

МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ  
НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ «ЗАПОРІЗЬКА ПОЛІТЕХНІКА»

Факультет комп'ютерних наук та технологій  
Кафедра комп'ютерних систем та мереж

## Пояснювальна записка

до дипломного проєкту (роботи)

магістра

(ступінь вищої освіти)

на тему СИСТЕМА РОЗГОРТАННЯ ЗАСТОСУНКІВ У ХМАРІ З  
ВИКОРИСТАННЯМ КОНТЕЙНЕРНИХ ТЕХНОЛОГІЙ

Виконав: студент 2 курсу, групи КНТ-522м  
спеціальності \_\_\_\_\_

123 Комп'ютерна інженерія

(код і найменування спеціальності)

Освітня програма (спеціалізація)

Комп'ютерні системи та мережі

СМІРНОВ В.В

(ПРИЗВИЩЕ та ініціали)

Керівник КИРИЧЕК Г.Г.

(ПРИЗВИЩЕ та ініціали)

Рецензент СТЕПАНЕНКО О.О.

(ПРИЗВИЩЕ та ініціали)

МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ  
**Національний університет «Запорізька політехніка»**

Факультет Комп'ютерних наук і технологій

Кафедра «Комп'ютерні системи та мережі»

Ступінь вищої освіти магістерський

Спеціальність 123 Комп'ютерна інженерія

(код і найменування)

Освітня програма (спеціалізація) «Комп'ютерні системи та мережі»

(назва освітньої програми (спеціалізації))

**ЗАТВЕРДЖУЮ**

Зав. кафедри Кудерметов Р.К.

“24” жовтня 2023 року

**З А В Д А Н Н Я**  
**НА ДИПЛОМНИЙ ПРОЄКТ (РОБОТУ) СТУДЕНТА**

СМІРНОВА Владислава Вікторівна

(ПРИЗВИЩЕ, ім'я, по батькові)

1. Тема проекту (роботи) Система розгортання застосунків у хмарі з використанням контейнерних технологій

керівник проекту (роботи) КИРИЧЕК Галина Григорівна, к. т. н., доцент

(науковий ступінь, вчене звання, ПРИЗВИЩЕ, ім'я, по батькові)

затверджені наказом вищого навчального закладу від “24” жовтня 2023 року №400

2. Строк подання студентом проекту (роботи) 10 грудня 2023 року

3. Вихідні дані до проекту (роботи) технології та основні модулі і функції системи розгортання застосунків у хмарі з використанням контейнерних технологій, методи алгоритми та моделі; платформа Google Cloud у системі Google Kubernetes Engine

4. Зміст розрахунково-пояснювальної записки (перелік питань, які потрібно розробити) Аналіз завдання та дослідження інструментів розгортання програм. Проектування розгортання у хмарі з контейнерними технологіями. Впровадження системи розгортання застосунків у хмарі. Тестування системи розгортання застосунків у хмарі.

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

6. Консультанти розділів проекту (роботи)

| Розділ        | Прізвище, ініціали та посада консультанта | Підпис, дата                                                                        |                                                                                     |
|---------------|-------------------------------------------|-------------------------------------------------------------------------------------|-------------------------------------------------------------------------------------|
|               |                                           | завдання видав                                                                      | прийняв виконане завдання                                                           |
| 1-4           | КИРИЧЕК Г. Г., к. т. н., доцент           |  |  |
| нормоконтроль | ПОЛЬСЬКА О.В., ст. викл.                  |  |  |
|               |                                           |                                                                                     |                                                                                     |
|               |                                           |                                                                                     |                                                                                     |
|               |                                           |                                                                                     |                                                                                     |
|               |                                           |                                                                                     |                                                                                     |

7. Дата видачі завдання 01.10.2023 р.

**КАЛЕНДАРНИЙ ПЛАН**

| № з/п | Назва етапів дипломного проекту (роботи)                                                                                    | Строк виконання етапів проекту (роботи) | Примітка |
|-------|-----------------------------------------------------------------------------------------------------------------------------|-----------------------------------------|----------|
| 1     | Аналіз наявних систем для розгортання програм їх тестування, перегляд отриманих результатів, виділення недоліків та переваг | 10.10.2023 р.                           |          |
| 2     | Побудова функціональної моделі та алгоритму роботи системи                                                                  | 15.10.2023 р.                           |          |
| 3     | Реалізація системи розгортання                                                                                              | 20.10.2023 р.                           |          |
| 4     | Тестування та дослідження системи                                                                                           | 01.10.2023 р.                           |          |
| 5     | Оформлення отриманих результатів у ПЗ                                                                                       | 10.11.2023 р.                           |          |
| 6     | Оформлення графічного матеріалу                                                                                             | 25.11.2023 р.                           |          |
| 7     | Оформлення допоміжного матеріалу                                                                                            | 10.12.2023 р.                           |          |
|       |                                                                                                                             |                                         |          |
|       |                                                                                                                             |                                         |          |
|       |                                                                                                                             |                                         |          |

Студент  Владислав СМІРНОВ  
(підпис) (ім'я, ПРІЗВИЩЕ)

Керівник проекту (роботи)  Галина КИРИЧЕК  
(підпис) (ім'я, ПРІЗВИЩЕ)

## РЕФЕРАТ

ПЗ: 96 с., 22 рис., 8 табл., 42 джерел.

МІКРОСЕРВІСИ, KUBERNETES, TERRAFORM, TEKTON,  
КОНТЕЙНЕРИ, КЛАСТЕРИ, РОЗГОРТАННЯ.

Об'єктом дослідження є процес застосування технології для розгортання програм у контейнері на суперкомп'ютері.

Предметом дослідження є моделі та методи контейнеризації для реалізації системи розгортання застосунків у хмарі з використанням контейнерних технологій.

Метою роботи є проведення досліджень та застосування технології для розгортання програм у контейнері на суперкомп'ютері НРС (High performance computing) за допомогою Kubernetes, при реалізації системи розгортання застосунків у хмарі з використанням контейнерних технологій, враховуючи ефективність, ресурсні витрати, безпеку та відмовостійкість.

Розгортання проводиться на базі платформи Google Cloud у системі Google Kubernetes Engine.

При реалізації системи розгортання веб-додатків та проведення тестування, досліджено продуктивність контейнерного розгортання у хмарному середовищі суперкомп'ютера.

## **ABSTRACT**

Explanatory note to the master's work: 96 p., 22 figures, 8 tables, 42 sources.

MICROSERVICES, KUBERNETES, TERRAFORM, TEKTON,  
CONTAINERS, CLUSTERS, DEPLOYMENT.

The object of research is the process of using technology to deploy programs in a container on a supercomputer.

The subject of the study is the models and methods of containerization for the implementation of the application deployment system in the cloud using container technologies.

The aim of the work is to conduct research and apply technology for deploying programs in a container on a HPC (High performance computing) supercomputer using Kubernetes, when implementing an application deployment system in the cloud using container technologies, taking into account efficiency, resource costs, security and fault tolerance.

The deployment is based on the Google Cloud platform in the Google Kubernetes Engine system.

When implementing the system for deploying web applications and conducting testing, the performance of containerized deployment in the cloud environment of a supercomputer is investigated.

## ЗМІСТ

|                                                                            |    |
|----------------------------------------------------------------------------|----|
| Вступ.....                                                                 | 7  |
| 1 Аналіз завдання та дослідження інструментів розгортання програм .....    | 8  |
| 1.1 Основні поняття та визначення розгортання застосунку у хмарі.....      | 8  |
| 1.2 Актуальність контейнеризації для розгортання застосунків у хмарі ..... | 13 |
| 1.3 Постановка завдань .....                                               | 26 |
| 2 Проектування розгортання у хмарі з контейнерними технологіями.....       | 29 |
| 2.1 Аналіз критеріїв інструменту для розгортання застосунків у хмарі.....  | 29 |
| 2.2 Архітектура системи .....                                              | 48 |
| 3 Впровадження системи розгортання застосунків у хмарі.....                | 53 |
| 3.1 Підготовка середовища та розгортання компонентів системи .....         | 53 |
| 4 Тестування системи розгортання застосунків у хмарі .....                 | 76 |
| 4.1 Стратегії тестування системи .....                                     | 76 |
| 4.2 Дослідження методів тестування системи .....                           | 83 |
| 4.3 Тестування системи.....                                                | 86 |
| Висновки.....                                                              | 91 |
| Перелік джерел посилання .....                                             | 93 |

## ВСТУП

Хмарні обчислення стають популярнішими в сучасному світі, оскільки вони пропонують свої переваги для різних галузей як послуга. Завдяки хмарним обчисленням користувачі з різних країн можуть отримувати доступ до інформації та ресурсів через мережу, таким чином хмарні системи сприяють вдосконаленню операцій для зменшення різнопланових витрати, що дозволяє підвищити ефективність та цінність послуг для отримання та обробки інформації завдяки об'єднанню та доступності ресурсів.

Таким чином, ми розглянемо метод розгортання програм у хмарі за допомогою методу контейнеризації, коли за монолітний стосунок ділиться на мікросервіси, так кожний сервіс розміщується у окремому контейнері.

Актуальність вирішення цього питання, є наслідком розвитку використання гнучких методологій розгортання програмного забезпечення, основним завданням яких є оптимізація виробничого процесу та мінімізація ризиків шляхом зведення процесів до серії коротких циклів, забезпечити захист кожного мікросервісу, і як наслідок унеможливити знищення всього застосунку.

У першому розділі проведено аналіз розгортання програмного забезпечення на базі віртуальної машини, описані передумови та представлено обґрунтування необхідності розгортання застосунку у хмарі за допомогою методу контейнеризації.

У другому розділі пропонується архітектура системи розгортання застосунків у хмарі та розглядаються технічні рішення, які будуть використовуватися при її реалізації.

У третьому розділі описується процес підготовки середовища для впровадження системи, а також установка та налаштування її компонентів.

У четвертому розділі розробляються тестові сценарії для перевірки функціональності системи та надаються результати тестування.

## **1 АНАЛІЗ ЗАВДАННЯ ТА ДОСЛІДЖЕННЯ ІНСТРУМЕНТІВ РОЗГОРТАННЯ ПРОГРАМ**

### **1.1 Основні поняття та визначення розгортання застосунку у хмарі**

Сучасний інформаційний ландшафт надзвичайно динамічний та конкурентоспроможний. Підприємства та організації повинні пристосовуватися до змін, забезпечувати доступність своїх сервісів і залишатися вперед за технологічними інноваціями. Одним із ключових підходів до досягнення цих цілей є розгортання застосунків у хмарі. Розгортання застосунків у хмарі є важливою частиною процесу управління хмарними ресурсами для хмарних обчислень [1].

Як власне поняття системи розгортання застосунків у хмарі для контейнеризованих застосунків – ми можемо розуміти, як набір програмних компонентів і інструментів, які дозволяють організаціям розгортати контейнеризовані застосунки в хмарі.

Самі контейнеризовані застосунки – це застосунки, які упаковані разом з усіма їхніми залежностями в один автономний одиничний об'єкт, який називається контейнером. Контейнери можуть запускатися на будь-якій хмарній інфраструктурі, яка підтримує контейнеризовані технології.

Як ми можемо розуміти, то система як сама по собі, складається з декілької компонентів, які цю систему і формують. Системи розгортання застосунків у хмарі для контейнеризованих застосунків зазвичай включають такі основні компоненти:

- контейнерний менеджер – це програмний компонент, який керує контейнерами. Контейнерний менеджер відповідає за запуск, зупинку, перезапуск та масштабування контейнерів;
- контейнерний реєстр – це сховище контейнерів. Контейнерний реєстр дозволяє зберігати та керувати контейнерами;
- скрипти розгортання – це скрипти, які автоматизують процес розгортання

застосунків у хмарі.

Ну і щодо переваг таких систем, вони мають ряд позитивних функцій, такі як:

- економія витрат – контейнеризовані технології можуть допомогти організаціям заощадити на витратах на обладнання, програмне забезпечення та технічне обслуговування;

- еластичність – контейнеризовані технології дозволяють легко масштабувати застосунки вгору або вниз відповідно до потреб бізнесу;

- доступність – контейнеризовані технології можуть забезпечити високий рівень доступності за стосунків;

- безпека – контейнеризовані технології можуть забезпечити високий рівень безпеки за стосунків завдяки принципам ізоляції, відстеження змін, автоматизації.

Тобто, ми можемо розуміти, що системи розгортання застосунків у хмарі для контейнеризованих застосунків можуть використовуватися для розгортання будь-якого типу застосунків, але вони особливо підходять для розгортання веб-застосунків, мобільних застосунків та мікросервісів.

Виходячи з цього, ми можемо підвести деякі приклади компонентів систем для розгортання застосунків у хмарі для контейнеризованих застосунків:

- kubernetes – це відкрита платформа для автоматизації розгортання, масштабування та управління контейнеризованими застосунками;

- docker Swarm – це контейнерний менеджер, який входить до складу Docker Engine;

- terraform – це інструмент для управління інфраструктурою як код, який використовує декларативну мову конфігурації для створення, оновлення та видалення ресурсів хмарної інфраструктури;

- tekton CI/CD – це інструмент для автоматизації CI/CD в хмарі, який використовує декларативну мову конфігурації. Він використовує декларативну мову конфігурації для створення пайплайнів CI/CD, які можна масштабувати та адаптувати до різних потреб.

Ми можемо описати на рисунку 1.1 загальний алгоритм для розгортання такої системи.

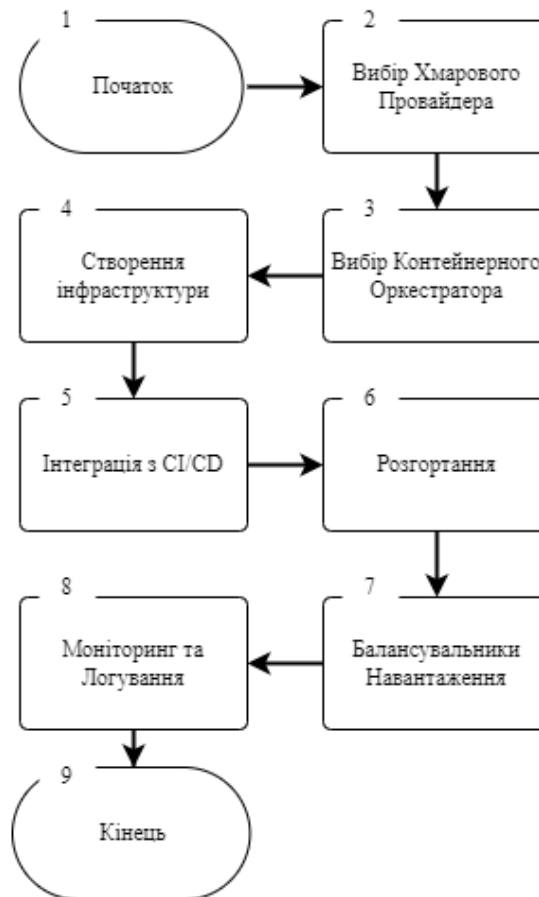


Рисунок 1.1 – Загальний алгоритм розгортання системи

Вибір системи розгортання застосунків у хмарі для контейнеризованих застосунків залежить від конкретних потреб організації [2].

Але для комплексного розуміння сутності розгортання застосунків у хмарі необхідно ознайомитися з деякими основними поняттями такої концепції.

Технологія хмарних обчислень є частиною великомасштабних розподілених обчислень, яка використовує існуючі технології, такі як віртуалізація, сервіс-орієнтація та ґрід-обчислення. Вона пропонує інший спосіб придбання та управління ІТ-ресурсами у великих масштабах. Простим прикладом хмарних обчислень є веб-пошта. Провайдер веб-пошти підтримує простір на сервері та надає доступ; Користувач веб-пошти просто підключає веб-

адресу до браузера та надсилає інформацію про користувача для доступу до облікового запису [1].

Розгортання застосунків у хмарі, передбачає використання інтерфейсів і стандартів, які сприяють взаємодії між різними складовими системи, розташованими в хмарі. Це означає, що програмні компоненти повинні бути розроблені таким чином, щоб вони могли легко обмінюватися даними та ресурсами через мережу. Важливим аспектом цього підходу є створення модульних та незалежних компонентів, які можуть бути масштабовані і розгорнуті окремо [2].

Хмарна інфраструктура є апаратною та програмною основою, яка надає обчислювальні та мережеві ресурси для розгортання програм у хмарі. Масштабування - здатність збільшувати або зменшувати ресурси залежно від навантаження для забезпечення високої продуктивності [3].

Балансування завантаження є методологією, яка використовується в інформаційних системах та комп'ютерних мережах з метою розподілу запитів, завдань та трафіку між різними серверами або ресурсами з метою досягнення найкращої продуктивності, надійності та доступності системи [3]. А віртуалізація є технологією, яка дозволяє створювати віртуальні версії обчислювальних, мережевих, та інших ресурсів замість фізичних апаратних компонентів. Ця технологія створює абстракцію між апаратними ресурсами та програмними засобами або сервісами, які їх використовують [4].

Контейнеризація в хмарі - це процес розгортання та управління контейнерами в хмарному середовищі. Контейнери є самодостатніми віртуальними середовищами, які містять додатки та їх залежності, що дозволяє їм працювати незалежно. Цей підхід надає розробникам і адміністраторам можливість ефективно створювати, розгортати і керувати контейнерами для різних хмарних додатків і сервісів, забезпечуючи гнучкість та оптимальне використання ресурсів. Крім того, такий підхід сприяє швидкому масштабуванню додатків і забезпечує їхню єдність у різних середовищах хмарного обчислення [5].

Щодо необхідності та актуальності розгортання застосунків у хмарі можна сказати, що це надає гнучкість, розгортання застосунків у хмарі та дозволяє організаціям швидко реагувати на зміни в обсягах роботи та вимог до ресурсів. Компоненти можуть бути легко гнучко масштабовані. Використання розгортання за стосунків у хмарі дозволяє ефективно використовувати ресурси, оскільки компоненти можуть бути розміщені на тих серверах, які найкраще відповідають їхнім потребам. Це може значно зменшити витрати на обладнання та підтримку. Хмарні постачальники часто мають розгалужені мережі та множину дата-центрів, що дозволяє забезпечити високий рівень доступності та надійності послуг. Розгортання за стосунків налаштовується таким чином, щоб забезпечити відмовостійкість і резервне копіювання даних.

Отже, підсумовуючи можна сказати, що впровадження додатків в хмарному середовищі є критичною стратегією в сучасному інформаційному середовищі. Цей підхід вирішує кілька важливих завдань і цілей. Розгортання в хмарі надає організаціям можливість оперативно реагувати на зміни та залишатися на передовому рівні в галузі технологічних досягнень. Іншою важливою потребою є забезпечення доступності послуг користувачам з будь-якої локації і в будь-який час. Хмарні ресурси надають гнучкість і зручність, необхідні для досягнення цього процесу. Крім того, розгортання додатків в хмарі дозволяє масштабувати ресурси відповідно до попиту, забезпечуючи високу продуктивність та ефективне використання ресурсів. Балансування навантаження також є критичним фактором, включаючи рівномірний розподіл завдань і трафіку для досягнення оптимальної продуктивності системи та доступності. Застосування віртуалізації та контейнеризації в хмарі створює рівень абстракції між апаратними ресурсами та програмними компонентами, спрощуючи управління ресурсами та підвищуючи їхню ефективність. Усі ці фактори загалом підкреслюють важливість впровадження додатків в хмарі як фундаментальної стратегії для сучасних організацій, які прагнуть зберегти конкурентоспроможність та адаптивність в умовах постійних змін та зростаючих вимог інформаційного ринку.

## 1.2 Актуальність контейнеризації для розгортання застосунків у хмарі

Хмарні обчислення вимагають високої портативності. Контейнеризація забезпечує сумісність середовища, інкапсулюючи додатки разом із бібліотеками, конфігураційними файлами та іншими залежностями. Додатки легко переміщуються та розгортаються між кластерами. Контейнеризація, як технологія віртуалізації, замість імітації повної операційної системи, як це робить віртуальна машина, застосовує контейнери. Вони використовують тільки операційну систему хоста, і ця особливість робить контейнери легшими за віртуальні машини.

Аналізуючи деякі дослідження та публікації, можна виявити деякі цікаві пояснення концепції. Андрю Йонг у своїй доповіді «A Tale of Two Systems: Using Containers to Deploy HPC Applications on Supercomputers and Clouds» [6] розглядає застосування контейнерних технологій для розгортання програм на суперкомп'ютерах. Контейнеризація представляє собою нову стратегію управління програмним забезпеченням у розподілених системах, яка дозволяє розробникам зменшити зусилля, необхідні для розгортання програмних додатків на різних обчислювальних платформах. Контейнери надають можливість точно визначити конфігурацію робочого середовища для програмного коду, включаючи операційну систему, бібліотеки, змінні середовища та залежності. В галузі високопродуктивних обчислень постійно зростає розмаїття апаратних засобів та системних архітектур, що ускладнює розгортання нових додатків. Розробники потребують тестування, налагодження та оптимізації продуктивності для різних HPC-платформ [3]. Крім того, черги на використання потужних суперкомп'ютерів можуть затримувати розробку коду та ускладнювати тестування. Це змушує розробників шукати маломасштабні тестові середовища або використовувати зовнішні ресурси для розробки та експлуатації високопродуктивних обчислень, тому застосування контейнерів на суперкомп'ютерах є однією з потенційних областей досліджень.

