна успішна збірка автоматично розгортається у виробниче середовище без ручного схвалення.

Ключовими перевагами CI/CD є: прискорення циклу релізів, раннє виявлення дефектів, підвищення якості ПЗ, зниження ризиків та витрат на розгортання.

Процеси CI/CD реалізуються через конвеєри (*pipeline*) - автоматизовані послідовності етапів від початкового коміту до фінального розгортання.

Існує низка популярних інструментів CI/CD, таких як *Jenkins, GitLab CI/CD, Azure DevOps, AWS CodePipeline* тощо. Вибір системи CI/CD залежить від потреб проекту, масштабованості, інтеграції з іншими інструментами, наявного досвіду команди та бюджету. Успішне впровадження CI/CD вимагає не лише технічних змін, а й культурних та організаційних трансформацій згідно з принципами *DevOps*. CI/CD підходи є невід'ємною складовою сучасного гнучкого процесу розробки програмного забезпечення і допомагають прискорити доставку цінності кінцевим користувачам.

Отже, безперервна інтеграція та безперервна доставка/розгортання є потужними практиками *DevOps*, які забезпечують ефективний та якісний цикл релізів програмного забезпечення при належному впровадженні.

# Розділ 2

# Інфраструктура як код (IaC)

## 2.1 Поняття про IaC

Інфраструктура як код (*Infrastructure as Code*  - IaC) - це парадигма, в якій вся інфраструктура ІТ-систем (обчислювальні ресурси, мережі, сховища даних тощо) визначається як версійований машинозчитуваний код замість ручного налаштування ресурсів. Ця інфраструктурна конфігурація може бути відтворена в будь-якому середовищі з коду за допомогою автоматизованих інструментів.

Інфраструктура — це ресурси, які потрібні для підтримки коду. Водночас дехто може уявити серверні стійки, комутатори та купу заплутаних кабелів... Але це вчорашній день. Сьогодні 99% проєктів живе в «хмарах». Тобто ресурси — це віртуальні машини, контейнери, *load balancers*.

Отже, усі хмарні ресурси — це інше програмне забезпечення, яке виконується на комп’ютерах нашого хмарного провайдера.

Утім, побутує міф про небезпеку використання хмарних сервісів для інформаційних систем, пов’язаних із сектором оборони, проте справжній досвід виразно показав, що такі побоювання являються марними. Як приклад можна навести досвід використання системи ситуаційної обізнаності *Delta.*

Окрім того, у сучасній боротьбі з противником, котрий має значну чисельну перевагу в традиційних озброєннях, нам вкрай необхідно мати перевагу у веденні мережоцентричної війни.

Отже, *Infrastructure as Code*  — це спосіб постачання та керування обчислювальними та мережевими ресурсами методом їх опису у вигляді програмного коду, на відміну від налаштовування необхідного обладнання власноруч чи з допомогою інтерактивних інструментів.

*Infrastructure as Code*  є (вже не таким) новим трендом, який розв’язує актуальну проблему автоматизації інфраструктури.

У багатьох могла трапитися подібна ситуація із наступним діалогом:

— Мені треба задеплоїти лоад-балансер...  
— Вибач, у нас завал! Будь ласка, створи квиток у JIRA і повертайся пізніше.

Якби інфраструктура була автоматизована, цього діалогу б не відбулося (як і затримки в роботі), бо лоад-балансер автоматично деплоївся б. Тому автоматизація інфраструктури така популярна. Вона розв’язує не тільки технічні питання, а й організаційні та комунікаційні. Автоматизація полегшує наше життя і перетворює безлад на передбачуваний процес.

Основні принципи IaC:

- уся інфраструктура описується декларативним способом, як код;

- код інфраструктури зберігається у системі контролю версій;

- налаштування інфраструктури за допомогою інструментів автоматизації;

- ідемпотентність - неодноразове застосування визначення завжди призводить до одного й того ж результату.

Ключові переваги впровадження IaC:

1. Відтворюваність - можливість швидко й послідовно відтворити ідентичні середовища з коду.
2. Масштабованість - легко створювати та керувати великою кількістю ресурсів.
3. Стандартизація конфігурацій та врахування найкращих практик.
4. Покращення співпраці та ефективності командної роботи завдяки використанню коду як єдиного джерела істини.
5. Потенційне зниження помилок від людського фактору, автоматизація змін.
6. Швидкість та ефективність впровадження змін в інфраструктуру.
7. Безпечне відкочування змін завдяки використанню систем контролю версій.
8. Легша інтеграція з практиками *DevOps* та безперервної доставки.
9. Економія часу та ресурсів на підтримку інфраструктури.
10. Перенесення політик доступу та аудиту до коду.

Інструменти IaC:

* спеціалізовані скрипти: *bash* та ін.;
* засоби управління конфігурацією: *Puppet, Ansible* та ін.;
* засоби шаблонізації серверів: *Packer, Vagrant*;
* засоби оркестрації: *Kubernetes* та ін.;
* засоби ініціалізації ресурсів: *Terraform, CloudFormation* та ін.

IaC є невід'ємним компонентом сучасних хмарних середовищ, *DevOps* процесів та розробки масштабованих, надійних додатків. Популярними інструментами для реалізації IaC є *Ansible, Terraform, AWS CloudFormation, Azure Resource Manager* тощо.

Загалом, підхід IaC забезпечує гнучкість, стабільність, можливість повторного використання та більший контроль над ІТ-інфраструктурою через моделі розробки програмного забезпечення.

Проблеми, котрі вирішує IaC:

1) Проблема масштабу. В середньому, кожен мікросервіс потребує від 10 до 12 інфраструктурних ресурсів, таких як *load balancer*, екземпляр RDS, групи безпеки та інші. Якщо у нас є три різних середовища — тестове, підготовче та продуктивне — це вже означає близько 30 різних ресурсів. Проте, коли кількість мікросервісів збільшується до 10, 20 або навіть 100, проблема стає ще більш виразною та масштабною.

2) Проблема передбачуваності. Якщо створювати всі ці ресурси вручну, то питання «Що робити, якщо ми припустимося помилки і наші середовища будуть відрізнятися; до яких багів це може призвести?» перетворюється на «Що робити, коли...» Бо вірогідність припуститися помилки в кількох сотнях ручних операцій наближається до 100%. З урахуванням цих проблем автоматизація інфраструктури стає не просто модним трендом, а необхідністю.

## 2.2 Важливість та переваги впровадження IaC в контексті ведення мережево-центричної війни

Мережево-центрична війна - це сучасна концепція ведення воєнних дій, котра орієнтована на досягнення інформаційної переваги за допомогою об'єднання військових об'єктів у одну інформаційну мережу. Ця концепція включає в себе поєднання різних джерел інформації ([розвідки](https://uk.wikipedia.org/wiki/%D0%92%D0%BE%D1%94%D0%BD%D0%BD%D0%B0_%D1%80%D0%BE%D0%B7%D0%B2%D1%96%D0%B4%D0%BA%D0%B0)), [органів управління](https://uk.wikipedia.org/wiki/%D0%9E%D1%80%D0%B3%D0%B0%D0%BD_%D1%83%D0%BF%D1%80%D0%B0%D0%B2%D0%BB%D1%96%D0%BD%D0%BD%D1%8F) та [засоби ураження](https://uk.wikipedia.org/w/index.php?title=%D0%97%D0%B0%D1%81%D0%BE%D0%B1%D0%B8_%D1%83%D1%80%D0%B0%D0%B6%D0%B5%D0%BD%D0%BD%D1%8F&action=edit&redlink=1) (придушення) та іншої інформації, тим самими забезпечуючи доведення до учасників операцій достовірної та повної інформації про обстановку практично в реальному масштабі часу.

Створення єдиної інформаційної мережі здатне в кілька разів збільшити потужність збройних сил без збільшення їх чисельності. Мережецентрична війна дозволяє піднятися на новий рівень управління військами, значно зменшуючи час прийняття рішень.

Перехід на мережоцентричні принципи ведення війни є важливим кроком для підвищення ефективності та боєздатності українських Збройних Сил у протистоянні російській агресії. Оскільки противник має чисельну перевагу в традиційному озброєнні, мережоцентритчність дозволить Україні компенсувати цю перевагу (буде, так звана, асиметрична перевага) за рахунок кращої координації, обміну розвідданими та маневреності сил. Ця концепція дозволяє більш ефективно використовувати наявні обмежені ресурси - як фінансові, так і людські та матеріальні; забезпечує адаптивність - мережева структура забезпечує гнучкість та можливість швидко адаптуватися до мінливої обстановки; забезпечує взаємодію з союзниками - стандартизовані мережі зв'язку та передачі інформації полегшують координацію та взаємодію з силами союзників.

Використання практик *Continuous Integration* (CI) та *Continuous Deployment/Delivery* (CD) у поєднанні з *Infrastructure as Code* (IaC) відіграє важливу роль у реалізації єдиної інформаційної мережі та системи для мережоцентричної війни.

Побудова системи для мережоцентричної війни без використання *Infrastructure as Code* (IaC) та *Continuous Integration/Continuous Deployment* (CI/CD) буде можливою, але вкрай складною та проблематичною. Ось основні проблеми, з якими доведеться зіткнутися:

1. Повільна розробка та розгортання. Відсутність автоматизації значно сповільнить процес внесення змін, тестування та розгортання оновлень системи. Це критично в умовах швидкоплинних бойових дій.

2. Ризики несумісності та помилок. Ручне налаштування та керування інфраструктурою різних компонентів підвищить ризик помилок, неузгодженостей та проблем з сумісністю між різними частинами системи.

3. Складність масштабування. Розширення або зміна масштабу інфраструктури без автоматизованих інструментів керування буде надзвичайно складною та тривалою задачею.

4. Відсутність стандартизації. Без IaC буде вкрай важко забезпечити стандартизовані та повторювані конфігурації компонентів у різних середовищах.

5. Обмежена видимість та контролю. Без автоматизованих процесів збірки, тестування та розгортання буде складно відстежувати зміни, проблеми та загалом контролювати стан системи.

6. Велике навантаження на персонал. Необхідність виконання більшості завдань вручну потребуватиме великої кількості людських ресурсів, що може стати проблемою, особливо зараз.

Хоча теоретично побудувати таку систему можливо, на практиці відсутність IaC та CI/CD може звести нанівець ключові переваги мережоцентричного підходу - гнучкість, адаптивність та швидке реагування. Система стане повільною, ненадійною та вразливою до помилок.

Враховуючи критичну важливість своєчасного розгортання оновлень, масштабованості та надійності для мережоцентричної війни, використання IaC та CI/CD є фактично обов'язковим для успішної реалізації такої системи.