У дослідженні «Container orchestration on HPC systems through Kubernetes» [7], зазначається важливість Kubernetes із системою керування ресурсами для контролю та надсилання завдань на суперкомп'ютери, кластери та мережі у хмарному кластері HPC. Завдання з великим обсягом обчислень чи даних отримують покращену продуктивність завдяки ефективному плануванню в хмарному кластері. Вміст вхідного вузла зазвичай представлено у звичайному YAML-форматі [9] Kubernetes замість використання PBS-скрипту TORQUE. При цьому оркестратор може аналізувати розмір завдання, щоб визначити, чи слід його виконувати на HPC-кластерах та для відправки завдання на HPC, автоматично генерується відповідний вбудований YAML-файл. Kubernetes має гнучку модульну архітектуру, яка абстрагує базову інфраструктуру і дозволяє внутрішнє налаштування. Він підтримує різні фреймворки для обробки великих обсягів даних, такі як Hadoop MapReduce [10], Apache Spark [11], і може інтегруватися з Ansible [4] - популярним інструментом для оркестрування програмного забезпечення в хмарних кластерах. Kubernetes має потужний набір інструментів для керування життєвим циклом додатків, включаючи автоматичне перерозгортання у разі відмови та управління станом, крім того в нього є розширена система планування, яка дозволяє вказувати різні планувальники для кожного завдання.

Хмарні обчислення вимагають високої портативності. Контейнеризація забезпечує сумісність середовища, інкапсулюючи додатки разом із бібліотеками, конфігураційними файлами та іншими залежностями. Додатки легко переміщуються та розгортаються між кластерами. Контейнеризація, як технологія віртуалізації, замість імітації повної операційної системи, як це робить віртуальна машина, застосовує контейнери. Вони використовують тільки операційну систему хоста, і ця особливість робить контейнери легшими за віртуальні машини [12].

Виходячи з цього, ми можемо підкреслити загальні переваги:

- масштабованість та гнучкість;
- безпека;

- зменшення витрат на підтримку інфраструктури;
- забезпечення близькості до користувача;
- швидкість розгортання;
- висока доступність;
- не потрібні стартові витрати.

Актуальність контейнерів полягає у тому, що вони призначені для запуску мікросервісів і в одному контейнері, зазвичай, розміщується один додаток, однак додатки з контейнерами можуть бути і складними. Наприклад, у виробництві виникають ситуації коли потрібні сотні окремих контейнерів, тому контейнерні оркестратори повинні забезпечити ефективне надання ресурсів і автоматичне масштабування [13]. Але контейнерна методологія розгортання програм з'явилася не одразу, передумовою стало використання програм із застосуванням віртуальних машин, на рисунку 1.2 відобразимо які вони мали свої переваги та недоліки.

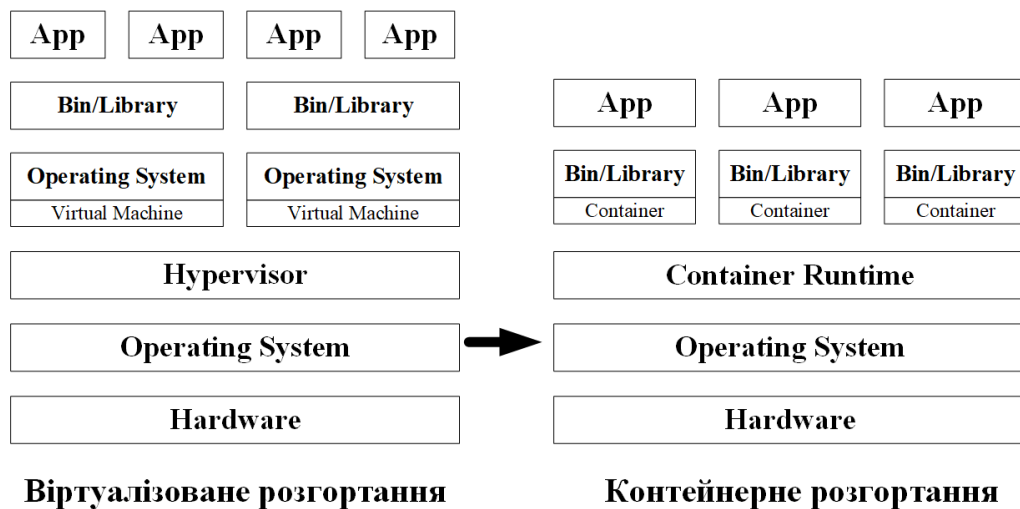


Рисунок 1.2 – Еволюція методів розгортання програм

Розглянемо підхід розгортання програм у віртуальних машинах [14]. Ця концепція дозволяє працювати декільком віртуальним машинам на одному фізичному центральному сервері та запускати багато програм на одній фізичній машині, дозволяючи ізолювати їх при запуску в різних операційних системах. Віртуальні машини є менш ефективними з точки зору розміру та продуктивності.

Контейнери схожі на віртуальні машини, але є більш легким варіантом ефективного використання ресурсів, швидкого розгортання, ізолюваності, портабельності, масштабованості, оркестрації та керування. Контейнери є портабельними, їх можна переносити та запускати на різних платформах, які підтримують контейнеризацію, без необхідності зміни кодової бази та налаштувань, у той час, як віртуальні машини вимагають налаштувань та адаптації для кожної платформи окремо. Контейнери мають можливість автоматичного масштабування. Це дозволяє керувати кількістю екземплярів застосунку в залежності від навантажень і зберігати надвисоку продуктивність, у той час, як віртуальні машини вимагають створення та налаштування кожної машини окремо. Контейнери надають зручні інструменти для керування та оркестрації контейнеризованих застосунків. Система керування контейнерами Kubernetes забезпечує автоматизованість процесів, а віртуальні машини мають власну операційну систему та служби, які необхідно налаштовувати та оновлювати незалежно один від одного. Але контейнерний метод розгортання застосовуються виключно для мікросервісів, тоді як віртуальні машини підтримують монолітні програми, тому розгортання на них програм з парадигмою мікросервісів не є актуальним методом [6].

Таким чином хмарні системи сприяють вдосконаленню операцій для зменшення різнопланових витрат, що дозволяє підвищити ефективність та цінність послуг для отримання та обробки інформації завдяки об'єднанню та доступності ресурсів. Такі можливості нам надає програмне забезпечення, яке працює за принципами: SaaS (програмне забезпечення, як послуга); PaaS (платформа, як послуга) та IaaS (інфраструктура, як послуга). Хмарні обчислення проводяться на системах High-performance computing (HPC) [15], які мають розподілення навантаження, високу ефективність та швидкість обробки великих даних, де кожна секунда є важливою. Як раз такі можливості і надають суперкомп'ютери, що складаються з великої кількості процесорів, оперативної пам'яті, та сховища даних. Подібні великі апаратні можливості вимагають чіткої, структурованої та масштабованої системи управління. На сьогоднішній день

найбільшу популярність для застосування такого рішення набуває метод розгортання програм у контейнерах.

Нам необхідно більш детально визначити, що з себе представляють такі поняття, як SaaS (програмне забезпечення, як послуга) [16]; PaaS (платформа, як послуга) [17] та IaaS (інфраструктура, як послуга) [18] на прикладі хмарних середовищ.

SaaS або програмне забезпечення як послуга, є моделлю хмарного обслуговування, де постачальник хмарних послуг повністю надає та керує програмним забезпеченням, доступ до якого надається клієнтам. У цій моделі клієнти не мають прямого контролю над програмним забезпеченням, але можуть використовувати його за допомогою веб-інтерфейсу або мобільного додатку. Розглянемо детальніше. Забезпечення основної інфраструктури для контейнеризованих додатків може бути реалізоване через SaaS-сервіси хмарних обчислень, такі як Google Cloud Platform, Amazon Web Services або Microsoft Azure. Ці платформи надають віртуальні машини, мережі, сховища та інші ресурси, необхідні для ефективного розгортання контейнеризованих додатків. Надання інструментів та послуг для управління контейнеризованими додатками також може здійснюватися через SaaS-пропозиції, такі як Kubernetes або Docker Swarm. Ці сервіси надають засоби для розгортання, масштабування та управління контейнерами, забезпечуючи зручність і ефективність у роботі з контейнеризованими додатками. Також важливою частиною екосистеми є надання даних та послуг для використання контейнеризованими додатками. Наприклад, SaaS-сервіси з аналізу даних, зберігання даних або управління взаємовідносинами з клієнтами можуть постачати необхідні дані та сервіси для використання контейнерами у роботі [19].

PaaS, або платформа як сервіс, представляє собою модель, в якій клієнти мають можливість використовувати платформу для створення та впровадження своїх застосунків. У цьому підході клієнти зберігають контроль над процесом розробки та впровадження своїх застосунків, однак не мають прямого впливу на інфраструктуру, на якій ці застосунки працюють. Послуги PaaS відіграють

важливу роль у впровадженні контейнеризованих застосунків у хмарному середовищі. Ці послуги можуть бути використані для надання базової інфраструктури для контейнеризованих застосунків. PaaS-сервіси хмарних обчислень, такі як Google App Engine, Amazon Elastic Beanstalk або Microsoft Azure App Service, можуть постачати необхідні ресурси, включаючи віртуальні машини, мережі, сховища та інші, для ефективного розгортання контейнеризованих застосунків. Забезпечення інструментів та сервісів для управління контейнеризованими застосунками. PaaS-сервіси для керування контейнерами, такі як Google Kubernetes Engine, Amazon Elastic Container Service або Microsoft Azure Kubernetes Service, можуть надавати засоби для розгортання, масштабування та управління контейнеризованими застосунками. Забезпечення доступу до даних та сервісів, які можуть бути використані контейнеризованими застосунками. PaaS-сервіси для аналізу даних, зберігання даних або управління взаємодією з клієнтами можуть надавати необхідні дані та послуги для використання контейнеризованими застосунками [19].

IaaS, або інфраструктура як сервіс, є моделлю, що надає клієнтам можливість користуватися фізичними ресурсами, такими як сервери, мережі та сховища. У цій моделі клієнти мають повний контроль над використанням цих ресурсів, але вони також несуть відповідальність за їхнє управління та технічне обслуговування. IaaS-сервіси відіграють ключову роль у розгортанні контейнеризованих застосунків у хмарному середовищі. Ці послуги забезпечують базову інфраструктуру, де віртуальні машини, надані провайдерами, такими як Google Compute Engine, Amazon Elastic Compute Cloud (EC2) або Microsoft Azure Virtual Machines, використовуються для запуску контейнерів з встановленими застосунками. Конкретно, організація може використовувати IaaS-сервіси для хмарних обчислень та створювати віртуальні машини, на яких будуть запускатися контейнери з розгорнутими застосунками. Крім того, IaaS-сервіси для керування контейнерами, такі як Docker Swarm або Kubernetes, надають інструменти для автоматизації процесів розгортання, масштабування та управління контейнеризованими застосунками в хмарному

середовищі. Це відкриває можливості для ефективного та гнучкого використання ресурсів, при збереженні повного контролю над процесами розгортання та функціонуванням за стосунків [19].

Виходячи з цього ми можемо навести перелік переваг контейнерів над віртуальними машинами, це:

- ефективне використання ресурсів; швидке розгортання;
- ізоляція та безпека;
- портабельність та сумісність;
- гнучкість та масштабованість;
- латентність та еластичність і можливість створювати контейнери із ізольованим середовищем для тестування.

Визначившись з перевагою та необхідністю концепції розгортання методом контейнеризації, нам необхідно навести структуру подібної системи. Вона складається з таких компонентів:

- хмарна інфраструктура виступає як фізична основа для розгортання контейнерів, включаючи сервери, мережі та сховища;
- інструменти розгортання, це допоміжні інструменти для автоматизації розгортання;
- контейнери служать стандартизованими одиницями виконання, що охоплюють усі необхідні файли та бібліотеки для запуску додатків;
- реєстр контейнерів, це сервіс, призначений для зберігання та керування контейнерами;
- контролер розгортання відповідає за нагляд за процесом розгортання контейнерів.

Хмарна інфраструктура забезпечує основні фізичні елементи для розгортання контейнерів, включаючи сервери, мережі та сховища. Розробники та адміністратори можуть використовувати будь-яку хмарну інфраструктуру, що підтримує контейнери, для отримання доступу до цих ресурсів та ефективного управління розгортанням контейнеризованих застосунків. Це надає їм можливість легко масштабувати ресурси, забезпечуючи гнучкість та швидкість

реагування на зміни в обсязі робочих навантажень.

Інструменти розгортання допомагають розробникам та адміністраторам у процесі розгортання контейнерів і можуть включати скрипти, інструменти командного рядка та веб-інтерфейси.

Контейнери відіграють центральну роль у розгортанні додатків у хмарі за допомогою контейнеризації. Вони дають змогу розробникам та адміністраторам швидко та без зусиль створювати, розгортати та масштабувати додатки в хмарних середовищах. Контейнери, як стандартизовані одиниці виконання, можуть працювати на будь-якому хмарному або фізичному сервері, який підтримує контейнери, забезпечуючи розробникам та адміністраторам гнучкість та масштабованість.

Але у контейнерів повинна бути система зберігання. Система зберігання контейнерів повинна забезпечувати такі характеристики [20]:

- надійність і доступність - надійність системи зберігання контейнерів означає, що контейнери не будуть втрачені або пошкоджені. Доступність означає, що контейнери будуть доступні для розгортання та використання в будь-який час. Коли ми говоримо про високу доступність, то маємо на увазі час безвідмовної роботи (uptime) системи. Uptime - це відсоток часу, протягом якого система доступна для використання. Для системи зберігання контейнерів, яка повинна бути доступною 24/7, високою доступністю вважається uptime не менше 99,9%. Це означає, що система може бути недоступна не більше 43,2 хвилин на рік.

Для забезпечення надійності та доступності системи зберігання контейнерів можна використовувати такі методи та технології:

- реплікація контейнерів. Контейнери можна реплікувати на кілька вузлів, щоб забезпечити їх доступність у разі відмови одного з вузлів;
- резервне копіювання контейнерів. Контейнери можна регулярно резервувати, щоб мати можливість відновити їх у разі втрати;
- автоматичне відновлення. Система зберігання контейнерів повинна мати можливість автоматично відновлювати контейнери у разі відмови;

- ефективне використання дискового простору - контейнери можуть бути досить великими, тому система зберігання контейнерів повинна ефективно використовувати дисковий простір.

Для ефективного використання дискового простору системи зберігання контейнерів можна використовувати такі методи та технології:

- швидкий доступ до контейнерів - контейнери повинні бути швидко доступні для розгортання та використання.

Для забезпечення швидкого доступу до контейнерів системи зберігання контейнерів можна використовувати такі методи та технології:

- кешування контейнерів. Контейнери можна кешувати в пам'яті, щоб забезпечити швидкий доступ до них;

- розподіл навантаження. Контейнери можна розподіляти на кілька вузлів, щоб забезпечити рівномірний доступ до них.

Отже, система зберігання контейнерів повинна забезпечувати надійність, ефективне використання дискового простору та швидкий доступ до контейнерів. Крім того, важливо враховувати стратегії масштабування. Система зберігання повинна бути готовою до збільшення обсягу робочих навантажень. Використання горизонтального та вертикального масштабування дозволяє збільшувати обчислювальні та зберігальні потужності системи відповідно до потреб.

Реєстр контейнерів функціонує як централізований сервіс зберігання та керування контейнерами. Він дозволяє користувачам знаходити та використовувати контейнери, створені іншими, і може бути як загальнодоступним, як Docker Hub, так і приватним, як Amazon Elastic Container Registry, що пропонує безпечні параметри зберігання.

Контролер розгортання відповідає за керування всім процесом розгортання контейнерів. Він отримує деталі програми від розробників або адміністраторів і планує та виконує розгортання контейнерів на хмарних ресурсах.

Зобразимо загальну структуру на рисунку 1.3.

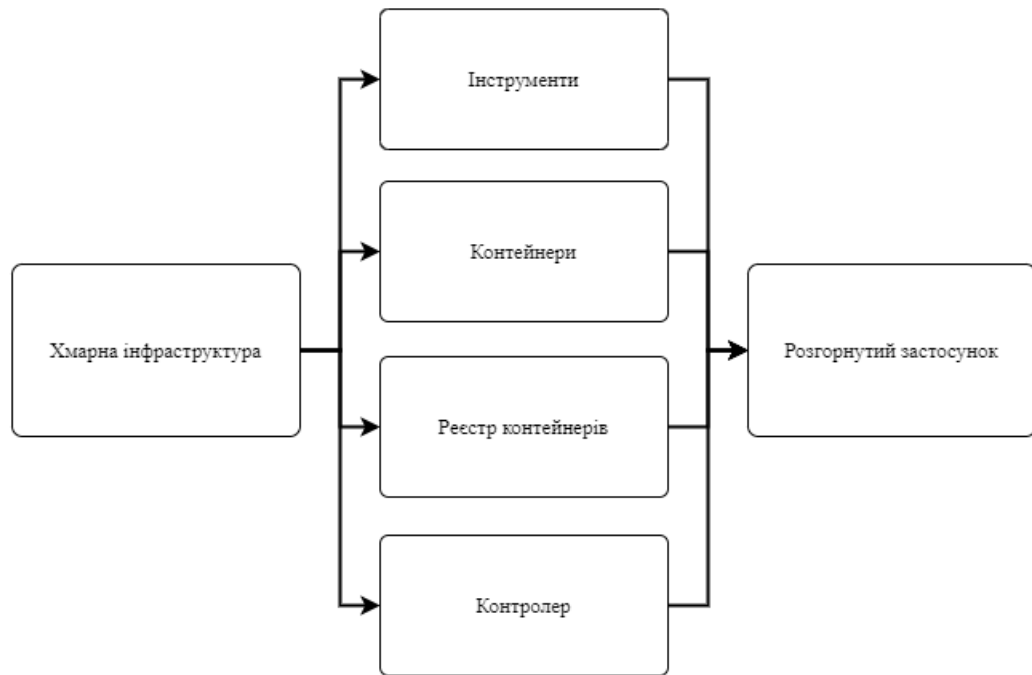


Рисунок 1.3 – Структура системи розгортання у контейнерах

На основі загальноприйнятої структури системи розгортання застосунків у хмарі методом контейнеризації ми можемо збудувати компоненти модулів, які у кінцевому результаті вибудовують цілісну систему [21, 22].

Кожен компонент відіграє важливу роль у забезпеченні успішного розгортання та функціонування застосунків. Давайте подивимося на ці модулі ближче, щоб зрозуміти, як вони працюють і які переваги вони пропонують:

- модуль керування контейнерами - відповідає за створення, запуск, зупинення та видалення контейнерів. Він також забезпечує інтерфейс для взаємодії з контейнерами, що дозволяє контролювати їхній стан і забезпечувати необхідні ресурси;

- модуль реєстру контейнерів - є сховищем для контейнерів, використовуваних в системі. Він може бути публічним або приватним. Публічні реєстри дозволяють розробникам спільно використовувати свої контейнери, тоді як приватні реєстри надають безпечне сховище для власних контейнерів;

- модуль контролера розгортання - керує процесом розгортання контейнерів на хмарних ресурсах. Він взаємодіє з іншими модулями, отримує інформацію про застосунок та визначає оптимальний спосіб розгортання

контейнерів;

- модуль інфраструктури хмарних ресурсів - забезпечує фізичну основу для розгортання контейнерів. Він включає в себе сервери, мережі та сховища, необхідні для виконання контейнерів. Розробники та адміністратори можуть вибирати хмарні платформи або сервіси, які найкраще відповідають їхнім потребам;

- модуль інструментів розгортання - надає засоби для розробників та адміністраторів для ефективного розгортання контейнерів. Ці інструменти можуть включати автоматизовані процеси, інтеграцію з іншими системами та зручний інтерфейс користувача;

- модуль безпеки та автентифікації - відповідає за забезпечення безпеки системи та контейнерів. Він включає механізми автентифікації для забезпечення доступу лише вповноваженим користувачам, контроль доступу до ресурсів та шифрування для захисту конфіденційної інформації;

- модуль моніторингу та журналювання - надає можливості для моніторингу стану контейнерів, ресурсів та операцій розгортання. Він включає в себе журналювання подій, збір статистики та візуалізацію даних для забезпечення ефективного моніторингу та виявлення проблем.

Така система модульності має як переваги, так і недоліки, понемо з переваг:

- чіткий розподіл обов'язків - кожен модуль бере на себе відповідальність за конкретний аспект процесу розгортання, встановлюючи чітке розділення завдань і відповідальності за кожен компонент. Це дозволяє розробникам зосереджуватися на розробці додатків, залишаючи адміністраторам управління інфраструктурою;

- модульність - гнучкість і масштабованість системи забезпечуються легкістю додавання, зміни або вилучення модулів. Ця адаптивність дозволяє системі легко пристосовуватися до змінних бізнес-потреб;

- безшовна інтеграція - модулі безперервно інтегруються з іншими системами та інструментами, сприяючи покращеній співпраці та взаємодії з

різними елементами системи. Ця інтеграція підвищує ефективність та продуктивність в цілому.

Цей структурований та модульний підхід до розгортання контейнеризованих систем у хмарі не лише забезпечує спрямований та відповідальний робочий процес, але також надає можливість адаптації до змінних потреб і ефективну інтеграцію з іншими компонентами системи, в кінці кінців сприяючи загальній підвищеній ефективності та продуктивності. Крім того, можливість ефективної інтеграції з іншими компонентами системи стає фундаментальною для створення комплексного та добре взаємодіючого інфраструктурного середовища.

Хоча архітектура системи розгортання контейнерів у хмарі має численні переваги, вона також включає деякі недоліки, які слід враховувати при виборі.