## 2.3 *Terraform*

Розглянемо інструмент *Terraform* для реалізації підходу "Інфраструктура як код". В контексті IaC, *Terraform* це засіб ініцілізації ресурсів. Засоби ініціалізації ресурсів, на відміну від інших інструментів IaC, створюють самі ресурси, тобто віртуальні машини, кластер *Kubernetes*, бази даних, мережі тощо.

За допомогою засобів ініціалізації ресурсів можна створювати: бази даних, кеші, балансувальники навантаження, системи моніторингу, налаштування підмереж і брандмауера, правила маршрутизації, сертифікати SSL і майже будь-який інший аспект вашої інфраструктури.

Наприклад, наступний код розгортає вебсервер за допомогою *Terraform*.

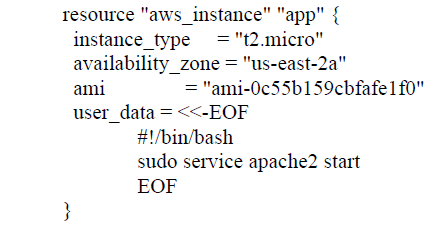
-*ami* – визначає ідентифікатор образу AMI, який потрібно розгорнути на сервері. -user\_data – bash-скрипт, що виконується за час завантаження вебсервера.

Рис. 2.2.1 Приклад *Terraform* коду для розгортання вебсервера

*Terraform* – це інструмент із відкритим вихідним кодом від компанії *HashiCorp*, написаний мовою програмування *Go.* Код на *Go* компілюється в єдиний двійковий файл з назвою *Terraform*.

Цей файл дає змогу розгорнути інфраструктуру прямо з вашого ноутбука (або будь-якого іншого комп'ютера) і для всього цього не потрібно ніякої додаткової інфраструктури. Все завдяки тому, що всередині виконуваний файл *Terraform* робить від вашого імені API-виклики до одного/декількох провайдерів: *AWS, Azure, Google Cloud, DigitalOcean, OpenStack*. Особливості *Terraform*:

1. Підтримка численних провайдерів. *Terraform* підтримує широкий спектр хмарних провайдерів (*AWS, Azure, GCP* та ін.), а також приватні віртуальні та фізичні середовища через власні чи сторонні провайдери.

2. Декларативний синтаксис. Мова конфігурації *Terraform* є декларативною, що означає, що вона описує бажаний кінцевий стан вашої інфраструктури. Конфігурація описується декларативною мовою HCL (*HashiCorp Configuration Language*). Користувач визначає бажаний стан інфраструктури, а *Terraform* визначає необхідні дії для досягнення цього стану (і постійно це контролює).

3. Графова модель залежностей. *Terraform* будує графову модель усіх ресурсів та їхніх залежностей для створення відповідного плану виконання.

4. Ідемпотентність - неодноразове застосування однакової конфігурації буде призводити до того самого результату.

Переваги використання *Terraform*: портативність між різними хмарними провайдерами, зменшення руйнівних помилок завдяки планам виконання, повторне використання конфігурацій як коду, безпечне гранулярне додавання та відкочування змін, можливість тестування та модульної структури конфігурацій, інтеграція з іншими інструментами та практиками *DevOps*.

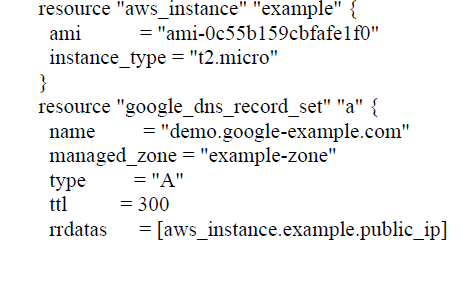
*Terraform* став популярним завдяки простоті синтаксису, гнучкості підтримуваних провайдерів та можливості створювати переносні визначення інфраструктури. Він активно використовується в контексті *DevOps* та безперервної доставки для впровадження практики IaC.

В наступних підрозділах буде описана певні базові технічні деталі про те як працює *Terraform*.

## 2.3.1 Провайдери *Terraform*

Провайдер (*provider*) у контексті *Terraform* - це компонент або плагін, який дозволяє *Terraform* взаємодіяти з певним типом ресурсів або сервісами. Провайдери абстрагують специфічні API різних постачальників хмарних послуг, систем керування конфігураціями тощо за єдиним інтерфейсом *Terraform*. Вони надають набори ресурсів, які можна використовувати у конфігураційних файлах.

Ось приклад конфігурації *Terraform* звідки *Terraform* дізнається, які API-виклики йому потрібно робити:

Рис. 2.2.1.1 Приклад провайдера

Цей фрагмент змушує *Terraform* виконати API-виклики до двох провайдерів: до AWS, щоб розгорнути там сервер, і до *Google Cloud*, щоб створити DNS-запис, який вказує на IP-адресу сервера з AWS.

Що роблять провайдери? Кожен провайдер додає набір типів ресурсів (*resource types*) та/або джерел даних (*data sources*), якими може керувати *Terraform*.

*Terraform* повинен оголосити, які провайдери йому потрібні, щоб *Terraform* міг їх встановити і використовувати.

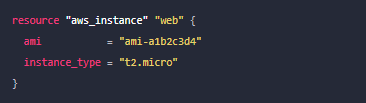
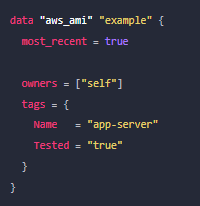
*Resource type* та *Data sources*:

Рис. 2.2.1.2 *Resource type* та *Data sources*

Кожен провайдер має два ідентифікатори:

Рис. 2.2.1.4 Ідентифікатори провайдера

1) Унікальна адреса джерела, яка використовується лише тоді, коли потрібен постачальник.

2) Локальна назва, яка використовується всюди в модулі *Terraform*.

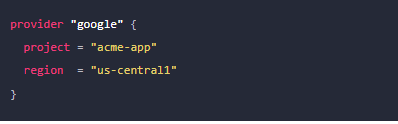
Конфігурації провайдерів налаштовуються в кореневому модулі конфігурації *Terraform*. (Дочірні модулі отримують свої конфігурації провайдера від кореневого модуля.) Конфігурація провайдера створюється за допомогою блоку *provider*:

Рис. 2.2.1.5 Конфігурація провайдера

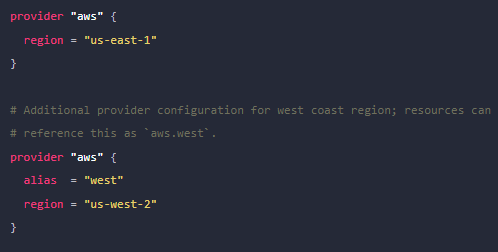
*alias* - для конфігурації кількох постачальників:

Рис. 2.2.1.6 Конфігурація кількох провайдерів

## 2.3.2 Каталоги та модулі

Модулі - це контейнери для упакування конфігурацій *Terraform* для повторного використання. Вони дозволяють інкапсулювати групу ресурсів та їхні залежності в єдиний пакет коду. Модулі можуть бути локальними або дистанційними. [Модуль](https://developer.hashicorp.com/terraform/language/modules) - це набір файлів .tf та/або .tf.json, що зберігаються разом у каталозі. Кореневий модуль *Terraform* складається з конфігураційних файлів верхнього рівня в каталозі, з якого запускається *Terraform*, та дочірніх модулів.

Переваги використання модулів:

* повторне використання коду та абстракція деталей реалізації;
* організація та структурування великих конфігурацій;
* можливість версіонування та оновлення модулів;
* заохочення модульного та компонентного підходу до розробки інфраструктури.

Модулі можуть бути викликані та параметризовані з основних конфігурацій, забезпечуючи гнучкість і повторне використання. Модулі викликаються з інших модулів за допомогою блоків *module*.

Каталоги (*Registries*) *Terraform* - це центральні сховища для публікації та розповсюдження модулів *Terraform*. Вони забезпечують легкий доступ до офіційних та сертифікованих модулів від *HashiCorp* та спільноти.

Основні каталоги: *Terraform* *Registry* - офіційний каталог від *HashiCorp* та приватні/внутрішні каталоги - власні сховища організацій

Переваги використання каталогів:

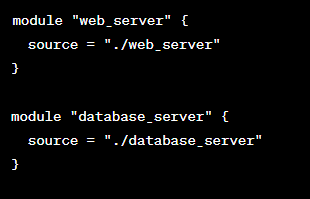
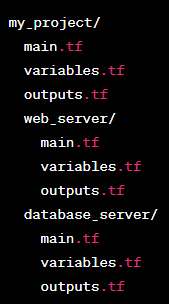
* Спрощений пошук, перегляд та встановлення модулів
* Перевірка якості та безпеки офіційних модулів
* Централізоване управління залежностями модулів
* Можливість використання приватних модулів організацій

Рис. 2.2.2.1 Приклад структури каталогів для конфігурації *Terraform*

Рис. 2.2.2.2 Приклад файлу конфігурації, *my\_project/main.tf*

Комбінування модулів та каталогів сприяє повторному використанню, структуризації та стандартизації кодової бази *Terraform* інфраструктури, забезпечуючи ефективність роботи та відповідність кращим практикам.

## 2.3.3 Основні команди *Terraform*

Основні команди для роботи *Terraform* наступні:

1) *terraform init* – здійснює ініціалізацію проєкту і завантажує провайдери, які необхідні для розгортання.

2) *terraform plan* – можна побачити які саме ресурси *terraform* хоче створити чи видалити.

3) *terraform apply/destroy* деплоїть/видаляє ресурси які побачили на етапі *terraform plan*.

## 2.4 *Ansible*

Розглянемо інструмент *Ansible* для реалізації підходу "Інфраструктура як код". *Ansible* - це потужна *open-source* платформа для автоматизації налаштування віддаленими системами та дозволяє описувати бажану конфігурацію систем у вигляді простих текстових інструкцій, називаних плейбуками (*playbooks*) та контролює її базовий стан.

Базове середовище *Ansible* складається з трьох основних компонентів:

1) Вузол управління (*Control node*) - система, на якій встановлено *Ansible*. Ви запускаєте команди такі як *Ansible* або *Ansible*-*inventory* на вузлі керування.

2) Керований вузол (*Managed node*) - віддалена система або хост, яким керує *Ansible*.

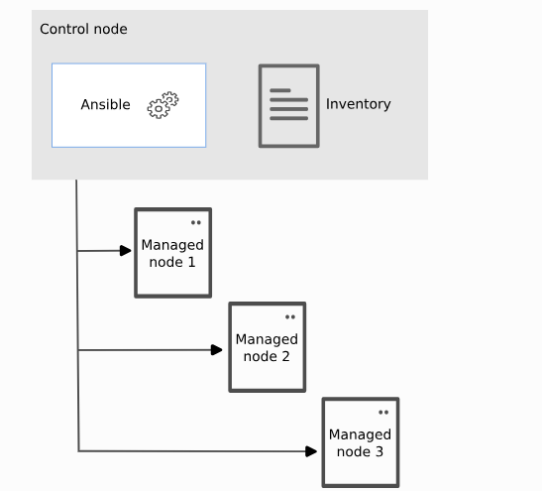
3) Інвентар (*Inventory*) - Список керованих вузлів, які логічно організовані. Ви створюєте інвентар на контрольному вузлі, щоб описати розгортання хосту для *Ansible*.