Одним з основних недоліків такої архітектури є можлива складність в адмініструванні. Система складається з декількох модулів, кожен з яких має власні функції та відповідальності. Ця складність може зробити систему важкою для розуміння та управління. Ще одним недоліком є те, що така архітектура може бути менш ефективною, ніж більш централізовані системи. Кожен модуль відповідає за певний аспект процесу розгортання, що може призвести до дублювання зусиль та неефективного використання ресурсів. Нарешті, така архітектура може бути менш гнучкою, ніж більш централізовані системи. Додавання нових функцій або зміна існуючих може бути складним і вимагати внесення змін до декількох модулів.

Отже, архітектура системи розгортання контейнерів у хмарі виявляється ефективним рішенням для розгортання та управління контейнеризованими застосунками. Її структурований та модульний підхід забезпечує спрямований робочий процес та можливість адаптації до змінних потреб, сприяючи загальній підвищеній ефективності та продуктивності. Проте, як будь-яка технологія, вона має свої переваги та недоліки.

Підсумовуючи, на рисунку 1.4 зобразимо загальну структуру модулів.

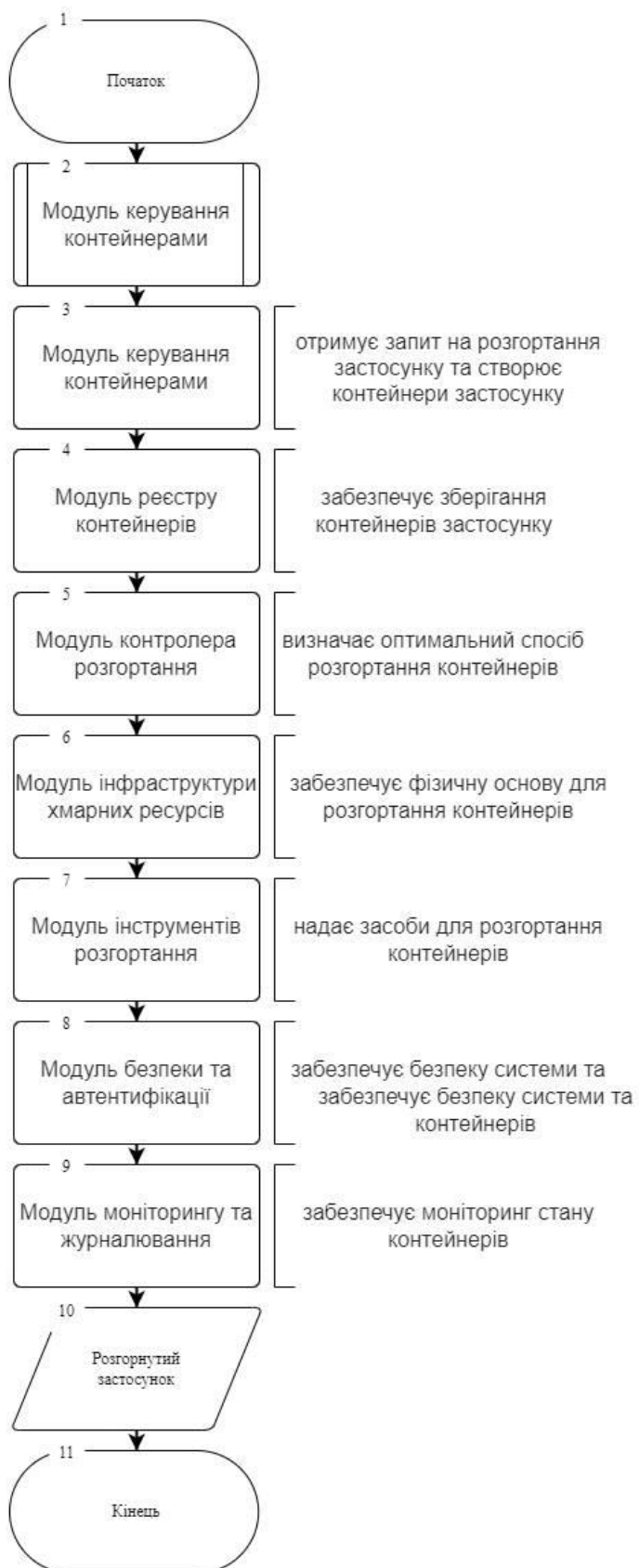


Рисунок 1.4 – Системи модулів розгортання застосунку

Варто зазначити, що така архітектура модулів може бути застосована на всіх хмарних платформах. Однак, деякі модулі можуть бути реалізовані по-різному в залежності від платформи. Наприклад, модуль реєстру контейнерів може бути реалізований як окремий сервіс або як частина іншої послуги, наприклад, служби керування контейнерами. Модуль управління контейнерами може приймати форму служби оркестрації, такої як Kubernetes або Docker Swarm. Реєстр контейнерів може бути виконаний як окремий сервіс, наприклад, Docker Hub або Amazon ECR. Контролер розгортання може бути частиною служби управління контейнерами або виступати як окрема функціональна одиниця. Модуль інфраструктури хмарних ресурсів може використовувати доступні ресурси у хмарному середовищі, такі як сервери, мережі та сховища. Інструменти розгортання можуть використовувати різні засоби для розгортання контейнерів, такі як інтерфейс командного рядка (CLI) або графічний інтерфейс користувача (GUI).

Отже, ми можемо дійти до висновку, що розгортання застосунку у хмарі підходить для розгортання мікросервісних архітектур, та має перевагу у своєму підході перед монолітним розгортанням у хмарі за допомогою віртуалізації завдяки чітко структурованій архітектурі модульності системи з розгортання застосунків у хмарі. Варто ще раз акцентувати на тому, що такий метод розгортання у хмарі може застосовуватись лише у мікросервісних архітектурах.

### **1.3 Постановка завдань**

Ми визначились яку хмарну платформу та які допоміжні інструменти ми будемо використовувати для розгортання застосунку у хмарі за методом контейнеризації. Хмарну платформу нам надає Google Cloud Platform, який має всю необхідну вбудовану інфраструктуру, у тому числі систему Google

Kubernetes Engine для будови контейнерів, та надає змогу інтеграції Terraform та Tekton CI/CD.

Terraform by HashiCorp надає можливість представляти нашу інфраструктуру за допомогою файлів конфігурації, що дозволяє автоматизувати розгортання та керування. За допомогою Terraform ми можемо описувати нашу інфраструктуру як код, визначаючи ресурси, їхні атрибути та взаємозв'язки. Підтримується різними постачальниками для різних хмарних платформ, включаючи Google Cloud Platform. Він має механізм планування, який попереджує про зміни в інфраструктурі перед їх застосуванням, забезпечуючи видимість змін. Інструмент відстежує поточний стан інфраструктури, що спрощує виявлення необхідних змін для досягнення бажаного стану. Крім того, він забезпечує надійне відновлення у разі збоїв. Також Terraform спрощує масштабування та адміністрування інфраструктури, особливо у великих та динамічних середовищах. Створення, змінення та видалення ресурсів можна без проблем виконувати за допомогою послідовних конфігурацій. Надає можливість зберігання файлів конфігурації в системах керування версіями, таких як GIT, Terraform дозволяє спільно працювати та ефективно відстежувати зміни, покращуючи загальне керування та обслуговування інфраструктури.

Тому, в першу чергу, завдання яке необхідно виконати, це написання пайплайнів, тобто скриптів, для автоматизованого розгортання контейнерів та ресурсів у середовищі Google Kubernetes Engine, оскільки Google Kubernetes Engine надає інфраструктуру для розгортання контейнерів Kubernetes у хмарі Google Cloud Platform.

Надалі при реалізації системи маємо завдання з розробки, розгортання та експлуатації програмного забезпечення. Для цих цілей підходить фреймворк Tekton, це відкритий фреймворк для неперервної інтеграції та постійної доставки (CI/CD), який надає можливість автоматизувати етапи розробки, розгортання і управління програмним забезпеченням. Тобто, за допомогою цього фреймворку ми зможемо розв'язати такі завдання:

- створення контейнерних зображень;

- запуск тестових сценаріїв для перевірки якості програми;
- розгортання програмного забезпечення у інфраструктурі;
- управління моніторингом та масштабуванням системи.

Tekton дозволяє автоматизувати ці завдання за допомогою простого та інтуїтивно зрозумілого синтаксису. Так, замість Tekton можна виконувати всі завдання CI/CD вручну. Однак це є трудомістким і громіздким процесом, який схильний до помилок.

Tekton дозволяє автоматизувати ці завдання, що робить їх більш ефективними, надійними та масштабованими. Крім того, Tekton забезпечує велику гнучкість та розширюваність, що робить його ідеальним інструментом для сучасних розробників. Завдяки можливості визначати власні кроки та ресурси, розробники можуть створювати індивідуальні конвеєри CI/CD, відповідні конкретним вимогам їхніх проектів. Використання Tekton спрощує процес налаштування та виконання автоматизованих робочих процесів, що підвищує ефективність розробки та забезпечує стабільність у процесі поставки програмного забезпечення.

Виділимо основні задачі:

- вибір технологій та методів;
- розробка архітектури системи;
- впровадження системи;
- тестування системи.

Принцип, якого нам необхідно дотримуватись, це доступність сервісів, платформи та інструментів.

Отже, ми визначили основні завдання, які ми розв'яжемо у ході нашої роботи. Оскільки ми шукаємо рішення для виконання нашої роботи загальнодоступне, ми зупинимось над системою Google Kubernetes Engine, на платформі Google Cloud Platforme, оскільки ця платформа є загальнодоступною, має вбудовані інструменти для роботи, підтримує високий ступінь автоматизації.

Надалі, ми перейдемо до проектування системи розгортання у хмарі.

## 2 ПРОЕКТУВАННЯ РОЗГОРТАННЯ У ХМАРІ З КОНТЕЙНЕРНИМИ ТЕХНОЛОГІЯМИ

### 2.1 Аналіз критеріїв інструменту для розгортання застосунків у хмарі

Далі, згідно з поставленими завданнями, розглянемо інструменти, та хмарні платформи, які застосуємо при розгортанні застосунків у хмарі.

Почнемо з аналізу хмарних платформ для виконання етапів розгортання застосунку. При проведенні аналізу ми орієнтуємось на основні елементи, які наведені на рисунку 2.1:

- підтримка контейнерів;
- підтримка керування контейнерами;
- функціональність;
- простота використання;
- вартість.

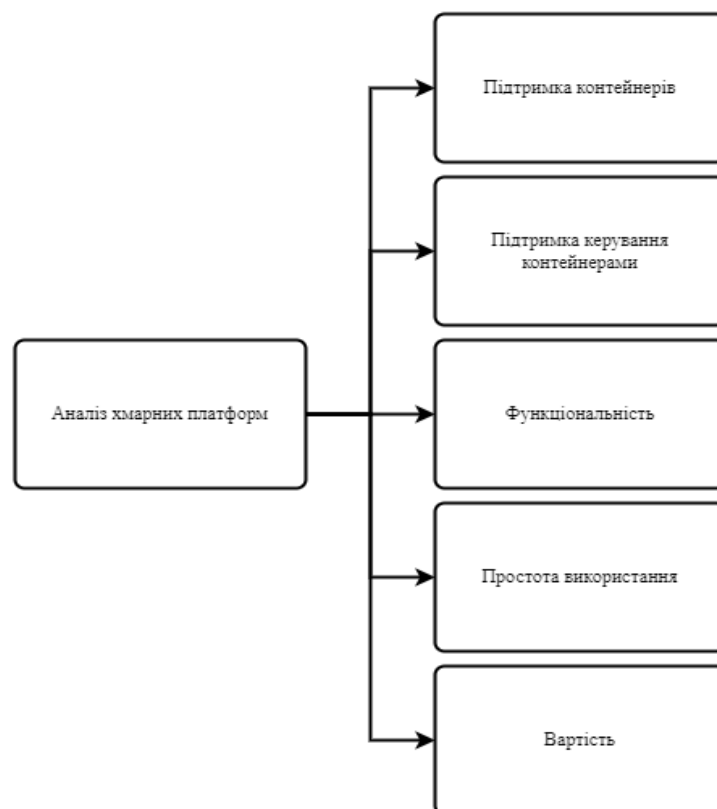


Рисунок 2.1 – Принципи аналізу хмарних платформ

Першим і найважливішим фактором при виборі інструменту розгортання контейнеризованих застосунків є підтримка контейнерів. Інструмент повинен підтримувати типи контейнерів, які ви збираєтеся використовувати для розгортання своїх застосунків. Найбільш популярним типом контейнерів є контейнери Docker.

Більшість інструментів для розгортання контейнеризованих застосунків підтримують контейнери Docker. Однак деякі інструменти також підтримують інші типи контейнерів, такі як Kubernetes або Open Container Initiative (OCI). Якщо ми не впевнені, який тип контейнерів використовувати, ми можемо почати з контейнерів Docker або Kubernetes. Вони є найбільш поширеними і підтримуються більшістю інструментів для розгортання контейнеризованих застосунків.

Другим важливим фактором при виборі інструменту для розгортання контейнеризованих застосунків є підтримка керування контейнерами. Але варто відповісти на питання:

- яку технологію керування контейнерами вибрати;
- який розмір і складність застосунку;
- яка необхідна кількість контейнерів, які необхідно розгорнути;
- чи можливе масштабування у майбутньому проекті.

Тому, інструмент повинен підтримувати технологію керування контейнерами, яку ви збираєтеся використовувати. Найбільш популярною технологією керування контейнерами є Kubernetes. Більшість інструментів для розгортання контейнеризованих застосунків підтримують Kubernetes. Однак деякі інструменти також підтримують інші технології керування контейнерами, такі як Docker Swarm або Mesos.

Якщо ми не впевнені, яку технологію керування контейнерами використовувати, ми можемо почати з Kubernetes. Вона є найбільш поширеною і підтримується більшістю інструментів для розгортання контейнеризованих застосунків.

Третім важливим фактором при виборі інструменту для розгортання

контейнеризованих застосунків є підтримка хмарних платформ. Інструмент повинен підтримувати хмарну платформу, на якій ми можемо розгортати свої застосунки.

Найбільш популярними хмарами є Amazon Web Services (AWS), Microsoft Azure та Google Cloud Platform (GCP). Більшість інструментів для розгортання контейнеризованих застосунків підтримують основні хмарні платформи. Якщо ми не впевнені, на якій хмарі розгортати свої застосунки, ми можемо почати з однієї з основних хмарних платформ, наприклад Google Cloud Platform (GCP), який підтримуються більшістю інструментів для розгортання контейнеризованих застосунків, має чудовий інтуїтивно зрозумілий інтерфейс та функціонал.

Четвертим важливим фактором при виборі інструменту для розгортання контейнеризованих застосунків є простота використання. Легкість використання є ключовим аспектом при виборі інструменту для розгортання контейнеризованих додатків.

Зручний інтерфейс користувача (UI) та ефективний командний рядок (CLI) визначають, наскільки швидко і зручно розробники та оперативні команди можуть пристосуватися до використання цього інструменту. Необхідно, щоб UI був інтуїтивно зрозумілим і легким у використанні, а також мав усі необхідні функції для успішного розгортання контейнеризованих застосунків. Щодо CLI, важливо, щоб він був належним чином задокументований і простий у використанні, надаючи всі необхідні можливості для ефективного управління контейнерами.

П'ятим важливим фактором при виборі інструменту для розгортання контейнеризованих застосунків є вартість. Інструмент повинен відповідати вашому бюджету. Ціна хмарних платформ для розгортання контейнеризованих застосунків може варіюватися в залежності від функцій, які вони пропонують. Деякі інструменти є безкоштовними, тоді як інші можуть бути платними.

Якщо ми маємо обмежений бюджет, ми можемо обрати хмару, яка у тріальній версії, або у версії до певного об'єму даних надає нам безкоштовний період, так звані кредити. Тому, враховуючи вартість самого інструменту та

послуг хмарної платформи, на якій ви розгортатимете свої контейнеризовані застосунки, ви можете вибрати інструмент, який відповідає вашому бюджету.

Також варто провести аналіз відповідності хмарних платформ за критерієм відповідності по параметрам IaaS, SaaS, PaaS у таблиці 2.1.

Таблиця 2.1 – Типи послуг хмарних обчислень

| Критерій   | IaaS<br>(Infrastructure as a Service)                                                                                                     | SaaS<br>(Software as a Service)                                                                          | PaaS<br>(Platform as a Service)                                                                                                                                                 |
|------------|-------------------------------------------------------------------------------------------------------------------------------------------|----------------------------------------------------------------------------------------------------------|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| Визначення | Провайдер хмарних послуг надає клієнтам доступ до базової інфраструктури, включаючи віртуальні машини, мережі, диски та зберігання даних. | Провайдер хмарних послуг надає клієнтам доступ до готового програмного забезпечення, яке працює в хмарі. | Провайдер хмарних послуг надає клієнтам доступ до платформи для розробки та розгортання застосунків, включаючи середовище виконання, інструменти розробки та засоби управління. |
| Переваги   | Гнучкість та масштабованість                                                                                                              | Зручність використання                                                                                   | Зручність розробки та розгортання застосунків                                                                                                                                   |
| Недоліки   | Контроль над власною інфраструктурою                                                                                                      | Зниження витрат на ІТ-інфраструктуру                                                                     | Зниження витрат на розробку та розгортання застосунків                                                                                                                          |

На основі вище визначених критеріїв ми можемо сформулювати порівняльну таблицю 2.2 хмарних платформ. Спираючись на дослідження Pallavi Wankhede «Comparative study of cloud platforms - microsoft azure, google cloud platform and amazon ec2» [23].

У порівняння ми ставимо три основні та найпоширеніші у світі хмарні платформи, Amazon Web Services (AWS); Microsoft Azure та Google Cloud

Platform (GCP). Оскільки вони надають весь необхідний повноцінний функціонал та інструментарій, але саме порівняння ми проведемо за власним дослідженням.

Таблиця 2.2 – Порівняння хмарних платформ

| Критерій                         | Amazon Web Services (AWS)                                                                                                | Microsoft Azure                                                                                                          | Google Cloud Platform (GCP)                                                                                              |
|----------------------------------|--------------------------------------------------------------------------------------------------------------------------|--------------------------------------------------------------------------------------------------------------------------|--------------------------------------------------------------------------------------------------------------------------|
| Підтримка контейнерів            | Docker, Kubernetes, Docker Swarm, Mesos                                                                                  | Docker, Kubernetes, Docker Swarm, Mesos                                                                                  | Docker, Kubernetes, Docker Swarm, Mesos                                                                                  |
| Підтримка керування контейнерами | Kubernetes, Docker Swarm, Mesos                                                                                          | Kubernetes, Docker Swarm, Mesos                                                                                          | Kubernetes, Docker Swarm, Mesos                                                                                          |
| Функціональність                 | створення і керування кластерами контейнерів та їх службами;<br>автоматизація розгортання контейнеризованих застосунків. | створення і керування кластерами контейнерів та їх службами;<br>автоматизація розгортання контейнеризованих застосунків. | створення і керування кластерами контейнерів та їх службами;<br>автоматизація розгортання контейнеризованих застосунків. |
| Простота використання            | Інтерфейс користувача для розгортання та керування контейнеризованими застосунками.                                      | Інтерфейс користувача для розгортання та керування контейнеризованими застосунками.                                      | Інтерфейс користувача для розгортання та керування контейнеризованими застосунками.                                      |
| Вартість                         | 12 місяців - \$100                                                                                                       | 30 днів - \$200                                                                                                          | 60 днів - \$300                                                                                                          |

При аналізі хмарних платформ, у порівняльній таблиці, ми бачимо що всі три основні та найпопулярніші хмарні платформи за функціональністю є майже

ідентичними, одна серед них Google Cloud Platform (GCP) має найбільш вигідніший безкоштовний кредитний період. Так, на перший погляд Amazon Web Services (AWS) у часовому вимірі надає нам найбільш тривалий безоплатний період, однак у грошовому еквіваленті на 12 місяців ми отримаємо лише 100\$, яких вочевидь не вистачить на такий тривалий період. У той час Google Cloud Platform (GCP) надає нам найбільш сбалансований кредитний період. Оскільки функціональність та складність використання хмарною платформою однакова, але Google Cloud Platform (GCP) надає нам найбільш вигідний тестовий період, оберемо Google Cloud Platform (GCP) за основну хмарну платформу у нашому дослідженні. Враховуючи, що GCP існує вже підготовлена інфраструктура до роботи із контейнерами Google Kubernetes Engine, для системи контейнеризації ми оберемо Kubernetes.

Google Kubernetes Engine (GKE) [24] - це хмарний сервіс, який дозволяє розгортати та експлуатувати контейнеризовані застосунки на Google Cloud Platform. GKE використовує Kubernetes для управління контейнеризованими застосунками.

Наступним не менш важливим етапом при аналізі хмарних платформ та інструментів, є дослідження інструментів, які допоможуть нам автоматизувати як створення та розгортання ресурсів у контейнерах, так і автоматизувати супроводження та оновлення наших ресурсів. Почнемо з факторів, ха якими ми зможемо провести аналіз таких інструментів.