Рис. 2.3.1 Основні компоненти *Ansible*

Основні принципи роботи *Ansible*:

1. *Ansible* не вимагає встановлення додаткових агентів на керованих вузлах, використовуючи SSH або *Windows* *Remote Management* для підключення.

2. Ідемпотентність: *Ansible* забезпечує застосування одних і тих самих *playbooks* (файл, котрий написаний на YAML та містить набір інструкцій, що визначають конфігурацію та політики управління для групи хостів або серверів) завжди призведе до однакового результату.

3. Модулі: *Ansible* використовує величезну бібліотеку модулів для виконання різноманітних завдань, від управління пакетами до налаштування хмарних сервісів.

4. Інвентар (файл *Inventory*): Керовані вузли описуються в інвентарних файлах у форматах INI, YAML або через динамічні сценарії.

5. *Playbooks*: Конфігурації задаються у вигляді YAML *playbooks*, які є наборами завдань для вузлів із заданими параметрами.

Ключові переваги *Ansible*: простий у вивченні та використанні завдяки декларативній природі та зрозумілому синтаксису, не потребує встановлення клієнтів-агентів на цільові машини, використовує SSH, конфігурації пишуться на YAML, має портал з готовими конфігураціями [*Ansible* *Galaxy*](https://galaxy.ansible.com/), графічна оболонка [UI *Ansible* *Tower*/AWX](https://docs.ansible.com/ansible-tower/latest/html/userguide/main_menu.html), інтеграція з інструментами оркестрації, контейнеризації та CI/CD. *Ansible* активно використовується для:

* налаштування та оркестрації конфігурації серверів, робочих станцій, контейнерів;
* автоматичного розгортання та оновлення застосунків;
* провізії та управління обчислювальними ресурсами у хмарі;
* налаштування мережевого обладнання - маршрутизаторів, комутаторів;
* забезпечення контролю та відстеження конфігурацій (*configuration management*);
* реалізації практик безперервної інтеграції та доставки.

*Ansible* є потужним IaC інструментом завдяки своїй простоті, універсальності та здатності наблизити інфраструктуру до моделей розробки програмного забезпечення із ідемпотентністю, модульністю та версіями *config*.

## Висновок до розділу 2

Інфраструктура як код (IaC) - це парадигма, в якій вся інфраструктура ІТ-середовища визначається за допомогою кодових інструкцій та машиночитаних файлів конфігурацій замість ручного налаштування обладнання та середовищ. Це підхід, при якому інфраструктура розглядається та керується так само, як і код програмного забезпечення. Основна ідея IaC полягає в тому, щоб забезпечити стандартизацію, узгодженість та автоматизацію процесів розгортання, налаштування та управління інфраструктурними компонентами, такими як віртуальні машини, мережі, сховища даних, балансувальники навантаження та інші ресурси. Використовуючи IaC, адміністратори та *DevOps* інженери можуть визначати та керувати всією інфраструктурою за допомогою коду, який може бути збережений у системі контролю версій, модифікований, протестований та автоматично розгорнутий на різних середовищах. Це забезпечує відтворюваність, масштабованість та стабільність інфраструктури, а також полегшує співпрацю, аудит та внесення змін.

*Ansible* є потужним інструментом для автоматизації та управління конфігураціями різних систем та додатків. Він використовує декларативний підхід, що дозволяє описувати бажаний стан систем, а не послідовність дій для досягнення цього стану. *Ansible* характеризується легкістю у вивченні та використанні, agentless архітектурою та можливістю паралельного виконання завдань на багатьох вузлах.

*Terraform* є інструментом для створення, змін та створення версій інфраструктури безпечним та ефективним способом. Він забезпечує узгоджений *workflow* для керування хмарними провайдерами та послугами за допомогою єдиного інструменту. Ключові переваги *Terraform* включають підтримку багатьох провайдерів, декларативний синтаксис, залежності ресурсів, паралельне створення ресурсів.

*Ansible* та *Terraform* мають різні цілі та області застосування, але можуть успішно використовуватися разом у CI/CD процесах. *Ansible* зосереджений на управлінні конфігураціями та розгортанні додатків, тоді як *Terraform* призначений для керування хмарною інфраструктурою та ресурсами різних провайдерів. Використання підходу Інфраструктури як коду та інструментів, таких як *Ansible* та *Terraform*, забезпечує ефективне, надійне та масштабоване управління інфраструктурою та конфігураціями. Це дозволяє підвищити швидкість розгортання, мінімізувати помилки, забезпечити відтворюваність та полегшити співпрацю між різними командами в рамках *DevOps* практик.

# Розділ 3 Аналіз моделі використання CI/CD під час розгортання інфраструктури на основі IaC

## 3.1 Аналіз способів інтеграції CI/CD з інструментами інфраструктури як коду, такими як *Ansible*, *Terraform*

Розглянемо способи інтеграції практик безперервної інтеграції та безперервної доставки (CI/CD) з інструментами інфраструктури як коду (IaC), такими як *Ansible* та *Terraform*. Інтеграція *Ansible* з CI/CD:

1. Виконання *Ansible* *playbook* у CI процесі: *playbook* можна запускати як один з етапів CI конвеєра, це дозволяє автоматично тестувати та розгортати конфігурації на тимчасових або тестових вузлах.

2. Моделювання інфраструктури за допомогою *Ansible* для тестування у CD: *Ansible* дозволяє розгорнути всю інфраструктуру в тимчасовому середовищі, в цьому середовищі можна виконувати розгалужене тестування перед релізом.

3. *Ansible* *Tower* або AWX для оркестрації CD процесів - ці інструменти забезпечують вебінтерфейс для керування *Ansible*, їх можна використовувати як центральний вузол для запуску та синхронізації CD конвеєрів.

4. Інтеграція *Ansible* з CI/CD системами (*Jenkins*, *GitLab* CI/CD, *CircleCI*): через забудовані плагіни або написання спеціальних етапів конвеєрів, це спрощує запуск та керування *Ansible* завданнями в межах CI/CD процесів.

Інтеграція *Terraform* з CI/CD:

1. Виконання планування та застосування змін *Terraform* в CI, команда: *terraform* *plan* генерує план змін після кожного коміту, команда *terraform* *apply* здійснює зміни на етапі CD після перевірок

2. Тестування *Terraform* конфігурацій: юніт-тести (*sentctl*, *Terraform*-*validator* та ін.) для перевірки коректності синтаксису, тести інтеграції для розгортання тимчасових середовищ.

3. Управління stage змінами та робочими просторами: робочі простори для розділення різних середовищ, *Remote state backends* для відстеження стану.

4. Конвеєри із застосуванням практик безпечного розгортання: плани змін вимагають схвалення перед застосуванням, використання інструментів *blue/green* або *canary* *deployment*

5. Інтеграція з CI/CD інструментами через специфічні провайдери: провайдери для різних платформ: *Kubernetes*, AWS, *Azure* тощо, написання власних етапів *Terraform* у CI/CD конвеєрах

Обидва інструменти, *Ansible* та *Terraform*, добре інтегруються з підходами CI/CD через різноманітні сценарії використання та можливість включення їх у автоматизовані конвеєри. Це забезпечує постійну перевірку та надійне розгортання інфраструктурного коду згідно з принципами *DevOps*.

## 3.2 Практичні аспекти інтеграції CI/CD з IaC інструментами

Розглянемо деякі практичні аспекти інтеграції інструментів інфраструктури як коду (IaC) з процесами безперервної інтеграції та безперервної доставки (CI/CD).

Але перед порядком необхідних дій та кроків реалізації та описом цих технічних аспектів необхідно підготуватися до цієї роботи, а власне встановити *Git*, *Ansible*, *Terraform*. Тож спершу про те що потрібно зробити:

1) Спершу потрібно здійснити установку *git*.

Окрім встановлення наступна необхідна дія це генерація SSH-ключів та додати їх до *ssh-agent* з використанням наступної команди у *Git Bash*:

*ssh-keygen -t ed25519 -C "your\_email@example.com"*

Коли вам буде запропоновано «Введіть файл, у якому потрібно зберегти ключ» (*Enter a file in which to save the key:*), ви можете натиснути ***Enter*** , щоб прийняти розташування файлу за замовчуванням. Зауважте, що якщо ви раніше створювали ключі SSH, *ssh-keygen* може попросити вас переписати інший ключ, у такому випадку ми рекомендуємо створити ключ SSH із власною назвою. Для цього введіть розташування файлу за замовчуванням і замініть *id\_ALGORITHM* на ім’я спеціального ключа.

Перш ніж додати новий ключ SSH до агента ssh для керування вашими ключами, ви повинні перевірити наявні ключі SSH і створити новий ключ SSH.

У новому вікні *PowerShell* із правами адміністратора переконайтеся, що ssh-агент запущено:

*Get-Service -Name ssh-agent | Set-Service -StartupType Manual*

*Start-Service ssh-agent*

У вікні терміналу без підвищених дозволів додайте свій закритий ключ SSH до ssh-агента. Якщо ви створили свій ключ з іншим ім’ям або якщо ви додаєте наявний ключ з іншим ім’ям, замініть id\_ed25519 у команді на ім’я вашого файлу приватного ключа:

*ssh-add c:/Users/YOU/.ssh/id\_ed25519*

Додайте відкритий ключ SSH до свого облікового запису на *GitHub* або *GitLab*.

2) Встановлення *Terraform* в залежності від операційної системи, все дуже швидко та просто знаходиться в документації

3) Встановлення *Ansible*.

Увага! *Ansible* має певні вимоги до вузла керування та керованого вузла, а саме: для вашого контрольного вузла (машини, на якій працює *Ansible*) ви можете використовувати майже будь-яку UNIX-подібну машину зінстальованим *Python* 3.8 або новішою, а для керованого вузла (машина, якою керує *Ansible*) не потребує встановлення *Ansible*, але вимагає *Python* 2.7 або *Python* 3.5 - 3.11 для запуску згенерованого *Ansible* коду *Python*. Керованому вузлу також потрібен обліковий запис користувача, який може підключатися через SSH до вузла з інтерактивною оболонкою POSIX. Перевірити яка версія *Python* встановлена можна за допомогою команди: *python*3 *–version*.

А після цього необхідно встановити *ansible* відповідно до того, яка операційна система використовується. У моїй роботі наводитиметься приклад із дистрибутивом *Ubuntu*. Отже для *Ubuntu* послідовність така:

Оновити список репозиторіїв: *sudo apt update*;

1) встановити пакет *software-properties-common*, який містить інструменти, які дозволять додавати нові джерела пакетів (репозиторії) до системи та керувати ними: *sudo apt install software-properties-common*

2) додати репозиторій *ppa:ansible/ansible* до вашої системи та оновити інформацію про пакети з цього репозиторію з використанням команди: *sudo add-apt-repository --yes --update ppa:ansible/ansible*

3) інсталювати *ansible*: *sudo apt install ansible*

Перевірити встановлення можна за допомогою команди:

*sudo ansible –version.*

Далі буде описано практичні аспекти роботи інструментів IaC, разом з процесами CI/CD.

## 3.2.1 Репозиторій коду та система контролю версій

Код IaC (*Ansible* *playbook*, *Terraform* конфігурації) має зберігатись в репозиторії контролю версій, це забезпечує відстеження змін, можливість повернення до попередньої версії та співпрацю. Власне для цього нам необхідно зробити наступне:

1) Встановлення *Git*, який є розповсюдженою системою контролю версій. Її потрібно встановити на всіх машинах, де буде редагуватися IaC.

2) Ініціалізація *Git* репозиторію: створіть новий репозиторій за допомогою команди *git* *init*, або ж клонуйте вже наявний репозиторій з віддаленого сховища (*GitLab* або *GitHub*) за допомогою *git* *clone*

3) Здійснення змін та комітів: після редагування файлів IaC, потрібно додати їх до зони *staging*: *git* *add* . Закомітити зміни із описовим повідомленням: *git* *commit -m* "Описовий коміт"

4) Гілкування та злиття: використовуйте різні гілки для функціоналу та виправлень (*git* *branch* нова\_гілка), після завершення роботи в гілці – виконайте злиття в основну гілку (*git* *merge* нова\_гілка)

5) Віддалені репозиторії: далі необхідно створити обліковий запис у *Git*-хостингу (*GitHub*, *GitLab*, *Bitbucket*), далі додати цей віддалений репозиторій: *git remote add origin* <https://github.com/repo.git> (як приклад), також регулярно пуште зміни у віддалений репозиторій з використанням команди: *git push -u origin main*

6) Керування доступом та перегляд логів (сповіщень, змін): використовуйте *git* *log* для перегляду історії комітів та відстеження змін.

Додатково можна налаштувати:

* *Git* *hooks* для автоматизації завдань на певних подіях.
* Використовувати *branching* *workflow* (*git*-flow, *GitHub* *flow*).
* Інтегрувати *Git* з системами CI/CD для автоматичного тестування/релізу

Використання системи контролю версій *Git* є обов'язковим для ефективної співпраці, відстеження історії та можливості повернення коду IaC в межах практик CI/CD та *DevOps*.

Коли розробники розміщують у системі контрою версій усі файли вихідного коду і файли конфігурацій, вона стає єдиним сховищем і містить точні стани, що призначаються для використання. Однак, оскільки для постачання продукту клієнту потрібен і код, і середовище, у якому він працює, треба щоб середовище також зберігалось в системі контролю версій.

Для того щоб ми могли повторно й передбачувано (а в ідеалі – ще й швидко) відновлювати виробничі послуги навіть при катастрофічних подіях, треба перевірити, чи наявні в репозиторії системи зберігання версій такі активи:

* Увесь код програми та всі залежності;
* усі сценарії, що використовуються для створення схем баз даних, довідникові дані застосунків і. т. п;
* усі інструменти для створення середовища й артефактів (образи *WMware*, або AMI, рецепти *Puppet* або *Chef* і . т. п);
* усі файли, що використовуються для створення контейнерів (наприклад, конфігураційні файли *Docker* або *Rocket*);
* усі допоміжні автоматичні тести і сценарії тестування вручну;
* усі сценарії, що забезпечують упаковку коду, розгортання, міграцію баз даних і надання робочого середовища;
* усі артефакти проєкту (документація з описом вимог, процедури розгортання, примітки до релізу тощо);
* усі файли конфігурації хмар (шаблони форматування AWS *Cloud*, файли *Microsoft Azure Stack* DSC, *OpenStack* HEAT);
* усі інші сценарії або інформація про конфігурацію, що потрібна для створення інфраструктури, котра підтримує декілька служб.

## 3.2.2 Структура репозиторію та організація коду

1) Розділення коду додатку та інфраструктури: далі для ефективного впровадження інструментів CI/CD для побудови інфраструктури на основі IaC, необхідно створити окремі *Git* репозиторії для коду додатку та конфігурацій IaC, наприклад, *repo-app* для коду додатку і *repo-iac* для *Ansible*/*Terraform*. Це забезпечить розділення проблем та незалежність життєвих циклів.

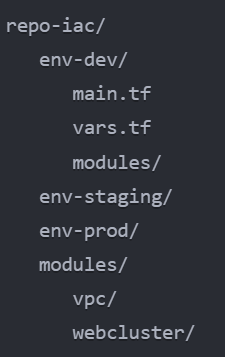
2) Модульний дизайн конфігурацій IaC: в *Ansible* використовуйте ролі для інкапсуляції компонентів інфраструктури, *Terraform* дозволяє створювати модулі для повторного використання. Зберігайте ролі/модулі в окремих підкаталогах репозиторію. Сепарація середовищ: створіть окремі директорії для *dev, staging, prod* та інших середовищ, також дублюйте всю ієрархію конфігурації для кожного середовища, або використовуйте параметри/змінні для розділення конфігурацій.

Рис. 3.2.2.1 Приклад *Ansible* репозиторію та приклад *Terraform* структури

3) Ієрархічна структура: головний конфігураційний файл (*site.yml* в *Ansible*, *main.tf* в *Terraform*), а також імпорт/включення ролей чи модулів за потреби, наприклад: *site.yml -> app.yml* -> вебсервери.yml, дб-сервери.yml.

Належна структуризація IaC коду допомагає впорядкувати все середовище, забезпечити модульність, можливість реутилізації компонентів та чітке розділення відповідальності між середовищами згідно з практиками *DevOps*.

## 3.2.3 Розгортання тимчасових середовищ

Щоби створити швидкий і надійний потік від розробки до експлуатації, ми маємо забезпечити використання середовищ, близьких до виробничих, на кожній стадії потоку створення цінності. Ба більше, ці середовища повинні створюватися автоматизовано, в ідеальному випадку - на вимогу, за допомогою сценаріїв та інформації про зміни, що зберігаються в системі контролю версій, також і середовища мають обслуговуватися цілком автоматично, не потребуючи будь-яких втручань із боку відділу експлуатації. Мета - забезпечити створення наново всього виробничого середовища, спираючись на дані, що зберігаються в системі контролю версій.

Занадто часто ми виявляємо, що наші програми працюють у середовищі наближеному до виробничого, лише під час розгортання — коли вже надто пізно усувати проблеми без негативного впливу на клієнтів.

Однією із причин хаосу, дезорганізації, а часом і катастрофічних наслідків релізів програмного забезпечення є те, що лише під час релізу ми вперше дізнаємось, як застосунок працює у середовищі з реальними навантаженнями і в реальному виробничому середовищі. Однак відділу експлуатації потрібен тривалий час для підготовки таких тестових середовищ, і команди не можуть отримати їх досить швидко. Замість того, щоби відділ експлуатації вручну створював і налаштовував середовище, що можемо використати автоматизацію для таких способів створення середовища:

* Копіювання віртуального середовища (наприклад, образу *WMware*, запуску сценарію *Vagrant*, завантаження файлу *Amazon Machine Image* в EC2).
* За допомогою інструментів управління конфігураціями IaC (*Ansible, Terraform, Chef, Puppet* тощо).
* За допомогою інструментів автоматизованого налаштування операційної системи (*Solaris Jumpstart, Red Hat Kickstart, Debian preseed*).
* Складання середовища з набору віртуальних образів або контейнерів (*Docker, Vagrant*)
* Складання нового середовища в загальнодоступній хмарі (наприклад, AWS*, Microsoft Azure, Google App Engine*), приватній хмарі або інших PaaS (на кшталт *OpenStack, Cloud Foundry*)

*Terraform* підтримує концепцію робочих просторів, що дозволяє керувати різними екземплярами однієї й тієї ж інфраструктури. Робочі простори забезпечують ізоляцію стану, що дозволяє, наприклад, розгортати різні версії або конфігурації інфраструктури в різних середовищах, таких як *development, staging* та *production*. Ось основні кроки для використання робочих просторів *Terraform* у CI/CD процесі для розгортання тимчасових середовищ:

1. Створення робочих просторів: Перед розгортанням інфраструктури необхідно створити робочий простір для цільового середовища. Це можна зробити за допомогою команди *terraform* *workspace new* *[NAME]*.

2. Вибір робочого простору: Після створення робочого простору його потрібно вибрати для подальших операцій *Terraform*. Використовуйте команду *terraform* *workspace select* *[NAME]*.

3. Ізольоване збереження стану: *Terraform* зберігає стан для кожного робочого простору окремо. Це забезпечує ізоляцію інфраструктури середовищ.

4. Конфігурація змінних середовища: Залежно від вимог, ви можете налаштувати змінні середовища для кожного робочого простору. Це дозволить налаштувати поведінку *Terraform* для різних середовищ, наприклад, використовувати різні значення для розмірів ресурсів або налаштувань безпеки.

5. Автоматизація розгортання в CI/CD: В CI/CD процесі можна автоматизувати створення, вибір та застосування робочих просторів для різних середовищ. Наприклад, у процесі безперервної інтеграції можна створювати тимчасовий робочий простір для тестового середовища, застосовувати зміни в інфраструктурі та виконувати тести. Після успішного проходження тестів можна застосувати зміни до наступного середовища (*staging* або *production*).

Після тестування видаліть робочий простір: *terraform* *workspace select default* && *Terraform* *workspace delete ci-test*.

Для *Ansible*: використовуйте динамічні інвентарі для автоматичного створення списку вузлів. Динамічні інвентарі використовуються в *Ansible* для автоматичного створення списку вузлів на основі різних джерел даних, таких як хмарні провайдери, системи конфігураційного менеджменту або інші інструменти автоматизації. Порядок необхідних кроків використання інвентарів:

**1) Створити скрипт або програму для динамічного інвентаря**

*Ansible* підтримує різні типи динамічних інвентарів, такі як скрипти, виконувані файли або програми на різних мовах програмування. Ці скрипти/програми повинні повертати JSON-дані, що описують хости та їх групи.