При дослідженні інструментів ми враховуємо такі фактори [25]:

- автоматизація - це процес використання програмних засобів та інструментів для автоматизації рутинних завдань, які раніше вимагали б людського втручання. В контексті розгортання додатків в хмарному середовищі, автоматизація грає критичну роль. Вона дозволяє автоматизувати процеси розгортання, масштабування, моніторингу та управління додатками, забезпечуючи швидкість, надійність та консистентність операцій. Засоби автоматизації, такі як скрипти, конфігураційні інструменти та системи управління конфігурацією, дозволяють значно скоротити час розгортання,

знизити ймовірність помилок та підвищити продуктивність. Важливою перевагою автоматизації є можливість створити інфраструктуру "як код" (Infrastructure as Code, IaC), що дозволяє легко відтворювати та масштабувати середовища, забезпечуючи консистентність між різними стадіями розробки та виробництва;

- масштабованість - в контексті розгортання додатків в хмарному середовищі вказує на здатність системи збільшувати або зменшувати ресурси в залежності від поточного навантаження. Це важливо, оскільки додатки можуть стикатися зі змінними обсягами роботи, і масштабованість дозволяє забезпечити оптимальну продуктивність та доступність системи. Засоби масштабування, такі як оркестратори контейнерів, автоматично додають або знімають ресурси (контейнери, віртуальні машини) згідно з потребами, що допомагає забезпечити надійну роботу додатків навіть при зростанні навантаження. Ця можливість є важливою для забезпечення ефективного використання ресурсів та зменшення витрат в хмарному середовищі, де ресурси можуть бути розраховані на летю, відповідно до потреб користувачів;

- моніторинг і логування – вони важливі для постійного відстеження та аналізу виконання додатків у хмарному середовищі. Моніторинг забезпечує збір та візуалізацію даних про роботу додатків, ресурси та навантаження, що дозволяє оперативно виявляти проблеми та оптимізувати продуктивність. Логування забезпечує реєстрацію подій та помилок, що спрощує пошук і аналіз проблем для швидкого вирішення. Вони допомагають забезпечити безпеку, надійність та ефективність додатків, а також служать для відповідності стандартам і регуляторним вимогам;

- контейнеризація та підтримка мікросервісної архітектури – це технологія, що дозволяє ізолювати та запускати додатки та їх залежності в інкапсульованих контейнерах, забезпечуючи високу ізоляцію між ними та оптимізуючи використання ресурсів. Вона дозволяє легко переносити додатки між різними середовищами, зменшує розмір та час запуску. Цей принцип важливий для розгортання додатків в хмарному середовищі через здатність

розділяти додатки на невеликі сервіси, які можуть бути упаковані в окремі контейнери та розгортати, масштабувати та керувати ними незалежно. Це дозволяє розробникам розвивати, оновлювати та масштабувати окремі компоненти додатку без впливу на решту системи. Засоби автоматизації і оркестратори контейнерів надають зручність керування мікросервісами та забезпечують гнучкість та швидкість в розгортанні та масштабуванні компонентів додатку, сприяючи більшій стійкості та швидкій реакції на зміни вимог та навантаження;

- захист даних – у хмарному середовищі важливий через великий обсяг конфіденційної інформації, яка зберігається та обробляється у хмарі. Технічні засоби захисту, такі як шифрування даних у спокої та під час передачі, контроль доступу та ідентифікація користувачів, системи виявлення та захисту від вторгнень, грають ключову роль у забезпеченні конфіденційності та цілісності даних. Важливо також регулярно аудитувати та моніторити безпеку системи, виявляти та ліквідовувати потенційні загрози та вразливості. Захист даних є критичним аспектом, особливо в контексті дотримання регуляторних вимог, таких як GDPR [26], HIPAA [27], а також для збереження довіри користувачів та партнерів до хмарного середовища;

- цільова платформа хмари - це обрана нами хмарна інфраструктура, наприклад, такі як Amazon Web Services (AWS) [28], Microsoft Azure [29] або Google Cloud Platform [30], на якій будуть розгортатися додатки. Важливо враховувати вибір цільової платформи, оскільки від цього залежить доступність ресурсів, вартість, швидкість та можливості для розгортання та керування додатками. Технічні деталі, такі як можливість інтеграції з іншими службами платформи, вибір віртуальних машин або контейнерних середовищ, можливості мережевої конфігурації та підтримка відповідних інструментів для автоматизації та моніторингу, грають важливу роль у визначенні, наскільки ефективно та надійно будуть розгортатися та функціонувати додатки у хмарному середовищі.

Таким чином на рисунку 2.2 ми можемо візуалізувати принцип, на який ми можемо спиратись при виборі інструменту для розгортання у хмарному середовищі.

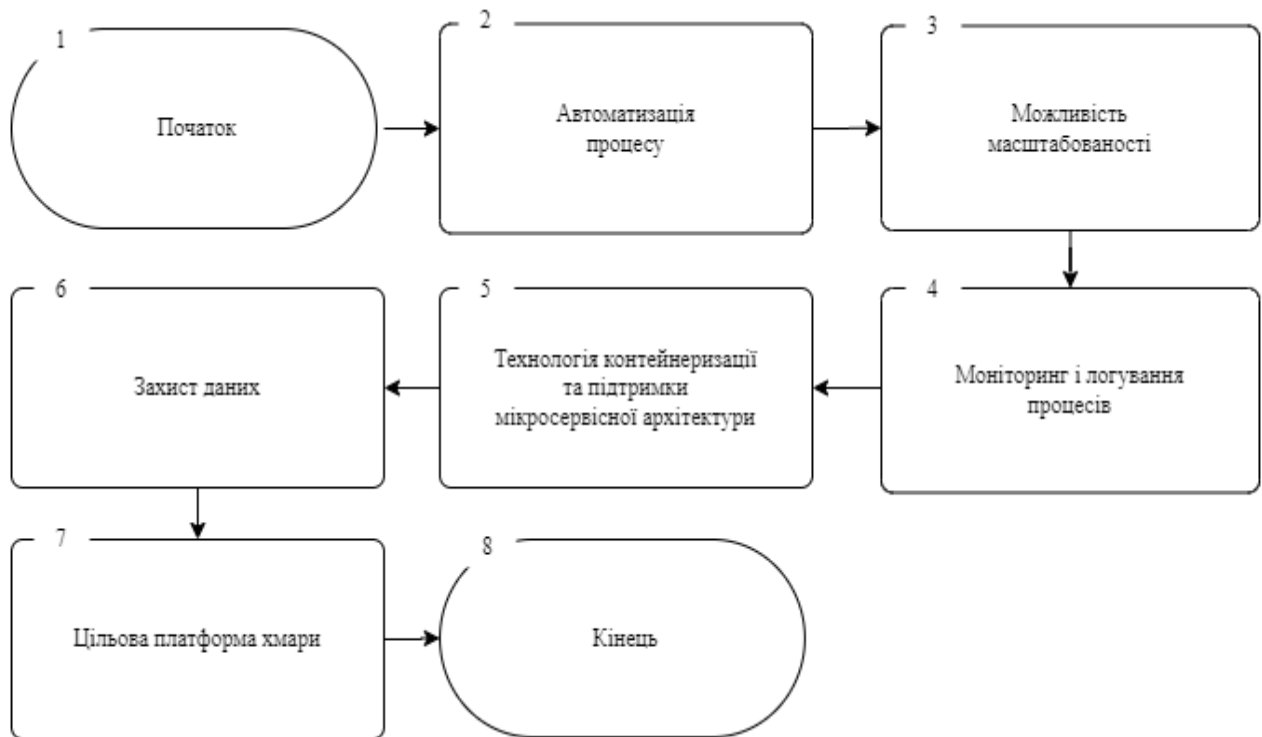


Рисунок 2.2 - Принцип вибору інструменту для розгортання у хмарі

Нам необхідні інструменти, які допоможуть керувати інфраструктурою та кодом для автоматизації і більш зручного керування ресурсами. Ці інструменти дозволяють автоматизувати процеси розгортання та зміни інфраструктури, що спрощує та прискорює впровадження змін, а також зменшує ризик помилок.

Шаблонізація конфігурацій допомагає стандартизувати та уніфікувати код, сприяючи повторному використанню та масштабуванню інфраструктури. Інфраструктура коду також забезпечує можливість визначення політик безпеки та стандартів в коді, що спрощує їх виконання та відслідковування за допомогою автоматизованих інструментів. Такий підхід є ключовим для забезпечення консистентності, автоматизації та ефективного управління ресурсами в сучасному інформаційному середовищі.

Почнемо з інструменту автоматизації інфраструктури, який дозволяє інженерам створювати, змінювати та знищувати інфраструктуру в середовищах. Для аналізу у таблиці 2.3, ми оберемо Terraform [31], Docker [31], Habitat [31] та Ansible [32].

Таблиця 2.3 – Порівняння інструментів для автоматизації керування інфраструктурою

| Критерій                                                           | Terraform                                                                        | Docker                                                                           | Habitat                                                                          | Ansible                                                                          |
|--------------------------------------------------------------------|----------------------------------------------------------------------------------|----------------------------------------------------------------------------------|----------------------------------------------------------------------------------|----------------------------------------------------------------------------------|
| Автоматизація процесу                                              | Автоматизація процесу за допомогою репозиторію Git                               | Автоматизація процесу за допомогою Dockerfile                                    | Автоматизація процесу за допомогою Habitat Runtime                               | Автоматизація процесу за допомогою YAML скриптів                                 |
| Можливість масштабованості                                         | Так, надає створення масштабованих кластерів контейнерів                         | Так, надає створення масштабованих кластерів контейнерів                         | Так, надає створення масштабованих кластерів контейнерів                         | Так, надає створення масштабованих кластерів контейнерів                         |
| Мониторинг і логування                                             | Підтримують моніторинг і логування контейнеризованих застосунків                 | Підтримують моніторинг і логування контейнеризованих застосунків                 | Підтримують моніторинг і логування контейнеризованих застосунків                 | Підтримують моніторинг і логування контейнеризованих застосунків                 |
| Технологія контейнеризації та підтримка мікросервісної архітектури | Підтримують технологію контейнеризації, яка є основою мікросервісної архітектури | Підтримують технологію контейнеризації, яка є основою мікросервісної архітектури | Підтримують технологію контейнеризації, яка є основою мікросервісної архітектури | Підтримують технологію контейнеризації, яка є основою мікросервісної архітектури |
| Захист даних                                                       | Функції шифрування даних та аутентифікація                                       | Функції шифрування даних та аутентифікація                                       | Функції шифрування даних та аутентифікація                                       | Функції шифрування даних та аутентифікація                                       |
| Цільова платформа хмари                                            | AWS, Azure, GCP                                                                  | AWS, Azure, GCP                                                                  | GCP                                                                              | AWS, Azure, GCP                                                                  |

Дамо пояснення, що це за інструменти, які ми порівнюємо.

Усі чотири інструменти забезпечують автоматизацію процесу розгортання контейнеризованих застосунків у хмарі, а також підтримують масштабованість, моніторинг і логування, технологію контейнеризації та підтримку мікросервісної архітектури. Щодо захисту даних, всі інструменти пропонують певні функції захисту, проте реалізація цих функцій може відрізнятися.

Terraform – це інструмент інфраструктури як коду, який дозволяє створювати, змінювати та видаляти інфраструктуру в хмарі за допомогою Git-репозиторію. Terraform підтримує різноманітні хмарні платформи, включаючи AWS, Azure та GCP. Для тих, хто прагне до повного контролю над своїми хмарами, Terraform є відмінним вибором.

Docker - це платформа контейнеризації, яка дозволяє створювати, розгортати та керувати контейнеризованими застосунками. Docker підтримує широкий спектр хмарних платформ, включаючи AWS, Azure та GCP. Для організацій, які мають намір використовувати контейнеризацію для розгортання застосунків у хмарі, Docker є відмінним рішенням.

Habitat - це платформа контейнеризації, спрямована на полегшення розробки та розгортання мікросервісних застосунків. Хоча Habitat підтримує лише GCP, він є ефективним вибором для тих, хто бажає використовувати мікросервісну архітектуру для розробки та розгортання застосунків у хмарі.

Ansible - це інструмент для автоматизації, який дозволяє автоматизувати завдання, включаючи розгортання контейнеризованих застосунків. Ansible підтримує різноманітні хмарні платформи, такі як AWS, Azure та GCP. Для тих, хто прагне використовувати Ansible для автоматизації розгортання контейнеризованих застосунків у хмарі, таке рішення є ефективним.

Отже, за результатами порівняння можемо зробити висновки, що всі три інструменти мають майже однаковий функціонал, а відрізняються лише у способах реалізації та їх інтерфейсах. Terraform, Docker, Habitat та Ansible є потужними інструментами, які можуть використовуватися для створення,

розгортання та управління контейнеризованими застосунками в мікросервісній архітектурі.

Terraform дозволяє автоматизувати процес розгортання контейнеризованих застосунків у хмарі шляхом використання Git-репозиторію. Використовуючи Terraform ми отримуємо можливість керувати створенням, зміною та видаленням різних ресурсів у хмарі, таких як віртуальні машини, мережі, диски та контейнери. Інструмент підтримує різноманітні хмарні платформи, такі як AWS, Azure та GCP.

Docker дозволяє автоматизувати процес розгортання контейнеризованих застосунків у хмарі шляхом використання Dockerfile. Dockerfile представляє собою текстовий файл з інструкціями для створення контейнера, який містить застосунок та його ресурси. Docker підтримує різноманітні хмарні платформи, такі як AWS, Azure та GCP.

Habitat дозволяє автоматизувати процес розгортання мікросервісних застосунків у хмарі. Використовуючи власну технологію контейнеризації, яку називають Habitat Runtime, Habitat підтримує лише GCP.

Ansible надає можливість автоматизувати різноманітні завдання, включаючи розгортання контейнеризованих застосунків. Використовуючи YAML-скрипти для автоматизації, Ansible підтримує різноманітні хмарні платформи, такі як AWS, Azure та GCP.

Щодо можливості масштабованості, то Terraform, Docker, Habitat і Ansible усі вони надають можливість масштабування контейнеризованих застосунків у хмарі. Ці інструменти дозволяють вам створювати розширювані кластери контейнерів.

Щодо моніторингу та логування, то моніторинг і логування контейнеризованих застосунків у хмарі можна здійснювати за допомогою Terraform, Docker, Habitat і Ansible. Ці інструменти надають можливість відслідковувати та реєструвати події у ваших контейнерах, щоб ви могли забезпечити ефективний моніторинг та отримувати необхідні дані для аналізу використання ресурсів.

Щодо технології контейнеризації та підтримка мікросервісної архітектури, то Terraform, Docker, Habitat та Ansible дозволяють використовувати технологію контейнеризації, яка є ключовою для реалізації мікросервісної архітектури. Ці інструменти дозволяють створювати контейнеризовані застосунки, які відповідають принципам та вимогам мікросервісів.

Щодо захисту даних, то Terraform, Docker, Habitat та Ansible усі вони мають певні можливості для захисту даних, такі як шифрування і аутентифікація. Спосіб реалізації цих можливостей може відрізнятися в залежності від кожного інструменту.

Спосіб реалізації тих чи інших інструментів ми можемо дізнатись з офіційних документів від команд розробників таких інструментів, та у офіційних документах провайдерів хмарних платформ для успішної інтеграції.

Terraform, Docker, Habitat та Ansible є потужними інструментами, які можуть використовуватися для створення, розгортання та управління контейнеризованими застосунками в мікросервісній архітектурі. Ці інструменти дозволяють забезпечити ефективне масштабування, управління конфігурацією та автоматизацію, що є важливими факторами для успішної реалізації мікросервісної архітектури.

Ми бачимо, що всі конкурентні інструменти мають майже однакові властивості, а мова написання у кожного інструмента індивідуальна. Але враховуючи, що саме Terraform є популярною та максимально адаптованою до роботи з різними інфраструктурами, у тому числі Google Cloud Platform.

Google Cloud Platform ми обрали для прикладу, як платформу з оптимальним кредитним безкоштовним періодом, ми зупинимо наш вибір на компонування, Google Cloud Platform, як основна хмарна платформа, а Terraform, як основний інструмент для автоматизації створення інфраструктури застосунку у хмарі. Отже, ми визначились, що для налаштувань автоматичного створення, зміни та знищення інфраструктури ми використаємо Terraform.

Наступним етапом для нас є необхідність визначитись з інструментом, який допоможе створювати так звані "pipelines" для автоматизації процесів

розробки, тестування та розгортання програмного забезпечення, так званий CI/CD, тобто інструменту для автоматизації обслуговування інфраструктури після її розгортання за допомогою Terraform. Але для початку визначимось, що є для нас CI/CD [33].

Безперервна інтеграція/безперервне розгортання (CI/CD) – це безперервна доставка та безперервне розгортання — це два процеси розробки програмного забезпечення, які зосереджені на швидкій доставці змін у програмному забезпеченні кінцевим користувачам.

Безперервна інтеграція (CI) є практикою розробки програмного забезпечення, коли члени команди регулярно інтегрують свою роботу та автоматизують створення, тестування та перевірку у єдиному репозиторії коду. Це допомагає швидко знаходити та виправляти помилки та покращувати якість програмного забезпечення.

Вона забезпечує покращену якість програмного забезпечення. CI допомагає забезпечити, щоб код був узгодженим і працював належним чином, перш ніж його розгорнути у виробництво. Це може допомогти уникнути випуску програмного забезпечення з помилками, що може призвести до занепокоєння клієнтів та негативно позначитися на репутації компанії.

Безперервне розгортання (CD) є процесом розробки програмного забезпечення, який зосереджений на швидкій доставці змін програмного забезпечення кінцевим користувачам, і в цьому процесі розробки програмного забезпечення поступові зміни програмного забезпечення автоматично тестуються та часто розгортаються у виробничих середовищах.

Тобто такий підхід дозволяє покращити якість програмного забезпечення: CD може допомогти забезпечити, щоб програмне забезпечення було стабільним і надійним, перш ніж його розгорнути у виробництво. Це пов'язано з тим, що CD передбачає, що всі зміни програмного забезпечення автоматично тестуються, перш ніж вони будуть розгорнуті.

На рисунку 2.3 наведено основний принцип роботи CI/CD.

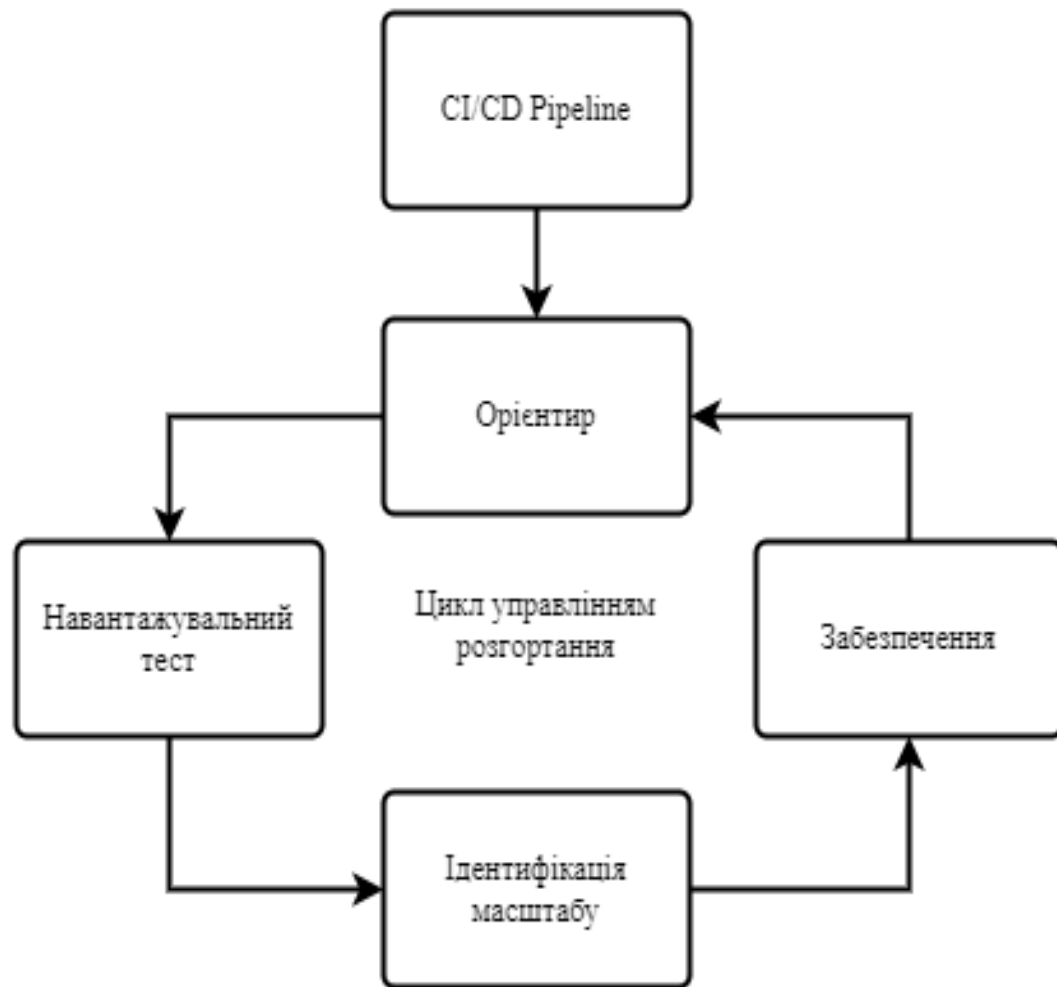


Рисунок 2.3 - Принцип роботи методу CI/CD

Отже, у таблиці 2.4 наведено результати аналізу та порівняння існуючих інструментів для створення CI/CD циклів. Існує велика кількість CI/CD інструментів, але ми в першу чергу орієнтуємось на інструменти, які є універсальними та кросплатформеними, щоб ми їх могли використовувати у разі необхідно на різних хмарних платформах, тому для нас існує необхідність аналізу наявних інструментів [34, 35]:

- Tekton CI/CD;
- Jenkins;
- Bitbucket Pipelines;
- GitHub Actions.