Наприклад, ви можете створити скрипт *dynamic*\_*inventory*.*py* на *python*, який буде отримувати список хостів з хмарного провайдера, такого як AWS EC2.

2) **Налаштувати *Ansible* на використання динамічного інвентаря**

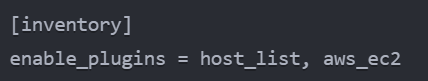
Оновіть файл конфігурації *Ansible* *ansible*.*cfg* або використайте опцію командного рядка, щоб вказати джерело динамічного інвентаря. Наприклад:

Рис. 3.2.3.1 Приклад використання динамічного інвентаря

Або використайте опцію командного рядка: *ansible*-*playbook* *-i dynamic\_Inventory.py site.yml*

3) **Виконати *playbook* або *ad-hoc* команду.** Після налаштування динамічного інвентаря ви можете виконувати *playbook* або ***ad-hoc*** команди, як зазвичай. *Ansible* автоматично отримає список хостів від динамічного інвентаря:

*ansible-playbook -i dynamic\_inventory.py site.yml або Ansible -i dynamic\_inventory.py all -m ping*

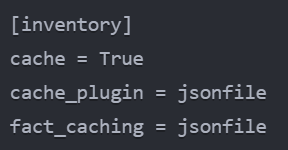
**4) Налаштувати кешування (опціонально).** Динамічний інвентар може бути ресурсоємним процесом, тому *Ansible* дозволяє кешувати результати динамічного інвентаря для підвищення продуктивності. Для цього налаштуйте кешування у файлі конфігурації або за допомогою змінних оточення.

Рис. 3.2.3.1 Приклад налаштування кешування

5) Тестування та налагоджнення

Перевірте вивід скрипту динамічного інвентаря, щоб переконатися, що він повертає належні дані. При налагодженні ви можете використовувати опцію --*list-hosts* для перегляду списку хостів, які повертає динамічний інвентар:

*ansible-inventory -i dynamic\_inventory.py --list-hosts*

Використання динамічних інвентарів дозволяє автоматично отримувати актуальний список хостів із зовнішніх джерел даних, що є особливо корисним у хмарних та динамічних середовищах. Це спрощує процес управління конфігураціями та оркестрацією завдяки *Ansible*.

Видалення тимчасових ресурсів: зазвичай етапи CI/CD мають виконувати "прибирання" після тестів. В *Ansible* через окремі завдання або *handlers*. В *Terraform* через *Terraform* *destroy* для відповідного робочого простору.

## 3.2.4 Універсальні вхідні параметри та секрети

Параметризація конфігурацій для переносимості між середовищами: для забезпечення переносимості конфігурацій *Terraform* між різними середовищами використовуються вхідні змінні (*input variables*). Змінні дозволяють параметризувати значення, такі як імена ресурсів, розміри екземплярів, налаштування мережі тощо. Визначення змінних відбувається у файлі `*variables.tf*`, а їх значення можна задавати за допомогою файлів змінних середовища (.*tfvars*) або командного рядка під час застосування конфігурацій. Використання змінних забезпечує гнучкість та можливість легко адаптувати конфігурації до різних вимог середовищ, не змінюючи основний код *Terraform*.

Під час розгортання інфраструктури часто потрібно передавати конфіденційну інформацію, таку як креденшали доступу до хмарних провайдерів, токени доступу або паролі. Зберігання таких секретів безпосередньо у конфігураційних файлах *Terraform* є небезпечною практикою, оскільки це може призвести до витоку конфіденційних даних. Для безпечного зберігання та передачі секретів рекомендується використовувати спеціалізовані інструменти керування секретами, такі як *HashiCorp Vault, AWS Secrets Manager, Azure Key Vault* тощо. Ці інструменти дозволяють централізовано та безпечно зберігати секрети та надавати тимчасові креденшали або токени під час виконання CI/CD процесів.

## 3.2.5 Тестові набори та перевірка IaC коду

На етапі автоматизованого тестування розробники та відділ експлуатації використовують у повсякденній роботі середовища, наближені до виробничих. Вони успішно виконують інтеграційну збірку і запускають код у такому середовищі після додання кожної нової функції, і при цьому всі зміни фіксуються в системі контролю версій. Однак, ми найімовірніше, отримаємо небажані результати, якщо почнемо шукати і виправляти помилки на окремому етапі тестування. І якщо тестування виконується лише кілька разів на рік, то розробники дізнаються про пропущені промахи лише за кілька місяців після того, як була внесена зміна, що призвела до помилки. За цей час зв’язок між причиною і наслідками буде втрачений, а розв’язання проблеми вимагатиме героїчних зусиль і майже археологічних розкопок. Що найгірше, значно зменшиться наша здатність вчитися на помилках і застосовувати отриманий досвід у майбутній роботі.

Автоматизоване тестування розв’язує ще одну серйозну і тривожну проблему. Гері Грувер зазначав: «Якщо немає автоматизованого тестування, то що більше коду ми пишемо, то більше часу і коштів знадобиться для перевірки – і в більшості випадків це абсолютно немасштабована бізнес-модель для будь-якої технічної організації». Особливо виразно важливість автоматизованого тестування показала команда GWS (див додаток 3).

Тестування є важливим аспектом у процесі розробки та застосування IaC для забезпечення якості, надійності та відповідності вимог. Розглянемо декілька підходів до тестування IaC коду, зокрема *Terraform*.

1) Статичний аналіз та юніт-тести.

Модульне тестування (юніт-тести) перевіряють один метод, клас або функцію ізольовано від інших, показуючи розробникам, що їх код працює так, як задумано

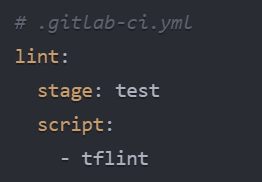
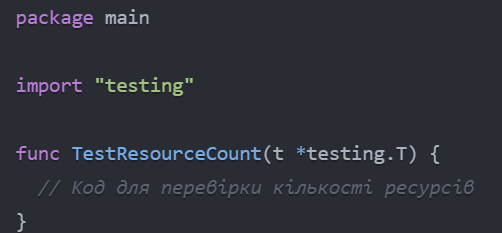
Перед застосуванням змін у *Terraform* конфігураціях, необхідно виконати базову перевірку синтаксису за допомогою команди *terraform* *validate*. Для більш глибокого статичного аналізу можна використати інструмент *tflint*. Його можна інтегрувати в CI/CD процес, наприклад, в *GitLab* CI/CD:

Рис. 3.2.5.1 Інструмент *tflint*

Для юніт-тестування *Terraform* коду можна використати фреймворк *go test.* Для прикладу, створіть файл *main\_test.go* поруч з main.tf:

Рис. 3.2.5.2 Приклад юніт-тесту

Додайте крок виконання юніт-тестів у CI/CD конвеєрі.

2) Приймальне тестування – зазвичай це тестування застосунку загалом. Воно необхідне для того, аби переконатися, що більш високий рівень функціональності працює так, як задумано (наприклад, відповідність вимог та критерій до результату, правильність API), і що відсутні помилки регресії (тобто не пошкоджені функції, що раніше працювали правильно)

3) Інтеграційне тестування:

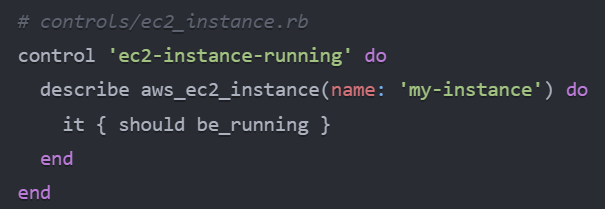
Для інтеграційного тестування розгорнутих ресурсів можна використати інструмент *inspec*. Створіть профіль *InSpec* з тестами для перевірки стану ресурсів, наприклад, перевірка стану EC2 instances в AWS:

Рис. 3.2.5.3 Перевірка стану EC2 instances в AWS

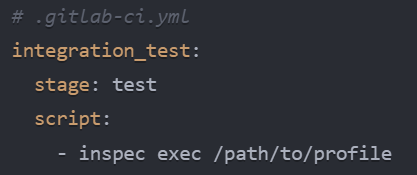
Додайте крок запуску InSpec тестів у CI/CD конвеєрі після успішного застосування *Terraform* змін:

Рис. 3.2.5.4 Запуск InSpec тестів у CI/CD конвеєрі

4) Тестування відповідності політикам

Для перевірки відповідності *Terraform* конфігурацій певним політикам безпеки та кодовим стандартам можна використати *terraform*-*compliance*.

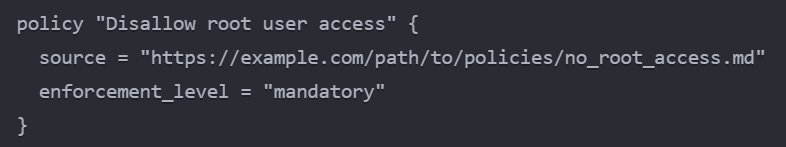
Створіть файл з правилами відповідності:

Рис. 3.2.5.5 Приклад *policies/policy.md*

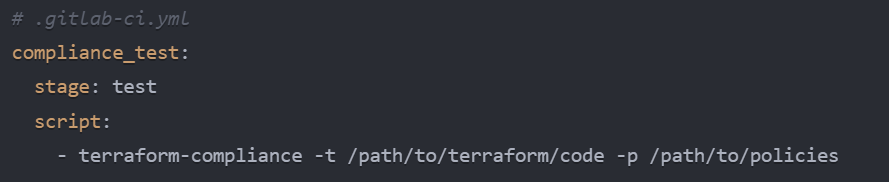
Інтегруйте виконання *terraform*-*compliance* в CI/CD конвеєрі:

Рис.3.2.5.6 інтегрування виконання *terraform*-*compliance* в CI/CD конвеєрі

Ці приклади демонструють, як інтегрувати різні види тестування IaC коду в CI/CD процеси. Послідовність дій може включати: статичний аналіз → юніт-тести → застосування *Terraform* змін → інтеграційні тести → тести відповідності політикам. Така практика забезпечує якість, надійність та безпеку IaC коду перед розгортанням у продуктивне середовище.

## 3.2.6 Артефакти конфігурацій

Розглянемо питання артефактів конфігурацій в CI/CD процесах з IaC інструментами, такими як *Terraform*.

1) Збереження та передача збірок конфігурацій між етапами конвеєра: в CI/CD процесах часто потрібно передавати артефакти (наприклад, конфігураційні файли, стан *Terraform*) між різними етапами конвеєра. Для цього використовуються спеціальні сховища артефактів (*artifact repositories*) або тимчасові сховища, інтегровані в CI/CD систему.

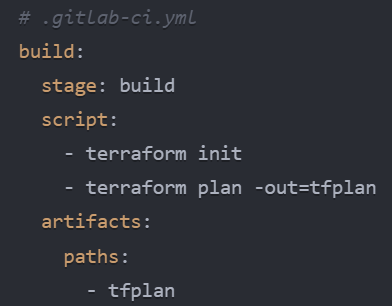
Наприклад, в *GitLab* CI/CD можна використовувати вбудовані сховища артефактів. Після створення збірки конфігурацій їх можна зберегти як артефакт:

Рис. 3.2.6.1 Приклад збереження артефакту

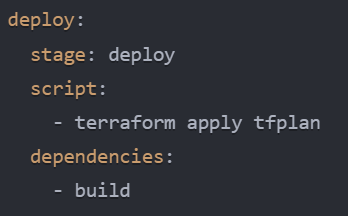
Наступний етап конвеєра - артефакт можна завантажити та використати:

Рис. 3.2.6.2 Приклад використання артефакту

2) Використання *artifact repositories* або *immutable infrastructure artifacts.*

Крім тимчасових сховищ артефактів, інтегрованих у CI/CD системи, можна використовувати спеціалізовані сховища артефактів інфраструктури (*artifact repositories*). Ці сховища зберігають незмінні (*immutable*) артефакти інфраструктури, такі як конфігураційні файли *Terraform*, стани, бінарні файли провайдерів тощо.

Використання незмінних артефактів забезпечує відтворюваність та стабільність розгортань інфраструктури в різних середовищах. Прикладами таких сховищ є *Terraform Cloud, Terraform Enterprise, Artifactory, Nexus Repository* тощо. Під час CI/CD процесу артефакти можуть завантажуватися з цих сховищ, а після успішного розгортання нові артефакти можуть зберігатися назад у сховище. Використання централізованих сховищ артефактів та незмінних артефактів інфраструктури забезпечує кращу організацію, відтворюваність та стабільність CI/CD процесів з IaC інструментами. Це також полегшує співпрацю, аудит та відстеження змін.

## 3.2.7. Схвалення змін та безпечне розгортання

Наша задача спростити та автоматизувати якомога більше виконуваних вручну кроків, а саме:

* упаковку коду відповідним для розгортання чином;
* створення попередньо налаштованих образів віртуальних машин або контейнерів;
* автоматизацію розгортання і налаштування конфігурації проміжного програмного забезпечення;
* копіювання пакетів або файлів на виробничі сервери;
* перезапуск серверів, застосунків або служб;
* створення файлів конфігурації із шаблонів;
* запуск процедур тестування;
* створення сценаріїв та автоматизація міграції бази даних.

Схвалення змін та безпечні стратегії розгортання є важливими аспектами CI/CD процесів, особливо коли йдеться про управління інфраструктурою за допомогою IaC інструментів, таких як *Terraform*. Розглянемо ці моменти:

1. Ручні або автоматизовані схвалення перед застосуванням змін:

Перед застосуванням змін в інфраструктурі, особливо в продуктивному середовищі, рекомендується впровадити етап схвалення змін. Ручне схвалення передбачає, що людина (наприклад, *DevOps* інженер або адміністратор) переглядає та схвалює зміни, перш ніж вони будуть застосовані. Автоматизоване схвалення може бути реалізоване шляхом перевірки певних умов або політик, наприклад, успішне проходження всіх тестів, відповідність правилам безпеки тощо. В *GitLab* CI/CD етап схвалення можна реалізувати з допомогою *manual job*:

Рис. 3.2.7.1 Приклад розгортання з ручним схваленням.

2. Стратегії розгортання: *blue/green, canary, rolling update*:

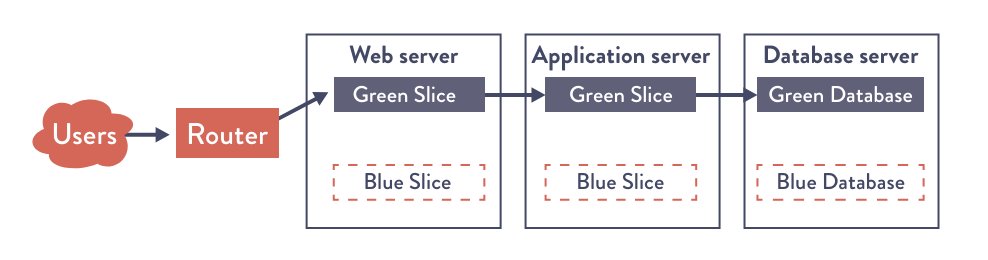
*Blue/Green*: Ця стратегія передбачає створення двох ідентичних виробничих середовищ (*blue* та *green*). Одне середовище активне, а інше використовується для розгортання нової версії. Для релізу нової версії ми розгортаємо його в неактивне середовище, де можна виконати тестування, не втручаючись у роботу користувачів Після успішного розгортання та тестування, трафік переключається на нове середовище. Повернення до початкового стану можна здійснити перенаправленням трафіку знову до зеленого середовища.

Рис. 3.2.7.2 Синьо-зелений шаблон розгортання

Існують й інші шляхи реалізації синьо-зеленого патерна:

* налаштування декількох *Apache/NGINX* вебсерверів на прослуховування різних фізичних або віртуальних інтерфейсів;
* використання декількох віртуальних кореневих об’єктів на серверах із *Windows IIS*, прив’язаних до різних портів;
* за допомогою різних каталогів для кожної версій системи і символічного посилання, що вказує, яке із середовищ є робочим;
* паралельна робота декількох версій сервісів або проміжного програмного забезпечення, що прослуховують окремі порти;
* застосування двох різних центрів обробки даних і комутація трафіку між ними замість того, щоби використовувати їх лише як «гарячий» або «теплий» резерв для аварійного відновлення;
* за допомогою створення різних зон доступності в середовищі хмарних обчислень;

Шаблон синьо-зеленого розгортання дуже простий в реалізації і може значно підвищити безпечність релізу ПЗ. Існують варіанти цього шаблону, що здатні сприяти подальшому покращенню безпеки і скороченню часу розгортання за допомогою автоматизації, але це «вартує» ускладненням процесу.

*Canary:* При використанні цієї стратегії нова версія спочатку розгортається на невелику частину продуктивних ресурсів (*canary*). Якщо нова версія працює коректне, вона поступово розгортається на решту ресурсів.

На рисунку нижче представлені групи середовищ, створені в компанії Facebook, які використовувалися для підтримки цього шаблона.

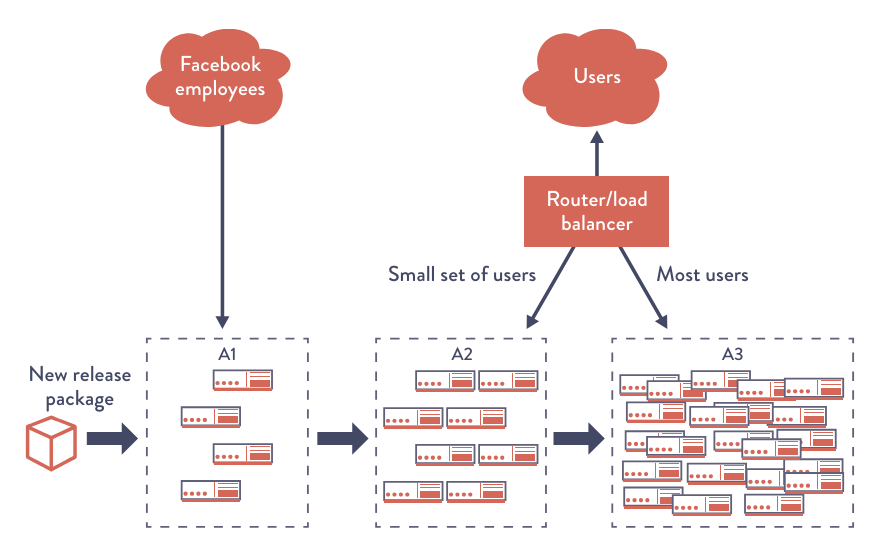


Рис. 3.2.7.2 Шаблон канаркового релізу

*Rolling Update*: Ця стратегія передбачає поступове оновлення ресурсів у продуктивному середовищі. Нова версія розгортається на частину ресурсів, а після успішної перевірки - на решту ресурсів. При цьому система залишається в робочому стані протягом усього процесу оновлення.

Реалізація цих стратегій в *Terraform* може бути досягнута шляхом використання робочих просторів, модульної структури коду, відповідної конфігурації провайдерів та ресурсів в хмарі. Наприклад, для *blue/green* можна створити два робочі простори для двох середовищ. Впровадження етапів схвалення змін та використання безпечних стратегій розгортання в CI/CD процесах з IaC інструментами допомагає мінімізувати ризики, забезпечити контрольоване розгортання змін та зменшити час простою під час оновлень.

## 3.2.8. Моніторинг та журналювання

Розглянемо важливий аспект моніторингу та журналювання в CI/CD процесах з IaC інструментами:

1. Відстеження подій і статусу розгортання через логи та моніторинг: Під час CI/CD процесів, особливо на етапах розгортання та застосування змін в інфраструктурі, важливо мати можливість відстежувати події, статус та результати виконання. Більшість IaC інструментів, таких як *Terraform*, *Ansible* та ін., генерують вихідні логи, які можна збирати та аналізувати. В CI/CD системах, таких як *GitLab* CI/CD, *Jenkins* або інших, логи кожної задачі (*job*) зберігаються та доступні для перегляду. Крім того, можна інтегрувати централізовану систему збору та аналізу логів, наприклад, *ELK Stack (Elasticsearch, Logstash, Kibana), Graylog або Splunk.* Ці системи дозволяють не лише зберігати та переглядати логи, але й виконувати пошук, фільтрацію, візуалізацію та встановлювати сповіщення на основі певних подій у логах.

2. Інтеграція з системами моніторингу інфраструктури та додатків:

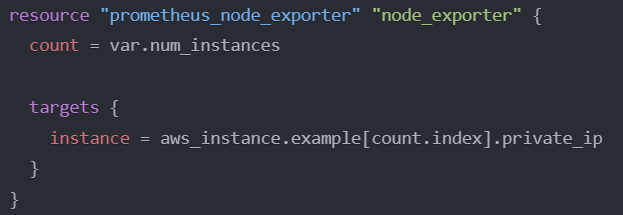
Після успішного розгортання інфраструктури та додатків за допомогою IaC інструментів важливо інтегрувати їх з системами моніторингу. Системи моніторингу, такі як *Prometheus, Datadog, New Relic* або *AWS CloudWatch,* дозволяють відстежувати стан та продуктивність розгорнутих ресурсів, включаючи обчислювальні ресурси, мережі, сховища даних, додатки тощо. Інтеграція з цими системами може бути реалізована різними способами, наприклад, за допомогою агентів моніторингу, які розгортаються разом з інфраструктурою, або шляхом використання вбудованих можливостей моніторингу хмарних провайдерів. Дані моніторингу можуть використовуватись для виявлення та усунення проблем, оптимізації ресурсів, встановлення сповіщень про інциденти та забезпечення відповідності вимогам SLA.