Таблиця 2.4 – Порівняння інструментів для автоматизації процесів розробки.

| Критерій                                  | Tekton CI/CD                                                       | Jenkins                                                           | Bitbucket Pipelines                                                     | GitHub Actions                                                     |
|-------------------------------------------|--------------------------------------------------------------------|-------------------------------------------------------------------|-------------------------------------------------------------------------|--------------------------------------------------------------------|
| Інтеграція з іншими інструментами         | Взаємодія з Kubernetes, Git та іншими інструментами DevOps.        | Широкі можливості інтеграції з різними інструментами CI/CD.       | Інтеграція з Bitbucket, Jira, Docker та іншими інструментами Atlassian. | Глибока інтеграція з GitHub, Docker і іншими інструментами GitHub. |
| Легкість налаштування та використання     | Доступний декларативний підхід до конфігурації пайплайнів.         | Має гнучкість, але вимагає деякого рівня конфігурації.            | Простий та легкий у використанні.                                       | Налаштовується з GitHub, що полегшує інтеграцію.                   |
| Масштабованість                           | Підтримка паралельних виконань, можливість розподілу навантаження. | Можливість налаштовувати робочі вузли для паралельного виконання. | Підтримка паралельних виконань, розподіл навантаження.                  | Розподілені агенти для більш швидкого виконання.                   |
| Підтримка мов програмування та технологій | Гнучка підтримка різних мов програмування та технологій.           | Підтримка великої кількості мов програмування та технологій.      | Підтримка різних мов програмування та технологій.                       | Різноманітна підтримка мов та технологій, включаючи Docker.        |
| Інструменти для тестування                | Інтеграція з різними інструментами тестування.                     | Розширена інтеграція з інструментами тестування.                  | Легка інтеграція з різними інструментами тестування.                    | Вбудована підтримка автоматизованого тестування                    |
| Безпека                                   | Заходи безпеки для кластерів Kubernetes та інших сервісів.         | Можливості автентифікації та авторизації.                         | Керування безпекою відомої компанії Atlassian                           | Інтеграція з інструментами, такими як Dependabot.                  |

Отже, ми маємо наступний опис порівняння CI/CD інструментів та їх інтеграцію з іншими інструментами:

- tekton CI/CD: використовує Kubernetes як основну платформу для виконання пайплайнів. Інтеграція з Git для взаємодії з репозиторіями коду та спрощення автоматизованого створення пайплайнів під час змін;
- jenkins: розширена підтримка різних систем контролю версій, включаючи Git, і іншими інструментами DevOps;
- bitbucket Pipelines: інтеграція з Bitbucket, Jira, Docker та іншими інструментами Atlassian;
- github Actions: глибока інтеграція з GitHub, що полегшує автоматизацію побудови та розгортання проектів.

Легкість налаштування та використання:

- tekton CI/CD: використовує декларативний підхід до конфігурації пайплайнів, описуючи їх у вигляді коду YAML;
- jenkins: має графічний інтерфейс та конфігураційні файли, але може вимагати деякої конфігурації для складних сценаріїв;
- bitbucket Pipelines: простий у використанні та конфігурації за допомогою файлу bitbucket-pipelines.yml;
- gitHub Actions: налаштовується безпосередньо з GitHub, спрощуючи інтеграцію та автоматизацію.

Масштабованість:

- tekton CI/CD: підтримка паралельних виконань та розподіл навантаження на кластерах Kubernetes;
- jenkins: можливість налаштовувати робочі вузли для паралельного виконання завдань та розподілу навантаження;
- bitbucket Pipelines: підтримка паралельних виконань для розподілу навантаження на різні вузли;
- gitHub Actions: має розподілені агенти для ефективного виконання завдань та паралельних виконань.

Підтримка мов програмування та технологій:

- tekton CI/CD: гнучка підтримка різних мов програмування та технологій завдяки концепції ресурсів та завдань (Tasks);
- jenkins: підтримує велику кількість мов програмування та технологій завдяки великому екосистемі плагінів;
- bitbucket Pipelines: підтримує різні мови програмування та технології для створення різноманітних пайплайнів;
- gitHub Actions: різноманітна підтримка мов та технологій, включаючи Docker для контейнеризації, та Python, Ruby, Go, Java.

#### Інструменти для тестування:

- tekton CI/CD: легка інтеграція з різними інструментами тестування завдяки ресурсам та завданням, які можна використовувати в пайплайнах;
- jenkins: широка інтеграція з інструментами для автоматизованого тестування через плагіни;
- bitbucket Pipelines: легка інтеграція з різними інструментами тестування для автоматизованого виконання тестів;
- github Actions: вбудована підтримка автоматизованого тестування та виконання тестів на різних платформах.

#### Безпека:

- tekton CI/CD: інтегрується з існуючими механізмами автентифікації та авторизації Kubernetes. Це означає, що ми можемо використовувати різні методи, такі як сертифікати, токени, або інші механізми для забезпечення доступу до пайплайнів, використовує Kubernetes під капотом, що дозволяє запускати кожний крок пайплайну у власному контейнері, використовувати механізми контролю доступу Kubernetes для управління правами доступу до пайплайнів та ресурсів;
- jenkins: має власну систему автентифікації та авторизації, де можна налаштовувати користувачів, ролі та права доступу, залежно від конфігурації, jenkins може використовувати агентів для запуску завдань на окремих вузлах, що надає деяку ізоляцію, надає гнучкі можливості для контролю доступу через налаштування ролей та прав доступу для користувачів та агентів;

- bitbucket Pipelines: інтегрується з системою автентифікації та авторизації Bitbucket, використовуючи токени та ключі доступу, кожний пайплайн виконується у власному контейнері, забезпечуючи високий рівень ізоляції, може використовувати налаштування прав доступу у Bitbucket для управління тим, хто має доступ до пайплайнів та конфігурації;

- github Actions: використовує токени для доступу до репозиторіїв та інших ресурсів. Токени можуть бути обмежені за обсягом та строком дії, Кожен крок у робочому процесі виконується у власному контейнері, забезпечуючи ізоляцію та безпеку виконання, інтегрується з системою керування доступом GitHub, що дозволяє налаштовувати доступ до робочих процесів.

Враховуючи вище перелічені фактори, нашим вибором є Tekton CI/CD. Оскільки цей інструмент є найбільш універсальним, уніфікованим, інтегрованим, та чудово адаптується до спільної роботи з Terraform, Kubernetes, повністю відповідає стандартам безпеки, і здатен повноцінно працювати з будь-яким хмарним провайдером, у тому числі з Google Cloud Platform, оскільки наш вибір серед хмарних провайдерів прийшовся на Google Cloud Platform.

Отже, провівши аналіз інструментів та хмарних провайдерів, ми можемо прийти до висновку, що Google Cloud Platform, Amazon Web Services, Microsoft Azure мають свою унікальну платформу, сервіс та всю необхідну інфраструктуру для роботи з розгортання програми у хмарі. Серед інструментів є великий вибір, але ми орієнтуємось на вартість, та адаптивність. Також при виборі даної конфігурації, важливо врахувати також високу рівень безпеки, що забезпечується інструментами Google Cloud Identity and Access Management (IAM). Тому, для подальшого дослідження, ми зупинимось на компонуванні системи з хмарного провайдеру Google Cloud Platform, системи автоматизації розгортання Terraform by HashiCorp, інструменту безперервної інтеграції та доставки Tekton CI/CD для автоматичного обслуговування розгорнутої системи.

## 2.2 Архітектура системи

Описуючи архітектуру системи розгортання застосунку у хмарі методом контейнеризації, зауважимо, що у вигляді системи для контейнеру, як сутності ми будемо використовувати, Kubernetes у середовищі Google Kubernetes Engine, це є спеціальне середовище у хмарній платформі, яку ми взяли за основу нашої роботи Google Cloud Platform для роботи з Kubernetes. Допоміжними інструментами у нас будуть Terraform та Tekton [34,35]. Надалі нам необхідно визначити що саме ми можемо розгорнути у контейнері хмари, це буде автоматизована система, яка буде допомагати розгорнути образ WordPress у кластері Kubernetes [36].

Розглянемо архітектуру нашого проекту. В GitHub Repository: наш проект починається з репозиторію на GitHub, де буде зберігатися весь вихідний код, конфігураційні файли та інші ресурси. Цей репозиторій стане центральним елементом для спільної роботи команди та автоматизації процесу розгортання.

Текстові файли конфігурації: використовуючи Terraform, ми можемо описати інфраструктуру Google Cloud Platform, таку як мережі, класи машин, а також інші ресурси. Ці файли конфігурації можуть бути збережені в окремій текі у репозиторії.

Automated Deployment Pipeline: Tekton буде використовуватися для створення автоматизованого конвеєра розгортання. Пайплайн Tekton буде включати кроки з побудови інфраструктури за допомогою Terraform, розгортання Kubernetes-кластера та завантаження контейнерів з додатком WordPress у Google Kubernetes Engine.

Kubernetes Cluster: доцільно використовувати Kubernetes у середовищі Google Kubernetes Engine (GKE). Kubernetes буде відповідальним за керування контейнерами, оркестрацію служб, масштабування та управління ресурсами, керування кластером буде відбуватись за допомогою kubectl.

WordPress Docker Image: образ WordPress разом з конфігураційними файлами та іншими ресурсами буде розміщений у репозиторії на GitHub, звідки він буде використовуватися під час розгортання.

Масштабування та Оновлення: Kubernetes дозволяє легко масштабувати додаток та проводити оновлення без перерв у роботі сервісу. Tekton допоможе в автоматизованому вигляді виконувати ці процеси.

При тестуванні ми можемо включити тести, які перевірятимуть конфігурацію Kubernetes, переконуючись, що наші ресурси правильно налаштовані та готові до роботи. Це допоможе уникнути проблем, пов'язаних з неправильною конфігурацією під час розгортання, а інтеграція тестування з GitHub дозволяє автоматично запускати тести при кожному внесенні змін до репозиторію, що забезпечує швидке виявлення та вирішення проблем.

На рисунку 2.4 наведена архітектура реалізованої системи.

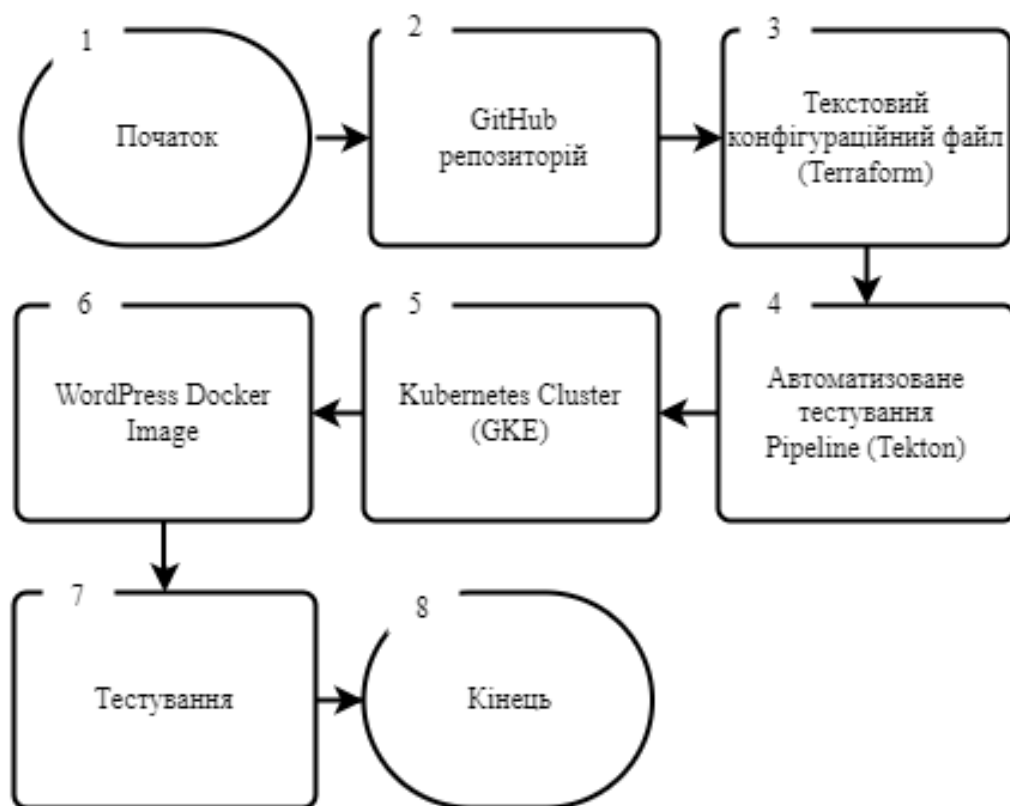


Рисунок 2.4 – Схема архітектура системи

Розглянемо схему та дамо визначення її основним елементам. GitHub репозиторій є нашим основним репозиторієм, де зберігається весь вихідний код, конфігураційні файли та інші ресурси проекту. Це центральний елемент для спільної роботи команди та автоматизації розгортання.

Текстовий конфігураційний файл Terraform, у ньому містяться файли конфігурації, які написані мовою Terraform, які визначають інфраструктуру на Google Cloud Platform (GCP), таку як мережі, класи машин тощо. Для цього визначається конфігураційний файл, що містить опис ресурсів, таких як кластер GKE, вузли та мережеві налаштування. Це дозволяє зручно задати параметри, такі як розмір кластера, регіон, типи та кількість вузлів, а також визначити правила мережі для забезпечення взаємодії між компонентами системи, та забезпечує автоматизоване створення та конфігурацію кластера для розгортання контейнерів. Завдяки цьому підходу до інфраструктурного програмування, ми можемо відтворювати та масштабувати інфраструктуру GKE для різних проектів, забезпечуючи стабільність та консистентність у середовищі розгортання. Цей етап є важливою основою для подальших кроків, таких як контейнеризація та налаштування системи для ефективного розгортання та управління WordPress на платформі GKE.

Автоматизоване розгортання Tekton CI/CD Pipeline [37], це автоматизований конвеєр розгортання. Він включає етапи від побудови інфраструктури за допомогою Terraform, для розгортання Kubernetes-кластера, завантаження контейнерів з WordPress у Google Kubernetes Engine (GKE). Визначаючи робочий процес Tekton, можемо створювати послідовність етапів, які відображають конкретні кроки необхідні для успішної доставки додатку в середовище виробництва. Це починається з клонування вихідного коду з репозиторію, що може бути виконаною через інтеграцію з різними системами контролю версій, такими як Git. Tekton може взаємодіяти з іншими інструментами CI/CD, ініціювати цей етап за наявності вже зібраних артефактів. До того ж, використання тригерів і обов'язків дозволяє автоматизувати запуск робочого процесу при виникненні певних подій, таких як зміни в репозиторії

коду. Це спрощує інтеграцію та розгортання нового функціоналу, а також забезпечує стабільність та надійність процесу доставки програмного забезпечення. Загалом, Tekton CI/CD стає важливим інструментом для автоматизації і управління процесами CI/CD в середовищі Kubernetes.

Kubernetes Кластер (GKE) на Google Kubernetes Engine, який відповідає за керування контейнерами, оркестрацію служб, масштабування та управління ресурсами. Взаємодія з кластером відбувається за допомогою `kubectl`, який встановлюється на локальному комп'ютері розробника, `kubectl` працює з конфігураційними даними, визначеними для підключення до Kubernetes-кластера, як-от адреса сервера API, автентифікаційні дані та контекст. Одним з основних причин використання `kubectl` є розгортання додатків у кластері. За допомогою команди `kubectl apply` і конфігураційних файлів, що описують структуру і параметри розгортання ресурсів, можемо ефективно керувати життєвим циклом своїх застосунків у Kubernetes.

У контексті нашої архітектури, `kubectl` використовується для управління додатком у Google Kubernetes Engine (GKE). Це охоплює не тільки розгортання і масштабування застосунку, а й взаємодію з базою даних Google Cloud SQL. Ми можемо використовувати `kubectl exec` для виконання команд усередині контейнера із застосунком, що забезпечує доступ і можливість взаємодії з базою даних зсередини застосунку.

WordPress Docker Image, є нашим Docker образом WordPress, разом з конфігураційними файлами та іншими ресурсами, які розміщено в GitHub, їх ми використовуємо під час розгортання самої програми. У цьому етапі створюється Docker-образ для WordPress з використанням Docker та створеного Dockerfile. Dockerfile містить інструкції для збірки образу, включаючи визначення базового образу та кроки для налаштування середовища та включення всіх необхідних компонентів для виконання WordPress.

Цей файл служить основою для створення контейнера, який містить усе необхідне для коректного функціонування веб-додатку. Після написання Dockerfile, запускаємо процес створення Docker-образу на основі вказаних

інструкцій. Це дозволяє створити ізольоване та переносиме середовище для WordPress, яке може бути легко перенесене та розгортане в інших середовищах. Останній крок цього етапу передбачає завантаження побудованого Docker-образу до контейнерного репозитарію, у нашому випадку, Google Container Registry. Це необхідно для того, щоб Kubernetes міг отримати доступ до образу та використовувати його під час розгортання контейнерів.

Масштабування та оновлення - масштабування в Kubernetes відбувається за допомогою поняття ресурсів, визначених у конфігураційному файлі. Зазвичай, це включає масштабування різних частин додатка, таких як репліки служб та робочі навантаження.

Наприклад, ми можемо вказати кількість реплік для додатка в межах ресурсів Deployment. При потребі система Kubernetes може автоматично розподіляти ці репліки між вузлами кластера для балансування навантаження. Оновлення додатка в Kubernetes може бути здійснене без перерв у його роботі завдяки стратегії деплойменту. При внесенні змін до конфігураційних файлів чи контейнерів, можемо використовувати поновлені версії ресурсів і дозволяти Kubernetes автоматично виконати оновлення.

І завершальний та надважливим етапом у архітектурі є тести, які перевіряють конфігурацію Kubernetes, гарантуючи правильність налаштувань перед розгортанням, а інтеграція тестування з GitHub дозволяє автоматично запускати тести при внесенні змін до репозиторію. Тести конфігурації Kubernetes можуть бути написані для перевірки, чи відповідають конфігураційні файли стандартам та вимогам. Наприклад, вони можуть включати перевірку наявності обов'язкових полів, правильності визначення служб та інших ресурсів. А інтеграція тестування з GitHub дозволяє автоматично запускати тести при кожному внесенні змін до репозиторію. Для цього використовуються можливості GitHub Actions, що дають змогу організовувати та виконувати автоматизовані робочі процеси.

Отже, підсумовуючи, ми маємо чіткі нариси архітектури системи, яка допоможе нам побудувати автоматизовану систему з розгортання образу

WordPress зі застосуванням технологій автоматизації, супроводження та тестування таким як, Terraform, CI/CD Tekton, Kubernetes, GitHub, kubectl, і саме головне, вся система може розгортатись як на базі хмарного провайдера Google Cloud Platform, так і його альтернативам Amazon Web Services та Microsoft Azure. Особливу цікавість такої архітектури представляє те, що може бути застосована на всіх трьох відмінних один від одного, але у той же час спільних хмарних платформах, за винятком певних технічних моментів, з якими ми можемо ознайомитись у офіційних документах провайдерів, але зберігаючи загальну концепцію архітектури.

## **3 ВПРОВАДЖЕННЯ СИСТЕМИ РОЗГОРТАННЯ ЗАСТОСУНКІВ У ХМАРІ**

### **3.1 Підготовка середовища та розгортання компонентів системи**

Цей розділ присвячено опису практичної частини роботи, а саме ми підготуємо середовище та опише процес безпосереднього розгортання системи. Якщо узагальнити то, цей розділ розглядає низку ключових етапів, які детально визначені у попередньому описі архітектури. Кожен етап цього процесу має стратегічне значення та відіграє важливу роль у створенні надійної, масштабованої та автоматизованої інфраструктури додатка.

Початковий крок - це створення та налаштування репозиторію на GitHub. Цей репозиторій буде центральним місцем зберігання вихідного коду, конфігурацій та Docker-образів додатка. Це дозволить команді спільно працювати над додатком та автоматизувати процеси розгортання.

Наступний етап - це використання Terraform для опису інфраструктури Google Cloud Platform. Terraform дозволяє створювати та керувати інфраструктурними компонентами, такими як мережі, класи машин та інші

ресурси. Це забезпечує автоматизований розгортання та управління інфраструктурою додатка.

Третій етап - це налаштування автоматизованого конвеєра розгортання за допомогою Tekton. Tekton - це платформа для автоматизації розгортання додатків. Конвеєр Tekton включає етапи побудови інфраструктури, розгортання Kubernetes-кластера та завантаження додатка. Це забезпечує швидкі та стабільні цикли розробки.

Четвертий етап - це створення та конфігурування Kubernetes-кластера в середовищі Google Kubernetes Engine (GKE). Kubernetes - це керуюча система для контейнерів. Kubernetes-кластер відповідає за управління контейнерами, забезпечуючи їхню оркестрацію, масштабування та ефективне розподілення ресурсів.

П'ятий етап - це підготовка Docker-образу для додатка. Docker-образ - це пакет, який містить весь необхідний код, конфігурації та ресурси для запуску додатка. Docker-образ можна зберігати в репозиторію на GitHub.

Шостий етап - це використання можливостей Kubernetes для легкого масштабування та оновлення додатка без перерв у роботі. Ми можемо динамічно змінювати кількість ресурсів та автоматично виконувати оновлення контейнерів.

Сьомий етап - це включення тестів, які перевіряють правильність конфігурації Kubernetes та готовність додатка до роботи. Тестування дозволяє швидко виявляти та вирішувати можливі проблеми.

Завершивши всі ці етапи, ми отримаємо готову до використання систему, яка автоматизує аспекти розробки, розгортання та моніторингу додатка на Kubernetes.

Отже, почнемо з першого етапу. Створення та налаштування репозиторію на GitHub є важливим етапом для ефективного управління вихідним кодом та ресурсами проекту. Створення нового репозиторію можна виконати через веб-інтерфейс GitHub. Нижче наведений лістинг 3.1, для створення репозиторію та клонування його локально:

### Лістинг 3.1 - Створення репозиторію та його клонування

```
curl -u 'наш_юзернейм' https://api.github.com/user/repos -d  
'{"name":"назва_репозиторію"}'  
  
git clone  
https://github.com/наш_юзернейм/назва_репозиторію.git  
  
cd назва_репозиторію
```

Далі нам необхідно створити репозиторій, тобто провести його ініціалізацію, як у лістингу 3.2.