Рис. 3.2.8.1 Приклад інтеграції з *Prometheus* в *Terraform*

Цей код розгортає *Node Exporter* на кожному створеному EC2 instance в AWS, дозволяючи *Prometheus* збирати метрики про стан та їх продуктивність.

Впровадження належного моніторингу та журналювання в CI/CD процеси з IaC інструментами забезпечує прозорість, відстеження подій, своєчасне виявлення проблем та полегшує усунення несправностей у разі виникнення будь-яких проблем під час розгортання або експлуатації інфраструктури.

## 3.2.9 Зворотні засоби (*rollback*, *rollforward*)

Розглянемо ще не менш важливу тему зворотних засобів (*rollback* та *rollforward*) у CI/CD процесах з інструментами IaC, такими як *Terraform*:

1. Зберігання попереднього та поточного станів інфраструктури:

- Більшість IaC інструментів, зокрема *Terraform*, зберігають стан інфраструктури у спеціальних файлах стану (*state files*). Ці файли містять інформацію про створені ресурси, їх налаштування та зв'язки між ними. Під час виконання операцій з інфраструктурою (створення, оновлення, видалення) *Terraform* зберігає поточний стан після застосування змін. Перед застосуванням нових змін *Terraform* також зберігає попередній стан інфраструктури, який можна використовувати для відкату змін у разі потреби.

2. Можливість відкату або перезастосування змін у разі невдачі:

*Rollback*: Якщо під час розгортання нових змін виникла помилка або непередбачена ситуація, *Terraform* забезпечує можливість відкату до попереднього робочого стану інфраструктури. Для виконання відкату потрібно використати команду *terraform* *apply* з попереднім файлом стану: *terraform* *apply -state=previous.tfstate*. Це призведе до відкату всіх змін, внесених під час останнього застосування, і відновить інфраструктуру до попереднього стабільного стану. *Rollforward*: У деяких ситуаціях може знадобитися перезастосування змін, якщо попередня спроба розгортання була невдалою через тимчасові проблеми. Для цього можна використати ту саму команду *terraform* *apply* з файлом плану змін, який було згенеровано під час невдалої спроби розгортання: *terraform* *apply tfplan*. Це дозволить спробувати застосувати ті самі зміни ще раз, без необхідності їх повторної генерації.

Для забезпечення можливості відкату та *rollforward* необхідно належним чином організувати зберігання файлів стану та планів змін. Це можна зробити, використовуючи віддалені сховища стану (*remote state backends*), такі як *Terraform* Cloud, *AWS S3, Azure Blob Storage* тощо. Також рекомендується зберігати файли стану та плани змін як артефакти в CI/CD конвеєрах.

Наявність зворотних засобів *rollback* та *rollforward* у CI/CD процесах з IaC інструментами забезпечує більшу гнучкість, стійкість до помилок та можливість швидкого відновлення у разі виникнення проблем під час розгортання або оновлення інфраструктури.

## 3.2.10 Гібридні середовища та різні хмарні провайдери

Розглянемо питання підтримки гібридних середовищ та різних хмарних провайдерів у CI/CD процесах з інструментами IaC, такими як *Terraform*.

1. CI/CD конвеєри мають підтримувати багатохмарність. Сучасна інфраструктура часто розгортається у різних хмарних середовищах або поєднує хмарні та локальні (*on-premise*) ресурси, утворюючи гібридну архітектуру. CI/CD процеси та IaC інструменти повинні бути здатними працювати з різними хмарними провайдерами, такими як AWS, *Azure*, GCP, а також з локальними платформами, наприклад, *VMware* або OpenStack. *Terraform* має вбудовану підтримку багатьох провайдерів, що дозволяє керувати інфраструктурою в різних середовищах за допомогою єдиного набору конфігурацій. У CI/CD конвеєрах можна визначати різні етапи або задачі для розгортання в різних хмарах або середовищах, використовуючи відповідні конфігурації *Terraform*.

2. Узгодження конфігурацій для різних платформ.

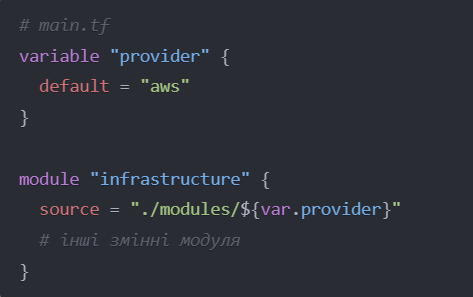
Хоча *Terraform* забезпечує уніфікований підхід до управління інфраструктурою, деякі ресурси та їх налаштування можуть відрізнятися між різними провайдерами або середовищами. Для вирішення цієї проблеми можна використовувати модульну структуру конфігурацій *Terraform* та умовну логіку на основі змінних. Наприклад, можна створити окремі модулі для ресурсів AWS та *Azure*, а в головній конфігурації визначити змінну для вибору провайдера та відповідного модуля. Альтернативно, можна використовувати умовні вирази *count* або *for\_each* для динамічного створення ресурсів залежно від вибраного провайдера або середовища.

Рис. 3.2.10.1 Приклад використання модулів для різних провайдерів

Тоді в директоріях *module*s/*aws* та *module*s/*azure* будуть знаходитись відповідні конфігурації для кожного провайдера. Крім того, у CI/CD конвеєрах можна визначати різні змінні середовища або використовувати файли змінних для різних середовищ розгортання.

Підтримка багатохмарності та узгодження конфігурацій для різних платформ є важливими аспектами для забезпечення гнучкості та масштабованості CI/CD процесів з IaC інструментами в сучасних гібридних ІТ-середовищах. Ретельне планування та врахування цих практичних аспектів забезпечить ефективну інтеграцію IaC інструментів з конвеєрами CI/CD, посилюючи переваги *DevOps* підходу.

## Висновки до розділу 3

У даному розділі було проаналізовано ключові аспекти та практичні підходи до інтеграції CI/CD процесів з інструментами IaC, такими як *Terraform* та *Ansible*, для забезпечення ефективного розгортання та управління інфраструктурою. Основні висновки можна сформулювати таким чином:

Для успішної реалізації CI/CD процесів з IaC необхідно встановити всі необхідні інструменти (*Git*, *Ansible*, *Terraform*) та налагодити інтеграцію із системою контролю версій і репозиторієм для зберігання та відстеження змін в інфраструктурному коді.

Використання робочих просторів (*workspaces*) в *Terraform* дозволяє ефективно розділяти та ізолювати тимчасові середовища для розгортання та тестування змін, забезпечуючи надійність та контрольоване управління інфраструктурою.

Параметризація конфігурацій за допомогою вхідних змінних та безпечне зберігання секретів у спеціалізованих сховищах (*Vault, Secrets Manager* тощо) забезпечує гнучкість, переносимість та безпеку під час розгортання інфраструктури в різних середовищах.

Впровадження статичного аналізу, юніт-тестів, інтеграційного тестування та тестування відповідності політикам за допомогою фреймворків, таких як *tflint*, *inspec, goss* та *terraform*-*compliance*, дозволяє підвищити якість та надійність IaC коду перед його розгортанням у продуктивні середовища.

Використання централізованих сховищ артефактів (*artifact repositories*) та незмінних артефактів інфраструктури (*immutable infrastructure artifacts*) забезпечує відтворюваність, стабільність та контроль версій під час розгортання інфраструктури.

Етапи схвалення змін (ручні або автоматизовані) та використання безпечних стратегій розгортання (*blue/green, canary, rolling update*) дозволяють мінімізувати ризики та забезпечити контрольоване впровадження змін в інфраструктурі.

Інтеграція з системами моніторингу та журналювання (*ELK Stack, Prometheus* тощо) забезпечує прозорість, відстеження подій, своєчасне виявлення проблем та полегшує усунення несправностей під час розгортання та експлуатації інфраструктури.

Можливість відкату змін (*rollback*) та перезастосування змін (*rollforward*) за допомогою зберігання попередніх станів інфраструктури та планів змін забезпечує гнучкість, стійкість до помилок та швидке відновлення у разі виникнення проблем.

Підтримка багатохмарності та гібридних середовищ, а також узгодження конфігурацій для різних платформ за допомогою модульної структури коду та умовної логіки забезпечують масштабованість та гнучкість CI/CD процесів з IaC інструментами в сучасних ІТ-середовищах. Впровадження цих підходів та практик у CI/CD процеси з інструментами IaC забезпечує ефективне, надійне та контрольоване розгортання інфраструктури, підвищує якість коду, мінімізує ризики та сприяє швидкому реагуванню на зміни та інциденти.

# Висновки

Отже, впровадження принципів CI/CD та інфраструктури як коду (IaC) в процесах розгортання ІТ-інфраструктури для Збройних Сил України є вкрай важливим для підвищення ефективності, надійності та безпеки.

Основні переваги даного підходу:

1. Скорочення часу на розгортання та оновлення систем за рахунок автоматизації процесів.

2. Підвищення стабільності та передбачуваності розгорнутого середовища завдяки систематичній перевірці змін.

3. Покращення безпеки шляхом швидкого реагування на виявлені загрози та вразливості.

4. Зменшення ризику людських помилок через зведення до мінімуму ручних операцій.

5. Централізоване управління конфігураціями та стандартизація середовищ.

6. Можливість швидкого відкату змін у разі виникнення проблем.

Для успішного впровадження необхідно провести навчання персоналу, інтегрувати відповідні інструменти та налаштувати безперервні конвеєри доставки для автоматизації розгортання. Також слід приділити особливу увагу забезпеченню безпеки на всіх етапах циклу.

Перехід до використання CI/CD та IaC дозволить Збройним Силам України підвищити загальну ефективність ІТ-операцій, скоротити витрати та забезпечити відповідність найсучаснішим практикам у галузі.