### Лістинг 3.2 – Ініціалізація GIT репозиторію

```
echo "# Project Title" >> README.md  
git add README.md  
git commit -m "Додано README"  
  
git add .gitignore  
git commit -m "Додано .gitignore"  
git add LICENSE  
git commit -m "Додано ліцензію"
```

Пояснемо детально цей лістинг. Командою додаємо до файлу README.md заголовок Project Title. Файл README.md - це файл, який зазвичай використовується для опису проекту. Він містить інформацію про мету проекту, його авторів та способи використання. Командою `git add` додаємо файл README.md до репозиторію Git. Репозиторій Git - це місце, де зберігаються файли та історії змін цих файлів.

Команда `git commit -m` створює коміт, який представляє зміни, внесені до файлу README.md. Коміти використовуються для відстеження змін у репозиторію Git. Команда `git add` - додає файл .gitignore до репозиторію Git. Файл .gitignore - це файл, який містить список файлів та каталогів, які не повинні підлягати контролю змін Git. Команда `git add LICENSE` додає файл LICENSE до репозиторію Git. Файл LICENSE - це файл, який містить ліцензію, яка регулює використання проекту. Команда `git commit -m "Додано ліцензію"`, ця команда створює коміт, який представляє зміни, внесені до файлу LICENSE.

Загалом, цей лістинг створює три коміти, які додають файли README.md, .gitignore та LICENSE до репозиторію Git. Тепер нам необхідно внести вихідний код у віддалений репозиторій командами `git commit -m "Додано вихідний код та ресурси"` та `git push origin master`.

Наступним етапом є Terraform, для створення та налаштування інфраструктури в середовищі Google Cloud Platform (GCP). Terraform дозволяє описати інфраструктуру як код, що спрощує управління та підтримку. Але для початку позначимо, як виглядає архітектура проекту Terraform на рисунку 3.1. Слід зазначити, що файли Terraform зберігаються із розширенням файл «example.tf».

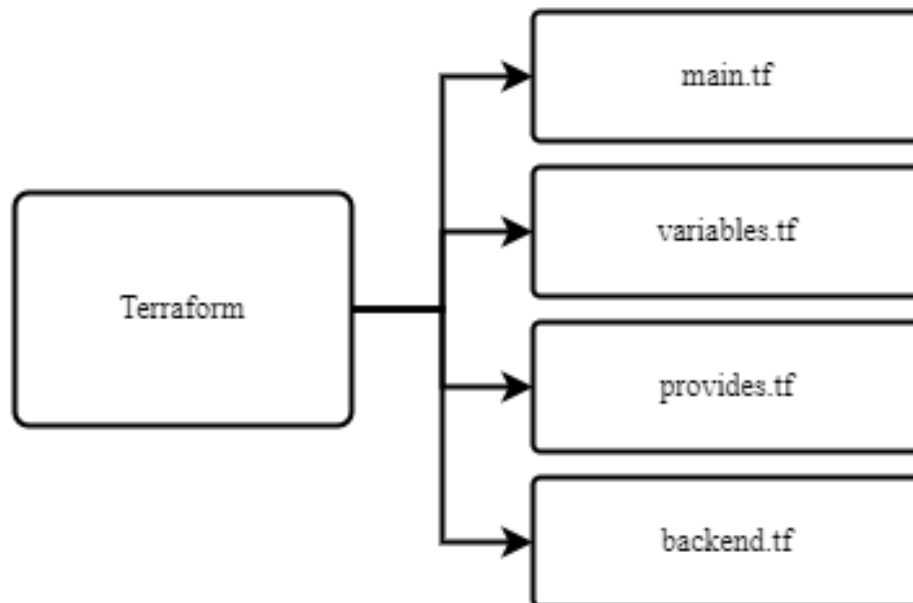


Рисунок 3.1 - архітектура проекту Terraform

Компанування архітектури саме у такій комбінації обумовлено саме тим, що таким чином ми визначаємо чітку структуру та організацію коду. У файлі `main.tf` розміщується основний код, який описує інфраструктуру проекту. Файл `variables.tf` визначає змінні, що полегшує параметризацію та передачу значень між частинами коду.

Файл `providers.tf` може використовуватися для визначення постачальників та їх конфігурацій для забезпечення взаємодії з конкретним середовищем,

наприклад, Google Cloud Platform. Файл `backend.tf` визначає конфігурацію бекенду, вказуючи, де точно зберігатиметься та керуватиметься станом інфраструктури. Ця розбита структура полегшує зрозуміння та підтримку проекту, дозволяючи концентруватися на конкретних аспектах конфігурації та розділяти відповідальності між різними частинами коду.

Перейдемо до складання головного файлу нашої інфраструктури. Перш за все, нам необхідно встановити Terraform на локальний комп'ютер. Тепер створюємо основний конфігураційний файл `main.tf`, де ми можемо описати необхідні ресурси GCP. Наприклад, визначити мережі, класи машин, балансувальники навантаження та інші ресурси. Зазначимо алгоритм конфігурації, алгоритм визначає кроки, необхідні для створення VPC-мережі, підмережі та кластера GKE на Google Cloud. Він також налаштовує HPA та масштабування кластера та оновлює записи DNS.

Першим кроком є створення VPC-мережі: визначаємо назву мережі за допомогою змінної `def_name` та зону проекту за допомогою змінної `zone_prj`. Встановлюємо `auto_create_subnetworks` на `false`, щоб запобігти автоматичному створенню підмереж.

Другим кроком є створення підмережі VPC: визначаємо назву підмережі за допомогою змінної `def_name`, зону проекту за допомогою змінної `zone_prj` та діапазон IP-адрес CIDR за допомогою змінної `local_ip_range`. Вказуємо регіон за допомогою змінної `region_prj` та мережу за допомогою змінної `google_compute_network.vpc_network.id`. Увімкнемо доступ Google до приватних IP-адрес за допомогою `private_ip_google_access = true`.

Третім кроком є створення кластера GKE: визначаємо назву кластера за допомогою змінної `def_name`. Вказуємо розташування кластера за допомогою змінної `region_prj`. Встановлюємо початкову кількість вузлів на 1 за допомогою `initial_node_count = 1`. Використовуємо VPC-мережу, створену в кроці 1, вказавши `network = google_compute_network.vpc_network.name`.

Далі використовуємо підмережу, створену в кроці 2, вказавши `subnetwork = google_compute_subnetwork.vpc_subnetwork.name`. Вмикаємо legacy АВАС за

допомогою `enable_legacy_abac = false`. Визначаємо діапазон IPv4 кластера за допомогою змінної `gke_ip_range`.

Важливий етап третього кроку, це встановлення конфігурації вузла, зазначено поетапно:

- встановлюємо `preemptible = true`, щоб дозволити випередження вузлів;
- використовуємо тип машини, наприклад, `g1-small` за допомогою `machine_type = "g1-small"`. Тому що цей тип є оптимальним для невеликих задач;
- вказуємо обліковий запис служби за допомогою `service_account = var.sa_account`;
- визначаємо OAuth-повноваження за допомогою `oauth_scopes = ["https://www.googleapis.com/auth/cloud-platform"]`;
- використовуємо тип диска `pd-standard` за допомогою `disk_type = "pd-standard"`;
- встановлюємо розмір диска на 30 ГБ за допомогою `disk_size_gb = 30`.

Наступним етапом третього кроку є налаштування масштабування горизонтальних контейнерів (HPA): увімкнемо HPA, встановивши `disabled = false` у `addons_config.horizontal_pod_autoscaling`.

Наступним блоком алгоритму є налаштування масштабування кластера:

Спочатку визначаємо значення за замовчуванням для автоматичного надання:

- встановлюємо розмір диска на 30 ГБ за допомогою `auto_provisioning_defaults.disk_size = 30`;
- встановлюємо тип диска `pd-standard` за допомогою `auto_provisioning_defaults.disk_type = "pd-standard"`;
- вказуємо OAuth-повноваження за допомогою `auto_provisioning_defaults.oauth_scopes = ["https://www.googleapis.com/auth/cloud-platform"]`;
- зазначаємо обліковий запис служби за допомогою `auto_provisioning_defaults.service_account = var.sa_account`;

Визначте залежність, щоб переконатися, що мережа та підмережа створені до кластера: встановлюємо залежності від ресурсів VPC-мережі та підмережі за

допомогою `depends_on = [google_compute_network.vpc_network, google_compute_subnetwork.vpc_subnetwork]`.

Четвертим кроком у даному алгоритмі є оновлення записів DNS: використовуйте провайдера DNS для оновлення імені ключа за допомогою змінної `domain`.

Підсумовуючи, наведемо опис алгоритму на рисунку 3.2.

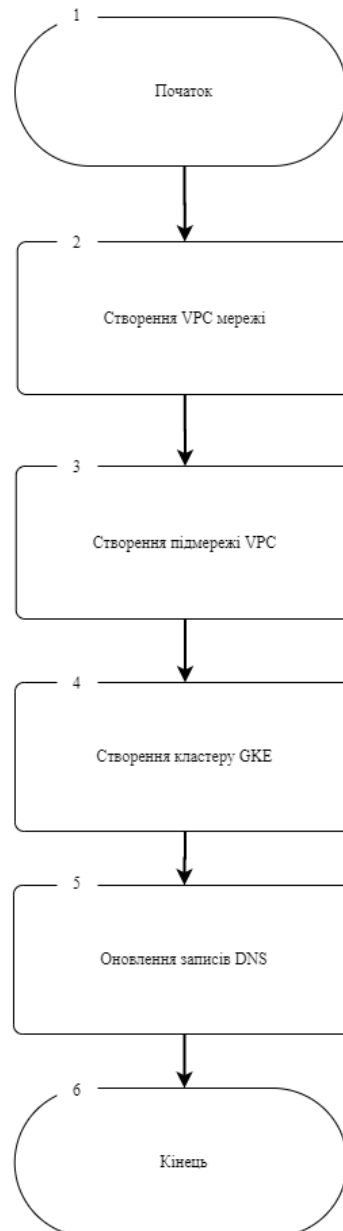


Рисунок 3.2 – алгоритм конфігураційного файлу `main.tf`

У файлі `variables` ми визначимо змінні та виводи. Змінні дозволяють параметризувати наш конфіг, а виводи надають дані, які можна використовувати на інших подальших етапах. Опишемо алгоритм роботи такого файлу.

У першому кроці ми визначимо набір змінних, які використовуються для конфігурації розгортання Terraform. Змінні - це зручний спосіб зберігання параметрів, які можна використовувати в конфігурації Terraform. Назвемо змінні наступним чином:

- `default` - значення за замовчуванням для назви ресурсів;
- `project_id` - ідентифікатор проекту;
- `region_prj` - регіон розгортання;
- `zone_prj` - зона розгортання;
- `sa_name` - назва служби;
- `sa_account` - адреса електронної пошти служби;
- `domain` - доменне ім'я;
- `local_ip_range` - діапазон локальних IP-адрес;
- `gke_ip_range` - діапазон IP-адрес GKE.

Кожна змінна має тип і значення за замовчуванням. Тип змінної визначає тип даних, які вона може зберігати. Значення за замовчуванням використовується, якщо не буде задано інше значення для змінної. Наприклад, змінна `default` має тип `string` і значення за замовчуванням `mondy`. Це означає, що якщо не буде задано інше значення для змінної `default`, то всі ресурси, створені в розгортанні Terraform, будуть мати ім'я `mondy`.

Другим кроком, ми встановимо значення за замовчуванням для змінних. Це можна зробити явно, вказуючи значення для кожної змінної, або неявно, використовуючи значення за замовчуванням. Всі змінні мають значення за замовчуванням.

Третім кроком є використання змінних у конфігурації Terraform. Конфігурація Terraform - це файл або набір файлів, які описують ресурси, які потрібно створити. Для використання змінної в конфігурації Terraform потрібно

використовувати оператор `var.` перед ім'ям змінної. Наприклад, для використання змінної `default` потрібно написати `var.default`.

Наведемо приклад того, як використовувати змінну `default` для встановлення значення за замовчуванням для назви ресурсу (лістинг 3.3).

### Лістинг 3.3 – змінні у конфігурації Terraform

```
resource "google_compute_instance" "example" {  
  name = var.default  
  machine_type = "n1-standard-1"  
  zone = var.zone_prj  
}
```

У цьому прикладі змінна `default` використовується для встановлення значення за замовчуванням для назви ресурсу `google_compute_instance.example`. Це означає, що інстанція Google Compute Engine буде створена з ім'ям `mondy`, якщо не буде задано інше значення. Змінні можна використовувати в конфігурації Terraform для встановлення значень для будь-якого атрибуту ресурсу. Наприклад, можна використовувати змінну для установки значення за замовчуванням для кількості ядер процесора або кількості пам'яті для інстанції Google Compute Engine.

Четвертим кроком є виконання розгортання Terraform. Після завершення конфігурації Terraform розгортання Terraform можна виконати, виконавши команду `terraform apply`. Ця команда створить ресурси, визначені в конфігурації Terraform. Коли команда `terraform apply` виконується, Terraform спочатку перевіряє, чи є всі необхідні ресурси вже створені. Якщо ресурси вже створені, Terraform не виконає жодних дій.

Якщо ресурси не створені, Terraform створить їх, використовуючи значення, задані в конфігурації Terraform. Terraform також створить будь-які залежності між ресурсами.

Наприклад, якщо конфігурація Terraform визначає інстанцію Google Compute Engine і мережу, Terraform спочатку створить мережу. Потім Terraform створить інстанцію Google Compute Engine, використовуючи мережу, створену

раніше. Після того, як всі ресурси будуть створені, Terraform виведе звіт про розгортання. Звіт містить інформацію про те, які ресурси були створені, а також будь-які помилки, які виникли під час розгортання. У рисунку 3.3 зазначимо алгоритм дії конфігураційного файлу variables.



Рисунок 3.3 – Алгоритм конфігураційного файлу variables.tf

Наступний етап архітектури, є файл `provides.tf`, він містить конфігурацію Terraform для визначення використаного постачальника та його налаштувань, його зміст наведено у лістингу 3.4.

#### Лістинг 3.4 – Конфігурація файлу `provides.tf`

```
terraform {
  required_providers {
    google = {
      source = "hashicorp/google"
      version = "4.57.0"
    }
  }
}
provider "google" {
  project = var.project_id
  region  = var.region_prj
  zone    = var.zone_prj
  credentials = "${file("credentials_file.json")}"
}
```

У блоці `terraform`, обов'язковий визначаємо постачальник `google` з версією `4.57.0`. Це означає, що Terraform використовуватиме дану версію постачальника для взаємодії з GCP. У блоку `provider "google"`, встановлюємо конфігурацію самого постачальника. Змінні `project_id`, `region_prj`, та `zone_prj` використовуємо для визначення проекту, регіону та зони в GCP. Також вказується шлях до файлу з обліковими даними (`credentials_file.json`), який використовується для автентифікації при взаємодії з GCP.

Більше детально стосовно визначення проекту, регіону та зони в Google Cloud Platform. У Google Cloud Platform (GCP), проект, регіон і зона є ключовими поняттями, які визначають, де і як розгортатимуться ресурси. Проект служить контейнером для групування та управління різними ресурсами. Задаючи унікальний ідентифікатор проекту за допомогою змінної `project_id` у Terraform, ви вказуєте, в якому проекті мають створюватися ваші ресурси. Регіон визначає географічну область для розташування обчислювальних ресурсів, впливаючи на доступність та вартість. Змінна `region_prj` у Terraform-коді визначає конкретний регіон для розгортання ресурсів. Зона, у свою чергу, вказує конкретний фізичний

дата-центр в межах обраного регіону. Змінна `zone_prj` в Terraform допомагає визначити, в якій саме зоні розташуватимуться ресурси, враховуючи доступність та відмовостійкість.

І фактично останнім етапом у архітектури Terraform, є конфігурація із визначенням місця, де буде зберігатися так керуватися інфраструктура. Ці конфігурації вкажемо у файлі `backend.tf` (лістинг 3.5).

### Лістинг 3.5 – Конфігурація файлу `backend.tf`

```
terraform {
  backend "gcs" {
    bucket = "bucketgcp" # Replace with your bucket name.
    prefix = "terraform/state"
    credentials = "credentials_file.json"
  }
}
```

У нашому випадку використовується бекенд GCS (Google Cloud Storage), що дозволяє зберігати стан Terraform у сховищі об'єктів у Google Cloud Platform. В блоку `terraform` визначається використання Google Cloud Storage як бекенду. Параметр `bucket` вказує ім'я сховища Google Cloud Storage, де буде знаходитися стан.

Опціонально можна вказати `prefix` для належного організаційного розміщення об'єктів у сховищі. Також вказується шлях до файлу з обліковими даними (`credentials_file.json`), який використовується для автентифікації при отриманні доступу до сховища GCS.

Таким чином ми задали основні конфігурації з налаштування системи розгортання.

На наступному етапі необхідно створити автоматизований конвеєр розгортання [38]. Пайплайн Tekton включає кроки побудови інфраструктури (за допомогою Terraform), розгортання Kubernetes-кластера та завантаження контейнерів з додатком WordPress у Google Kubernetes Engine (GKE).

Структура файлів має наступний вигляд (рис. 3.4).

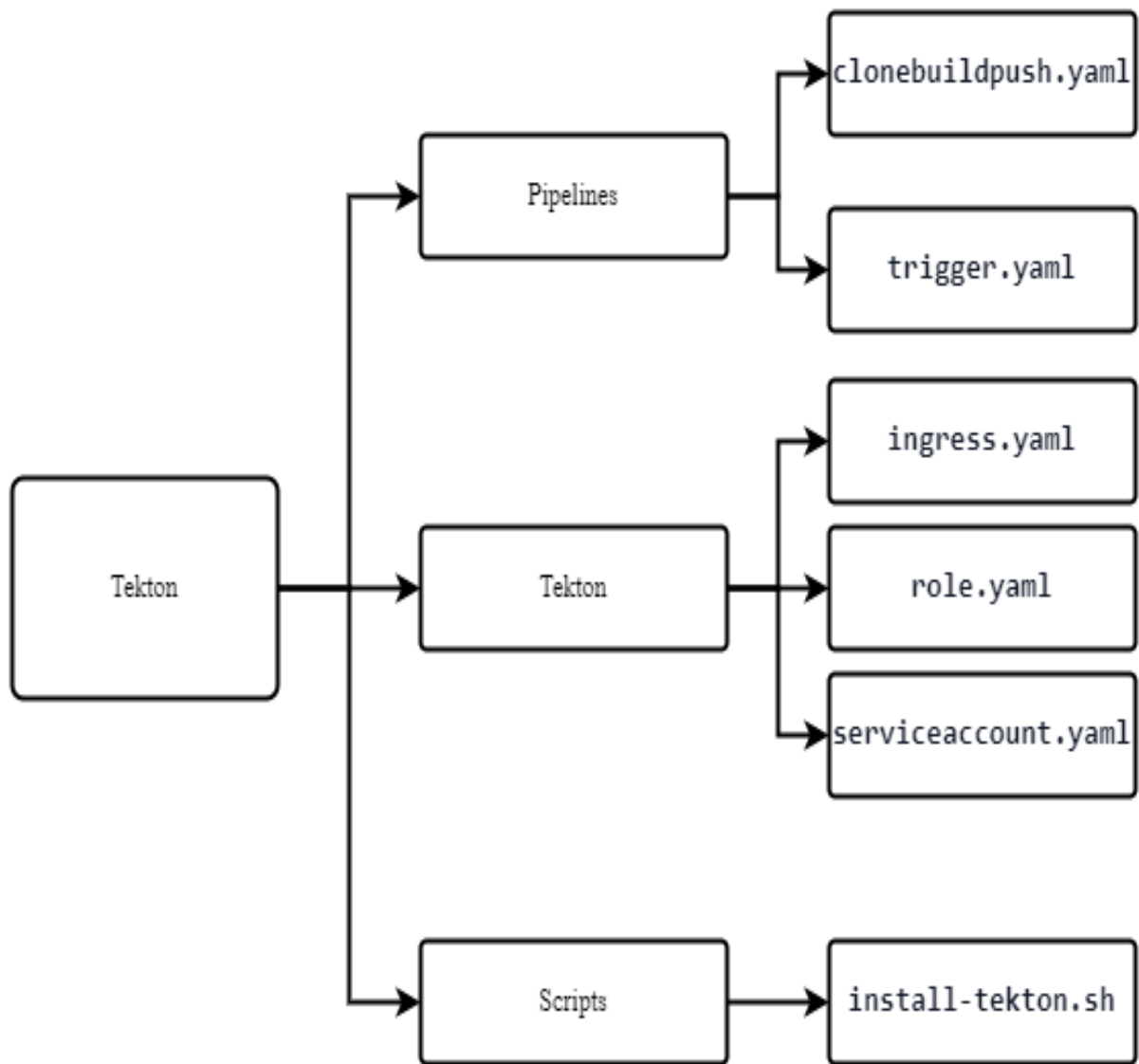


Рисунок 3.4 - структура файлів Tekton

У папці Pipeline містяться файли конфігурації для Tekton Pipelines. Файл `clonebuildpush.yaml` визначає конкретний пайплайн, який включає етапи клонування репозиторію, будівництва додатку та пушу змін. Файл `trigger.yaml` містить конфігурацію тригера, який може активувати цей пайплайн у відповідь на певні події. В папці Tekton розташовані файли конфігурації для різних компонентів Tekton. Файл `ingress.yaml` містить конфігурацію Ingress, яка дозволяє звертатися до ресурсів Tekton ззовні. `role.yaml` визначає RBAC роль для керування доступом. `serviceaccount.yaml` містить конфігурацію ServiceAccount, яка може бути використана для встановлення прав доступу Tekton до інших ресурсів у кластері. У папці Scripts розташований скрипт з назвою `install-`

tekton.sh, який призначений для автоматизованого встановлення Tekton. Такої структури ми притримуємось тому що, вона дозволяє чітко відокремити файли конфігурації Tekton Pipelines від інших складових, таких як ролі RBAC, конфігурація Ingress та інші. Скрипт для встановлення розташований у власній папці, що полегшує управління та використання його в контексті встановлення та налаштування Tekton.

Почнемо розгляд з директорії pipelines, директорія pipeline дозволяє автоматизувати процес розгортання контейнерів на кластер Kubernetes. Ця директорія має два файли clonebuildpush.yaml та trigger.yaml. Алгоритм роботи файлу clonebuildpush.yaml наведено на рисунку 3.5.

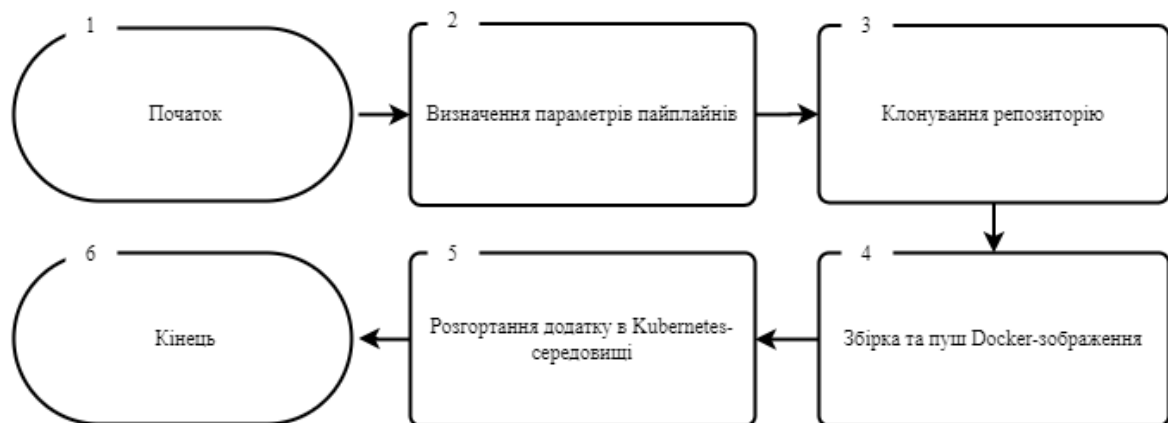


Рисунок 3.5 - структура файлу clonebuildpush.yaml

Наповнення файлу clonebuildpush.yaml наведено у лістингу 3.6.

### Лістинг 3.6 – Конфігурація файлу clonebuildpush.yaml

```

Create a Pipeline named "clonebuildpush":
Parameters:
  - revision
  - gitrepositoryurl
  - IMAGE
  - TAG
  - DOCKERFILE (with description and default value
"./terraform_gcp/Dockerfile")
Workspaces:
  - Create a workspace named "workspace"
Tasks:
  
```

```

- Task "fetch-repository":
  Use git-clone to clone the repository with the
  following parameters:
    - revision: $(revision)
    - url: $(gitrepositoryurl)
    - deleteExisting: "true"
  Workspaces:
    - Create an output workspace named "workspace"
- Task "build-push-image":
  Use buildah to build and push the image with the
  following parameters:
    - IMAGE: "$(IMAGE):$(TAG)"
    - FORMAT: "docker"
    - CONTEXT: "$(workspace.path)/terraform_gcp"
  After completing "fetch-repository"
- Task "kubectl-run":
  Use kubernetes-actions to run kubectl with the script:
    kubectl apply -f https://raw.githubusercontent.com/#
  Replace with your repo raw
  After completing "build-push-image"

```

Представляємо конфігурацію для створення і виконання CI/CD (Continuous Integration/Continuous Deployment) пайплайну за допомогою інструменту Tekton. Цей пайплайн призначений для автоматизації ряду операцій, пов'язаних з розгортанням програмного забезпечення. По-перше, він визначає параметри, такі як версія ревізії коду, URL репозиторію Git, імена та теги Docker-зображення, а також шлях до Dockerfile. Після цього, використовуючи Tekton, пайплайн клонує вказаний репозиторій, враховуючи вказану версію, та видаляє попередні дані. Далі, відбувається збірка та пуш Docker-зображення. З Dockerfile, використовується інструмент buildah. Після успішної збірки пайплайн формує та пушить Docker-зображення, використовуючи ім'я та тег, визначені в параметрах.

На завершення, пайплайн використовує інші інструменти, такі як kubectl (Kubernetes Command Line Tool), для автоматизації операцій у середовищі Kubernetes, kubectl необхідний для розгортання WordPress у Kubernetes. У конкретному випадку, застосовується конфігураційний файл, розташований за вказаним URL, для розгортання додатку в Kubernetes-середовищі. Отже, цей пайплайн дозволяє автоматизувати процеси розгортання та оновлення програмного забезпечення в середовищі Kubernetes, забезпечуючи швидку та надійну роботу CI/CD.

Файл `trigger.yaml`, алгоритм його роботи наведено на рисунку 3.6.

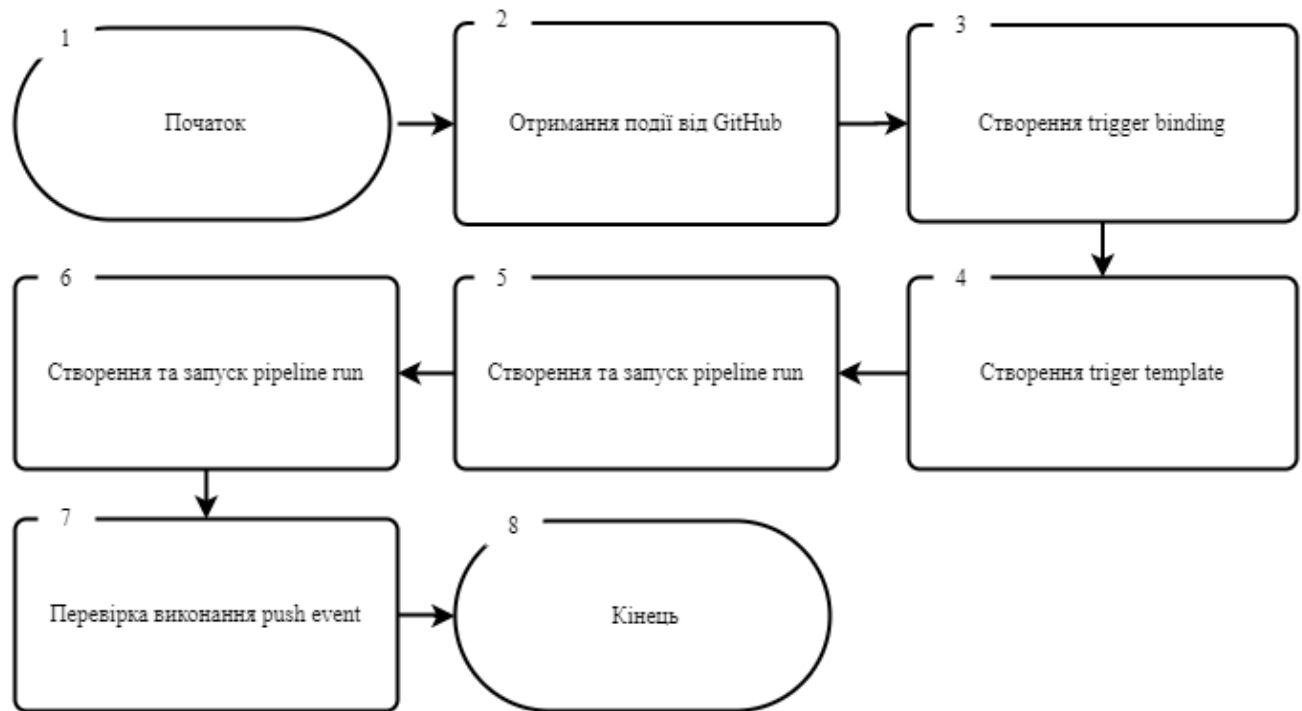


Рисунок 3.6 - структура файлу `trigger.yaml`

За цим алгоритмом, файл визначає тригер Tekton, який запускає конвеєр `clonebuildpush`, коли відбувається `push` в гілку `main` репозиторію Git, цей процес наведено у лістингу 3.7

### Лістинг 3.7 – Конфігурація файлу `trigger.yaml`

```

# Trigger Binding
apiVersion: tekton.dev/v1beta1
kind: TriggerBinding
metadata:
  name: git-push-binding
spec:
  params:
    - name: repositoryURL
      value: $(body.repository.html_url)
    - name: commits
      value: $(body.commits)

# Trigger Template
apiVersion: tekton.dev/v1beta1
kind: TriggerTemplate
metadata:
  name: git-push-template
  
```

```

spec:
  resourceTemplates:
    - pipelineRun:
        name: clonebuildpushrun-
        spec:
          serviceAccountName: pipeline
          pipelineRef:
            name: clonebuildpush
          podTemplate:
            securityContext:
              fsGroup: 65532
            params:
              - name: revision
                value: $(body.branch)
              - name: gitrepositoryurl
                value: $(params.repositoryURL)
              - name: IMAGE
                value: <span class="math-
inline">\(body\.repository\.name\) /</span>(body.branch)
              - name: TAG
                value: latest
            workspaces:
              - name: workspace
                volumeClaimTemplate:
                  spec:
                    accessModes:
                      - ReadWriteOnce
                    resources:
                      requests:
                        storage: 2Gi
# Event Listener
apiVersion: tekton.dev/v1beta1
kind: EventListener
metadata:
  name: git-push-listener
spec:
  triggers:
    - name

```

Отже, ми маємо те, що Trigger binding отримує інформацію про репозиторій та коміти з події push, наприклад, ім'я користувача, який виконав push.

Ця інформація може бути використана для додаткової логіки в pipeline за необхідності у вашому випадку. Ця інформація зберігається в змінних repositoryURL та commits. Trigger template використовує інформацію з trigger binding для запуску pipeline. Pipeline отримує наступні дані з trigger template:

- url-адресу репозиторію;

- гілку, на яку було виконано push.

Pipeline виконує наступні дії:

- створює робочий простір для зберігання файлів та артефактів;
- виконує clone репозиторій в робочий простір;
- створює образ Docker з комітами;
- виконує push образ Docker в репозиторій.

А створення робочого простору відбувається за допомогою Pipeline, він створює робочий простір для зберігання файлів та артефактів, які будуть створені або використані під час виконання pipeline. Робочий простір створюється за допомогою команди volumeClaimTemplate.

Клонування репозиторію Pipeline відбувається виконанням клону репозиторія в робочий простір за допомогою команди clone. Pipeline створює образ Docker з комітами за допомогою команди docker build. Виконання push образу Docker в репозиторій за допомогою команди docker push.

Надалі перевіряються гілки, Pipeline перевіряє, чи подія push була виконана на зазначеній гілці.

Якщо подія push була виконана на іншій гілці, pipeline не виконується. Pipeline отримує параметри image та tag з trigger template. Ці параметри використовуються для створення образу Docker.

Наступна директорія, яку ми розглянемо, це директорія Tekton, а саме структуру з файлу ingress.yaml. Структура файлу виглядає наступним чином, і наведена на рисунку 3.7:

- створити Ingress ресурс з назвою "ingress-resource";
- встановити клас Ingress на "nginx";
- вимкнути перенаправлення SSL;
- визначити правило для Ingress ресурсу;
- відповідати на запити до шляху "/hooks";
- використовувати точне співпадіння шляху;
- маршрутизувати запити до служби "el-git-push-listener";
- використовувати порт 8080.



Рисунок 3.7 - структура файлу ingress.yaml

Код файлу за цією структурою створює Ingress ресурс, який маршрутизує запити до шляху `/hooks` до служби `el-git-push-listener` на порту 8080. Це означає, що будь-який запит до Ingress ресурсу з шляхом `/hooks` буде відправлений до служби `el-git-push-listener`, де він буде оброблений. Більш конкретно, код виконує такі кроки: Створює Ingress ресурс з назвою `ingress-resource`. Встановлює клас Ingress на `nginx`. Вимикає перенаправлення SSL. Визначає правило для Ingress ресурсу, яке відповідає на запити до шляху `/hooks` і маршрутизує їх до служби `el-git-push-listener` на порту 8080. Цей код бути використаний для маршрутизації запитів Git-репозиторія до служби, яка обробляє ці запити. Наприклад, служба може бути використана для відстеження змін у репозиторії або для запуску завдань CI/CD.

Розглянемо файл `role.yaml`, наведений у лістингу 3.8. Код цього файлу встановлює дозволи для контролера Tekton Pipelines в середовищі Kubernetes. Ця система управління доступом дозволяє Kubernetes-адміністраторам контролювати, які користувачі або служби можуть виконувати які дії над певними ресурсами. Це допомагає захистити ресурси від несанкціонованого доступу.

#### Лістинг 3.8 – Конфігурація файлу `role.yaml`

```

# Роль
role:
  type: Role
  
```

```

name: secret-viewer
rules:
  - apiGroups: [""]
    resources: ["secrets"]
    verbs: ["get", "watch", "list"]

# Прив'язка ролі
role_binding:
  type: RoleBinding
  name: secret-viewer-binding
  rol_ref:
    type: Role
    name: secret-viewer

```

Роль визначає набір дозволів для користувача або служби. Вона має назву, тип (Role або ClusterRole) і список правил. Правило визначає, які дії користувач або служба може виконувати над певними ресурсами. Прив'язка ролі пов'язує роль з користувачем або службою.

Вона має назву, тип (RoleBinding або ClusterRoleBinding) і посилання на роль. У прикладі псевдокоду роль secret-viewer надає користувачу або службі право отримувати, спостерігати та перераховувати секрети. Прив'язка ролі secret-viewer-binding прив'язує роль secret-viewer до служби tekton-pipelines-controller. Тобто, ми можемо використовувати таку конфігурацію для надання доступу до певних ресурсів лише певним користувачам або службам.

Наприклад, можна надати доступ до секретів лише користувачам, які мають необхідні права безпеки, або використовувати для обмеження доступу до ресурсів. Наприклад, можна надати доступ до секретів лише для читання, щоб запобігти їх зміні. Також, можна відстежувати доступи до ресурсів. На рисунку 3.8 гаведено алгоритм роботи файлу.

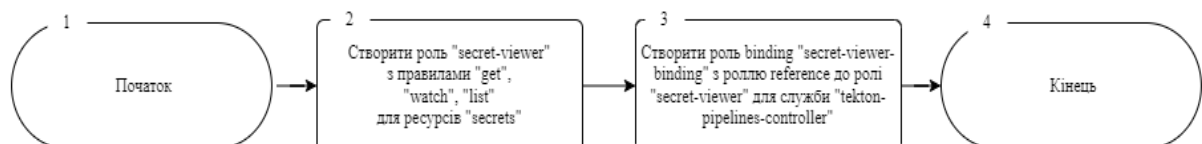


Рисунок 3.8 - структура файлу role.yaml

Узагальнюючи, цей код надає контролеру Tekton Pipelines доступ до всіх ресурсів Kubernetes у кластері, включаючи секрети. Контролер також може отримувати, переглядати та перераховувати секрети у всіх просторах імен.

У файлі `serviceaccount.yaml` просто визначаємо обліковий запис служби Kubernetes, цей обліковий запис служби має доступ до секрету, його алгоритм ми позначемо у рисунку 3.9. Визначаємо версію API, тип об'єкта, який описує код, метадані про об'єкт, та список секретів, доступ до яких має обліковий запис служби.

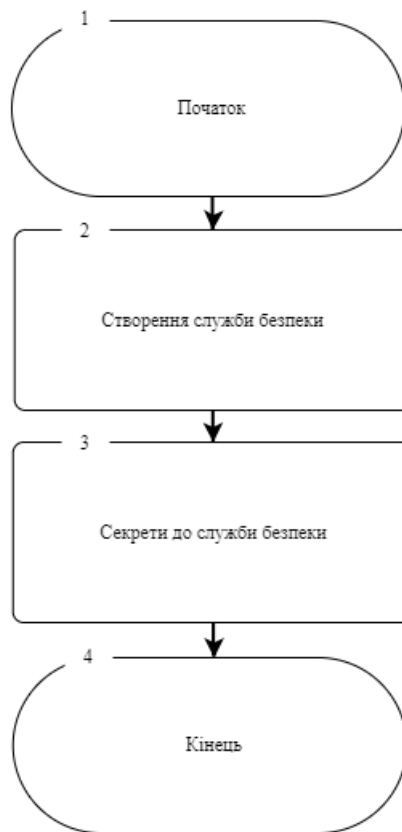


Рисунок 3.9 - структура файлу `serviceaccount.yaml`

У директорії `scripts` розміщується файл `install-tekton.sh`, який описує як він встановлює та розгортає компоненти Tekton Pipelines на кластері Kubernetes. А саме, завантажує пакет Tekton CLI, застосовує YAML-маніфест випуску Tekton Pipelines, встановлює завдання `git-clone`, `buildah` та `kubernetes-actions`, застосовує YAML-маніфест випуску Tekton Triggers та Tekton Operators, створює секрет

regsecret з вмістом, що містить назву та пароль для репозиторію Docker, застосовує ресурси з каталогу pipeline та tekton. Тобто цей файл використовується для створення конвеєрів Tekton, які автоматизують завдання, такі як клонування репозиторіїв Git, будівництво образів Docker та розгортання контейнерів, та описується у довільному порядку, спираючись на документації.

Таким чином, ми проілюстрували архітектуру та алгоритми роботи інструменту безперервної розробки та доставки CI/CD Tekton у контексті розгортання та супроводження застосунку у середовищі GKE.

Нашим наступним кроком є цій системі є створення та конфігурація Kubernetes-кластера у GKE, який дозволяє нам використовувати потужності оркестрації контейнерів. А керування кластером за допомогою kubectl надає зручний спосіб управління контейнерами, масштабування та ресурсами. Або кластер ми також можемо створити користуючись інтерфейсом у Google Cloud.

Далі необхідно провести розгортання компонентів системи, для цього створюється файл формату .yaml для, умовно називатись він буде deployment.yaml, алгоритм його роботи гаведено на рисунку 3.10.

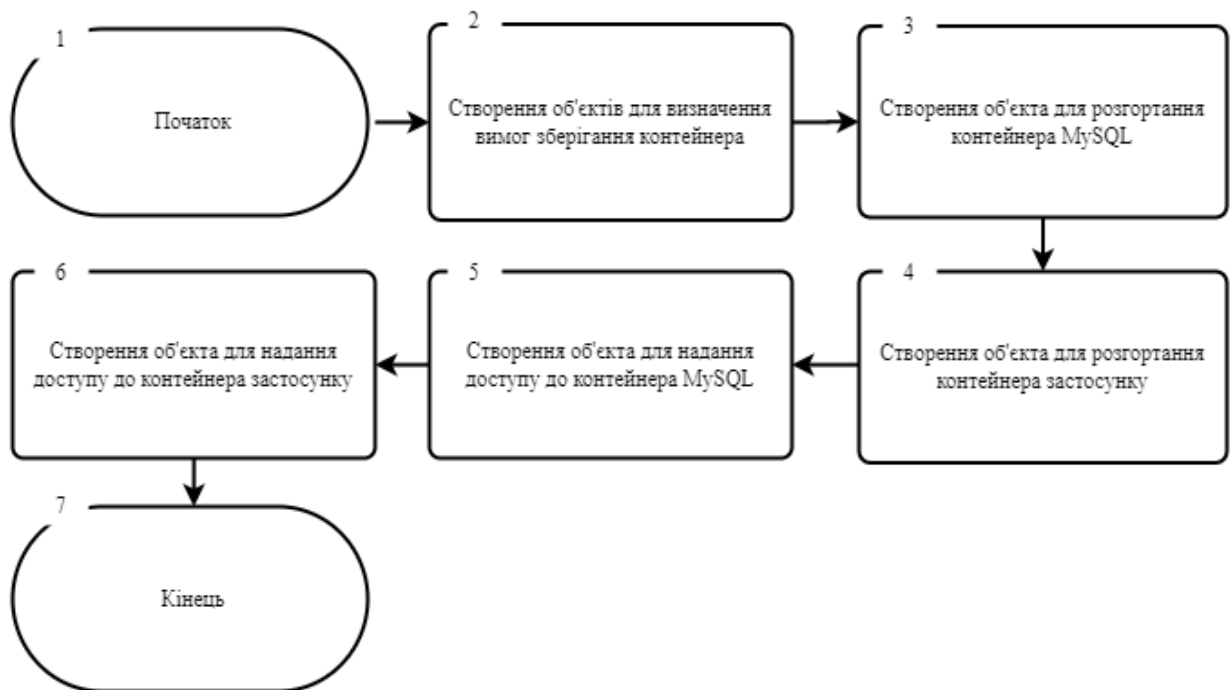


Рисунок 3.10 - структура файлу deployment.yaml

Проаналізуємо його код у лістингу 3.9.