# Список використаних джерел

1. Refactoring: Improving the Design of Existing Code. 2-ге вид. Addison-Wesley Professional, 2019. 455 с. Signature Series (Fowler).
2. Gene Kim Patrick Debois John Willis. The DevOps Handbook: How to Create World-Class Agility, Reliability, and Security in Technology Organizations. IT Revolution Press, 2016. 480 с.
3. Michael Nygard. Release It!: Design and Deploy Production-Ready Software. 2-ге вид. Pragmatic Bookshelf, 2018. 378 с.
4. Craig Berg. DevOps For Beginners: A Complete Guide To DevOps Best Practices (Including How You Can Create World-Class Agility, Reliability, And Security In Technology ... With DevOps). 2020. 117 с.
5. Adam Wiggins. The Twelve-Factors App. [*https://12factor.net/*](https://12factor.net/).
6. Mykyta Protsenko. Infrastructure as Code: Базові принципи vs інструменти, що еволюціонують. [*https://dou.ua/lenta/articles/infrastructure-as-code/*](https://dou.ua/lenta/articles/infrastructure-as-code/)
7. Install Terraform | Terraform | HashiCorp Developer. *Install Terraform | Terraform | HashiCorp Developer*. URL: <https://developer.hashicorp.com/terraform/tutorials/aws-get-started/install-cli> (date of access: 14.05.2024).
8. Installing Ansible – Ansible Community Documentation. *Ansible Documentation*. URL: <https://docs.ansible.com/ansible/latest/installation_guide/intro_installation.html> (date of access: 14.05.2024).
9. Git - Downloads. *Git*. URL: <https://git-scm.com/download> (date of access: 14.05.2024).
10. Battle-tested code without the battle. *Speaker Deck*. URL: <https://speakerdeck.com/garethr/battle-tested-code-without-the-battle> (date of access: 14.05.2024).
11. https://visuresolutions.com/uk/blog/top-cicd-tools/

# Додатки

## Додаток 1 Спадна спіраль

Стовпчасту форму низхідної спіралі представлено у книжці The Phoenix Project, наведено нижче:

Таблиця 2 Низхідна спіраль

|  |  |
| --- | --- |
| Фахівці з експлуатації бачать, що … | Розробники бачать, що … |
| Вразливі застосунки схильні до відмов | Вразливі застосунки схильні до відмов |
| Необхідно багато часу, щоби з’ясувати, який біт був інвертований | Найбільш термінові проєкти і проєкти із чіткими дедлайнами чекають у черзі |
| Контроль та виявлення помилок здійснюються фахівцями із продажів | Навіть найбільш вразливий (менш захищений) код передається в експлуатацію |
| Дуже багато часу потрібно для відновлення сервісу | Зростає кількість релізів із хаотичним установленням |
| Занадто багато ліквідації недоробок та незапланованої роботи | Цикли релізів продовжуються, щоб амортизувати вартість розгортань |
| Потрібне термінове доопрацювання та усунення несправностей у системі безпеки | Збої масштабних розгортань значно складніше діагностувати |
| Запланована робота над проєктом не може бути завершена | Більшість провідних та відповідальних співробітників з експлуатації мають все менше часу для усунення фундаментальних проблем процесу |
| Розчаровані клієнти залишають компанію | Постійно зростає обсяг невиконаних робіт, що могли би принести бізнесу прибуток |
| Доля ринку зменшується | Постійно зростає напруга між експлуатацією, розробкою та проєктуванням |

## Додаток 2 Основи *Lean Thinking*

Чому масове виробництво не є найефективнішим способом створення продукту?

Уявіть, що вам потрібно надіслати клієнтам 10 листів, пройшовши наступні етапи:

* скласти 10 брошур в три рази;
* вставити їх у конверти;
* заклеїти конверти;
* штампувати конверти.

Як гадаєте який метод буде кращим із двох?

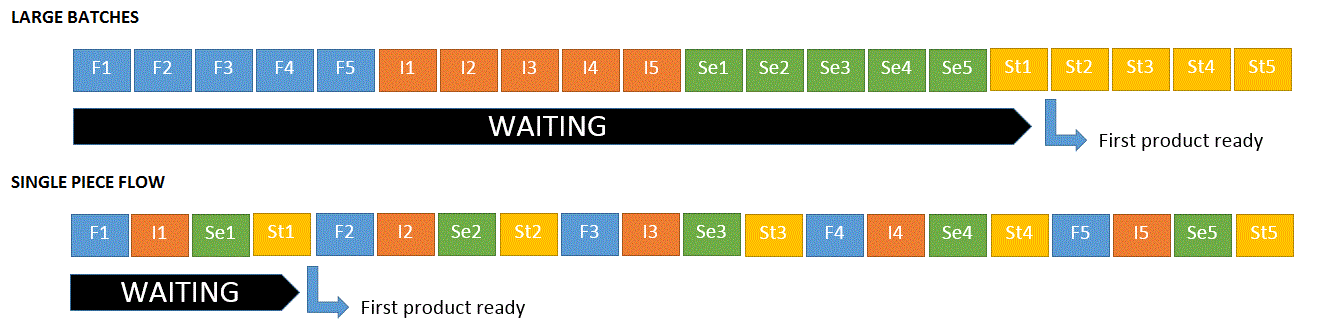
1) Завершити кожен крок (написаний зверху) окремо повністю: спершу скласти всі 10 брошур, потім вставити всі 10 у конверти, потім заклеїти всі 10 конвертів та проштампувати в кінці, коли інші кроки зроблено.

2) Скласти один лист в три рази, потім вставити його у конверт, заклеїти конверт, проштампувати його і після цього приступити до складання наступного листа.

Більшість людей обрали би перший підхід. Такий метод називається «велика партія» або «масове виробництво». Більшість людей будуть здивовані, дізнавшись, що другий підхід насправді кращий (у більшості ситуацій). Цей процес називається «окремий потік», який вперше використовувався Toyota і тому називався важливим елементом того, що пізніше стало «способом Toyota» робити речі. Сьогодні це називають «ощадливим виробництвом». Чому?

По-перше, це насправді швидше. Переконатися можна тут: [<https://blog.gembaacademy.com/2008/02/18/one-piece-flow-versus-mass-production/>]

По-друге, час очікування значно коротший у потоці складання однієї штуки. Клієнт може отримати перший готовий продукт на початку процесу. На рисунку нижче показані відмінності між 2ма процесами для листів:

Рис 1 Час очікування

По-третє, це зменшує запаси для зберігання незавершеної продукції не потрібно інвентаризації.

По-четверте, дуже вагомою перевагою є швидке виявлення дефектів. У рамках підходу до потоку однієї частини ви зможете виявити проблему під час завершення першого продукту.

По-п’яте, це підвищує зворотний зв’язок. Припустимо, що клієнту не подобаються брошури. Використовуючи великі партії, вам доведеться викинути не лише брошури, але й усі конверти, оскільки їх неможливо знову відкрити, не пошкодивши. Це призведе не тільки до збільшення відходів (також до викидання конвертів), але й до збільшення часу очікування, оскільки все доведеться переробляти, перш ніж клієнт отримає нові продукти.

Усі ці елементи призводять до наступних найкращих практик майже для будь-якого виробничого чи офісного середовища.

## Додаток 3 Досвід команди *Google Web Server*

Хоча сьогодні компанія *Google*, безсумнівно, є прикладом внутрішньої виробничої культури і належним чином цінує автоматизоване тестування, але так було не завжди. Майк Бленд, котрий долучився до команди GWS (*Google Web Server*), пояснює: «Команда GWS потрапила в ситуацію, коли їм було надзвичайно важко внести зміни у вебсервер – застосунок, створений на C++, що обробляв усі запити до головної сторінки google.com і багатьох інших вебсторінок сайту. При всій важливості і популярності, робота у складі команди GWS була аж ніяк не гламурним заняттям. Найчастіше вона нагадувала копирсання на звалищі коду, що реалізовував різні функції, написані командами, які працювали незалежно одна від одної. Розробники стикалися з такими проблемами, як занадто тривалі збірки й тестування, запуск у виробництво непротестованого коду, запис змін код коду, що відбувався лише зрідка, причому ці зміни конфліктували зі змінами, внесеними іншими командами».

Наслідки цього були серйозними: результати пошуку могли містити помилки, або сам пошук був неприйнятно повільним, а це впливало на тисячі пошукових запитів на сайті. Потенційний результат – втрата не лише заробітку, але і довіри споживача.

Бленд описує, як це вплинуло на розробників, які впроваджували зміни: «Страх став убивцею мислення. Страх перед змінами зупиняв нових членів команди, бо вони не розуміли як працює система. Але страх зупиняв також і досвідчених робітників, оскільки вони дуже добре розуміли потенційні наслідки».

Бленд був членом тієї відчайдушної групи, що намагалася розв’язати проблему. Водночас керівник команди GWS Бхарат Медіратта вважав, що автоматизоване тестування допоможе розв’язати проблему. Бленд писав: «Вони зайняли жорстку позицію: зміни не будуть прийняті у GWS, не пройшовши автоматизоване тестування. Вони створили безперервну збірку і буквально із релігійним завзяттям дотримувалися цього правила. Вони організували відстеження рівня тестового покриття і забезпечили постійне його зростання з плином часу. Вони написали керівництву з політики і тестування, наполягаючи, щоби усі учасники, пов’язані із цими процесами як усередині команди, так і поза нею, суворо дотримувалися встановлених правил».

Результати виявилися вражаючими. Бленд зазначав: «GWS швидко стала однією з найпродуктивніших команд у компанії. Вона щотижня виконувала інтеграційну збірку великої кількості змін, що надходили від різних команд, дотримуючись при цьому графіка швидких релізів. Відтепер нові члени команди майже відразу починали плідно працювати, вносячи вклад у складну систему завдяки вдалому покриттю коду тестами і його якості. Зрештою така радикальна політика дозволила головній сторінці сайту швидко розширити можливості та досягти успіху у стрімко змінюваному світі конкуруючи технологій».

GWS була всього-на-всього відносно невеликою командою у великій зростаючій компанії, але вони прагнули поширити застосовані ними методи на всю організацію. Тому на світ з’явилася «*Testing Grouplet*» - неофіційна група інженерів, які обрали своєю метою поширення автоматизованого тестування. Протягом наступних п’яти років вони допомогли розповсюдити культуру автоматичного тестування на всі без винятків підрозділи компанії *Google*.

Зараз, коли будь-який розробник із компанії *Google* виконує запис змін коду, негайно запуск набір із сотень тисяч автоматизованих тестів. Якщо код проходить перевірку, то він автоматично включається в основну гілку і виявляється готовим до розгортання у виробничому середовищі. Багато продуктів *Google* (наприклад, через глобальні зміни в інфраструктурі або якщо дефект внесений до однієї з основних бібліотек, від яких залежить кожна програма).

Результати, отримані Майком Бренданом і командою *Testing Grouplet*, зробили компанію *Google* однією з найпродуктивніших технологічних організацій у світі. Весь код зберігається в одному репозиторії, він складається з мільярд файлів, і вони постійно використовуються для складання та інтеграції, причому щомісяця код оновлюється на 50%.

Деякі інші статистичні дані щодо продуктивності фахівців Google виглядають вельми переконливо:

* 40000 записів змін коду на день;
* 50000 збірок на день (деякі дні це може бути і 90000);
* 120000 наборів автоматизованих тестів;
* 75 мільйонів тестів виконується щодня;