### Лістинг 3.9 – Конфігурація файлу deployment.yml

```
db-pvc:
  capacity: 5Gi
  accessMode: ReadWriteOnce
  storageClass: standard
wp-storage:
  capacity: 10Gi
  accessMode: ReadWriteOnce
  storageClass: standard
# Database Service
db:
  replicas: 1
  podTemplate:
    container:
      name: database
      image: mysql:8.0
      command: mysqld --default-authentication-
plugin=mysql_native_password
      environment:
        - name: MYSQL_ROOT_PASSWORD
          value: secure-password
        - name: MYSQL_DATABASE
          value: app-database
        - name: MYSQL_USER
          value: app-user
        - name: MYSQL_PASSWORD
          value: secure-password
      ports:
        - containerPort: 3306
# Application Service
wp:
  replicas: 1
  podTemplate:
    container:
      name: application
      image: wordpress:latest
      environment:
        - name: WP_DB_HOST
          value: db
        - name: WP_DB_NAME
          value: app-database
        - name: WP_DB_USER
          value: app-user
        - name: WP_PASSWORD
          value: secure-password
      ports:
        - containerPort: 80
```

Таким чином ми описуємо конфігурацію Kubernetes для простого веб-додатку WordPress. Він запускає два контейнери: один для MySQL і один для WordPress. Контейнер MySQL зберігає дані додатку WordPress. Контейнер WordPress запускає веб-сайт додатку WordPress. Ми використовуємо два об'єкти PersistentVolumeClaims (PVC) для визначення вимог до зберігання для кожного контейнера. Два об'єкти Service для надання доступу до контейнерів. Service для MySQL має тип ClusterIP, що означає, що він доступний лише зсередини кластера Kubernetes. Service для WordPress має тип LoadBalancer, що означає, що він доступний з будь-якої точки Інтернету. Таким чином, ми описуємо конфігурацію, яка дозволяє розгорнути простий веб-додаток WordPress у Kubernetes.

Отже, ми можемо розгорнути веб-додаток WordPress у Kubernetes, забезпечуючи зберігання даних у базі даних MySQL і доступність веб-сайту WordPress з будь-якої точки Інтернету.

## **4 ТЕСТУВАННЯ СИСТЕМИ РОЗГОРТАННЯ ЗАСТОСУНКІВ У ХМАРІ**

### **4.1 Стратегії тестування системи**

Загальна концепція тестування полягає в тому, щоб перевірити, чи відповідає система її вимогам. Тестування включає в себе виконання набору тестових випадків, які визначають, що саме має бути перевірено. Тестові випадки розробляються на основі вимог до системи. Результати тестування аналізуються, щоб виявити помилки. Помилки виправляються, а потім система тестується повторно, щоб переконатися в тому, що помилки були виправлені. Тестування є важливим етапом процесу розробки та розгортання системи. Воно допомагає з'ясувати те, що система відповідає її вимогам і працює належним чином.

Оскільки ми проводимо розгортання на базі Google Kubernetes Engine, важливим етапом у процесі розробки та впровадження застосунків, нам

необхідно провести тестування системи розгортання застосунку у хмарі. Тестування дозволяє виявити та усунути помилки в системі розгортання, перш ніж вони призведуть до збоїв у роботі.

Систему розгортання хмарних застосунків на GKE можна тестувати двома способами: автоматизованим тестуванням і ручним тестуванням [39].

Перший підхід полягає в тому, щоб використовувати інструменти автоматизованого тестування для виконання сценаріїв розгортання. Цей підхід дозволяє виконувати тести швидко та ефективно, але може не виявити всіх помилок. Другий підхід полягає в тому, щоб використовувати ручне тестування для перевірки системи розгортання, та перекрити недоліки автоматизованого тестування.

Для забезпечення ефективного тестування рекомендується використовувати обидва підходи. Але, ручне тестування проводиться використовуючи інтерфейс Google Kubernetes Engine і не вимагає додаткових спеціалізованих знань. Тож ми акцентуємось на автоматичному тестуванні.

Основними завданнями поведення тестування є:

- перевірити, чи система розгортання може успішно розгорнути та керувати за стосунками;
- перевірити, чи система розгортання може успішно відновитися після збоїв;
- перевірити, чи система розгортання може успішно працювати під навантаженням.

Автоматизовані тести покривають:

- тестування розгортання нового за стосунку;
- тестування оновлення існуючого за стосунку;
- тестування відновлення застосунку після збоїв;
- тестування роботи системи розгортання під навантаженням.

Таким чином було визначено основні цілі тестування та об'єкти покриття тестування, це дозволяє дослідити стратегії тестування у середовищі Google Kubernetes Engine.

Стратегії тестування системи є планом дій, які визначають, як буде проводитися тестування на етапах розгортання, щоб забезпечити її відповідність вимогам і мінімізувати ризики. У середовище Google Kubernetes Engine ми маємо три стратегії щодо тестування [39].

Першою моделлю тестування є так званий тестовий шаблон канарки. Тестовий шаблон канарки є методом тестування програмного забезпечення, який використовується для виявлення проблем з продуктивністю або масштабованістю до того, як вони стануть мати серйозні проблеми. Цей метод отримав свою назву від канарок, яких використовували шахтарі для виявлення отруйних газів. Під час тестування Canary ми частково розгортаємо змінну, а потім оцінюємо її продуктивність порівняно з базовим розгортанням, як показано на рисунку 4.1.

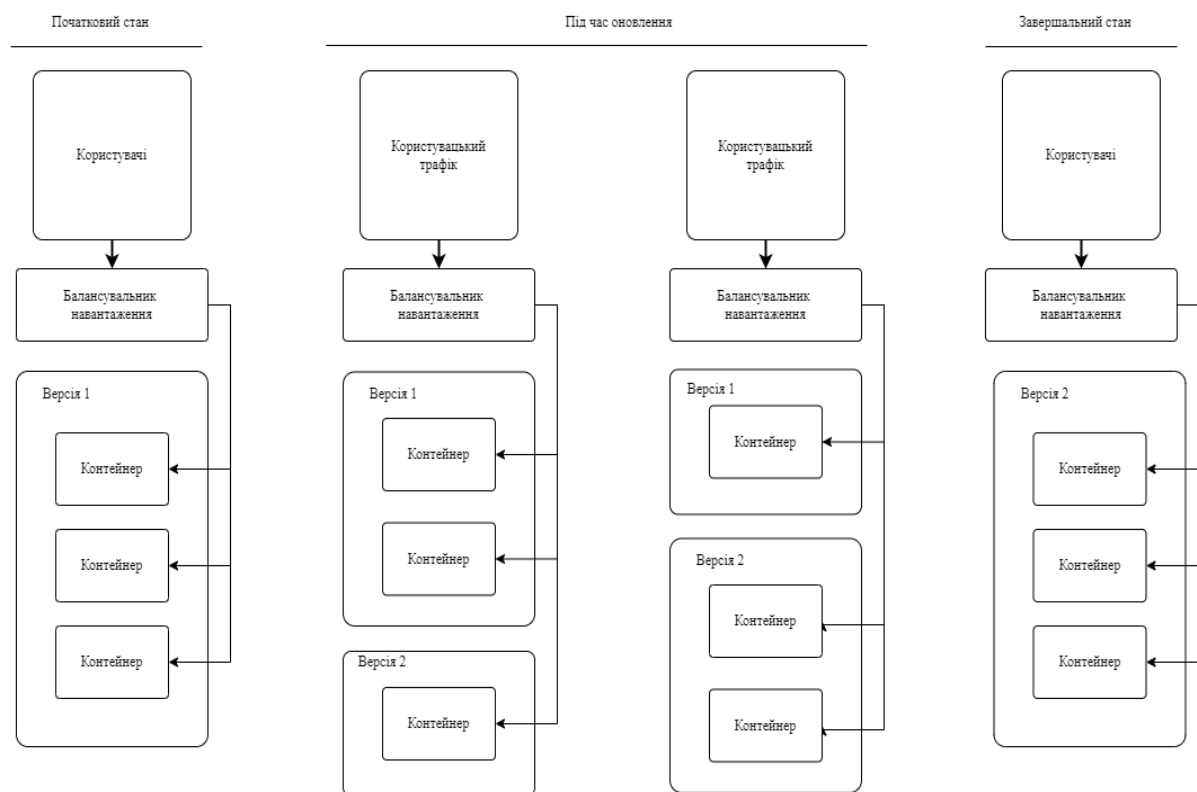


Рисунок 4.1 – Принцип тестування за моделлю Canary

Тестовий шаблон канарки є методом тестування програмного забезпечення, який використовується для виявлення проблем з продуктивністю або масштабованістю до того, як вони стануть серйозними проблемами. Він працює шляхом розгортання нової версії програми разом із робочою версією. Потім частина трафіку відправлялася до нової версії, а частина - до робочої версії. На основі даних про продуктивність нової версії можна визначити, чи є у неї проблеми.

Які ми маємо переваги тестового шаблону канарки:

- можливість тестувати на реальному трафіку. Canary дозволяє тестувати нову версію програми на реальному трафіку користувачів, а не на імітованому трафіку в проміжному середовищі. Це дає більш точне уявлення про те, як нова версія буде працювати в реальному світі;
- швидкий відкат. Якщо Canary виявляє проблеми, можна швидко відкотитися до старої версії програми, не перериваючи роботу користувачів;
- відсутність простоїв. Canary дозволяє оновлювати програму без переривання роботи користувачів.

Наступною моделлю стратегії тестування є тестовий шаблон A/B. Тестовий шаблон A/B - це метод тестування, який використовується для порівняння двох різних версій одного і того ж продукту або послуги. Він працює шляхом випадкового розподілу користувачів між двома групами:

- група A отримує версію A;
- група B отримує версію B.

Правила маршрутизації є механізмом, який визначає, яка версія продукту або послуги буде відображатися користувачеві. Вони можуть бути налаштовані на основі таких факторів, як:

- версія браузера користувача;
- операційна система користувача;
- географічне розташування користувача;
- агент користувача.

Після того, як ви виміряєте результати двох версій, ви можете прийняти рішення про те, яку версію використовувати в робочому середовищі, як наведено на рисунку 4.2.

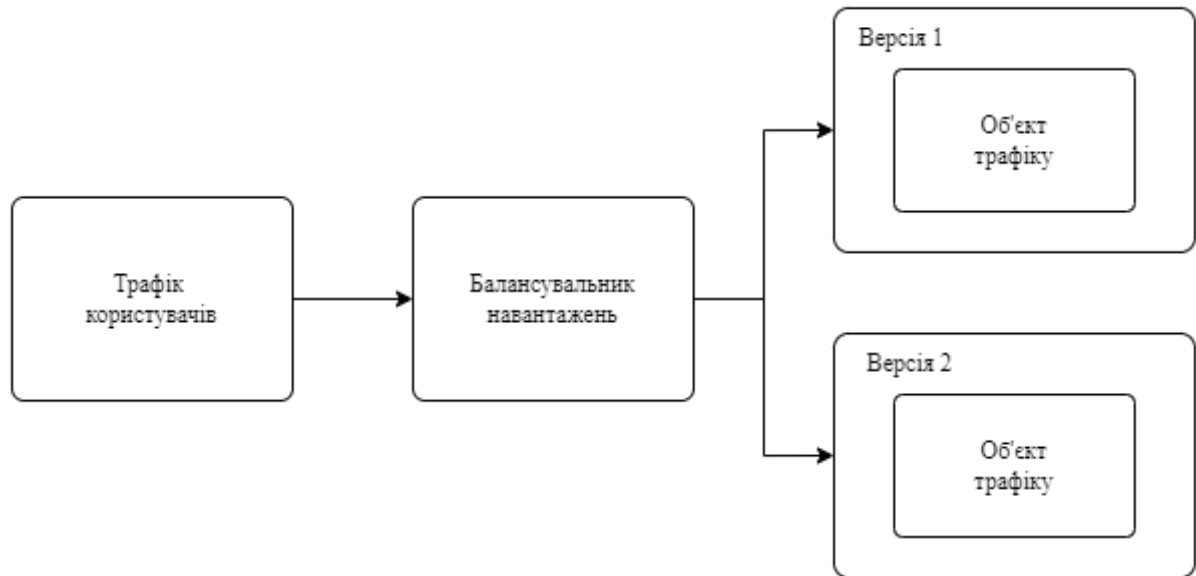


Рисунок 4.2 – Принцип тестування за моделлю A/B

Використання A/B-тестування є оптимальним для оцінки ефективності функціоналу програмного забезпечення. Методологія спрямована на безпечне впровадження нових програмних рішень та передбачуваний відкат. Під час A/B-тестування ви маєте можливість контролювати цільову аудиторію щодо нововведень і виявляти будь-які статистично значущі відмінності в користувацькому поведінці.

Третім методом стратегії тестування, є тіньовий тестовий шаблон. Використання методів послідовних експериментів, таких як тестування на канарках, може призвести до того, що клієнти опиняться на менш ефективній версії програми на початкових етапах тестування. Цей ризик можна управляти за допомогою офлайн-методів, таких як моделювання. Однак офлайн-методи не забезпечують підтвердження покращень програми, оскільки користувач не взаємодіє з новими версіями. Із застосуванням тіньового тестування ви можете впровадити та запустити нову версію, паралельно із поточною, проте зробити це так, щоб нова версія залишалася прихованою від

користувачів, як показано на рисунку 4.3.

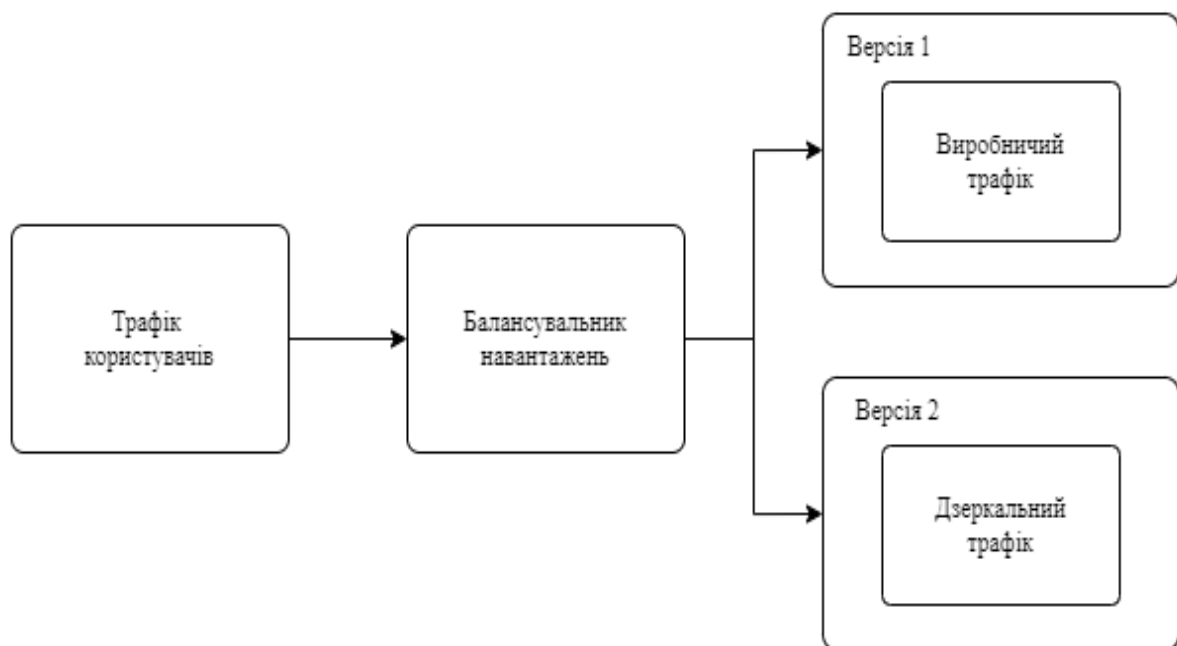


Рисунок 4.3 – Принцип тестування за моделлю тіньового тестового шаблону

Вхідний запит дублюється та відтворюється в тестовому середовищі, і цей процес може відбуватися в режимі реального часу або асинхронно, після того, як копія раніше захопленого робочого трафіку відтворюється на недавно розгорнутій службі. Необхідно впевнитися, що тіньові тести не мають побічних ефектів, які можуть вплинути на існуюче виробниче середовище або стан користувача.

Основні переваги такого методу:

- мінімальний вплив на виробництво. Дякуючи дублюванню трафіку помилки в обробці тіньових даних не впливають на ефективність, забезпечуючи невідхідну стабільність;

- тестування нових функцій бекенду під реальним навантаженням. Застосування інструментів, таких як Diffy, в поєднанні з відстеженням трафіку, дозволяє вимірювати поведінку сервісу в порівнянні з активним трафіком. Це надає можливість виявлення помилок, виключень, визначення продуктивності та порівняння результатів між версіями програми;

- зменшений ризик розгортання. Відстеження трафіку часто поєднується з

іншими методами, такими як тестування на канарках. Після випробування нової функції з використанням відстеження трафіку контролюється її взаємодія з користувачем, поступово впроваджуючи функцію для широкого кола користувачів з часом. Повне впровадження відбувається лише після того, як програма відповідає вимогам стабільності та продуктивності.

Отже, проаналізувавши стратегії тестування розгортання, ми можемо навести порівняльну таблицю (табл.4.1).

Таблиця 4.1 – Порівняння моделей стратегії тестування

| Переваги /<br>Характеристики                                         | Тестовий шаблон<br>Canary | Тестовий шаблон<br>A/B | Тіньовий тестовий<br>шаблон                 |
|----------------------------------------------------------------------|---------------------------|------------------------|---------------------------------------------|
| Можливість<br>тестування на<br>реальному трафіку                     | Так                       | Ні                     | Так, дублювання<br>робочого трафіку         |
| Швидкий відкат у<br>випадку проблем                                  | Так                       | Ні                     | Ні (потрібна повна<br>реалізація)           |
| Відсутність простоїв<br>під час оновлення                            | Так                       | Ні                     | Ні                                          |
| Мінімальний вплив<br>на виробництво                                  | Ні                        | Ні                     | Так                                         |
| Тестування нових<br>функцій бекенду під<br>реальним<br>навантаженням | Так                       | Ні                     | Так, з використанням<br>відстеження трафіку |
| Зменшений ризик<br>розгортання                                       | Так                       | Ні                     | Так (поступове<br>впровадження<br>функцій)  |

Отже, в порівнянні стратегій тестування, тестовий шаблон канарки вирізняється можливістю тестування на реальному трафіку, швидким відкатом та відсутністю простоїв під час оновлень, проте він може не зменшувати ризик

розгортання. Тестовий шаблон A/B дозволяє порівнювати дві різні версії продукту, але не тестує на реальному трафіку та може вимагати великої уваги до правил маршрутизації. Тіньовий тестовий шаблон зменшує ризик розгортання, має мінімальний вплив на виробництво та дозволяє тестування нових функцій під реальним навантаженням, але може вимагати додаткових зусиль для виявлення помилок під час етапу тіньового тестування.

На основі перелічених стратегій тестування, у контексті тестування саме розгортання у середовищі Google Kubernetes Engine, ми скористаємось стратегією Canary, по перше вона є найбільш адаптованою під різні потреби та ризики, а по друге, цю стратегію рекомендують розробники Google Platform у офіційній документації.

## **4.2 Дослідження методів тестування системи**

Спираючись на рекомендації від розробників у офіційній документації платформи. Ми проведемо дослідження тестування двома способами та інструментами [40, 41]. Chef InSpec з Google Cloud Build, та з автоматизованим Canary тестування.

Критерії на які ми будемо посилались у порівнянні, це мета тестування, сценарії використання, зворотній зв'язок щодо результатів тестування, який обсяг трафіку необхідний для проведення тестування, можливість інтеграції зі сторонніми інструментами розгортання ресурсів.

Розглянемо ці критерії та визначимо, чому вони є ключовими при порівнянні обраних інструментів:

Мета тестування:

- Chef InSpec: Спрямований на тестування безпеки та відповідності. Призначений забезпечити відповідність інфраструктури з безпековими політиками;

- Canary тестування: Орієнтоване на поступовий реліз нових версій додатку для оцінки продуктивності та надійності в реальному середовищі.

Сценарії використання:

- Chef InSpec: Ідеально підходить для визначення та перевірки безпекових політик, а також відповідності інфраструктури;

- Canary тестування: Підходить для поступового випуску нових функцій, оцінки їх впливу на користувачів та продуктивність системи.

Тобто, наш вибір залежить від того, на чому ми акцентуємось, на безпеці та відповідності, чи на поступовому розгортанні та зборі реальних відгуків від користувачів. Зворотній зв'язок щодо результатів тестування:

- Chef InSpec: Надає швидкий зворотний зв'язок стосовно безпеки та відповідності зі звітами про тестування;

- Canary тестування: Дозволяє отримувати реальний зворотний зв'язок від користувачів за допомогою поступового релізу.

Обсяг трафіку для проведення тестування:

- Chef InSpec: Може використовуватися незалежно від обсягу трафіку, оскільки спрямований на тестування безпеки та відповідності;

- Canary тестування: Вимагає достатнього трафіку для ефективного оцінювання нових функцій.

Можливість інтеграції зі сторонніми інструментами розгортання ресурсів:

- Chef InSpec: Інтегрується з Terraform для тестування інфраструктури як коду;

- Canary тестування: Може інтегруватися з іншими інструментами розгортання, такими як Tekton та Terraform.

Узагальнюючи, якщо ми зосереджені на безпеці та відповідності, то Chef InSpec є хорошим вибором. Якщо ми зосереджені на поступовому випуску нових функцій та зборі реальних відгуків від користувачів, то Canary тестування є хорошим вибором.

Порівняємо ці два інструменти у таблиці 4.2.

Таблиця 4.2 – Порівняння інструментів тестування

| Критерії                        | Chef InSpec з Google Cloud Build                                        | Canary тестування з Google Cloud Build                                                          |
|---------------------------------|-------------------------------------------------------------------------|-------------------------------------------------------------------------------------------------|
| Мета                            | Тестування безпеки та відповідності                                     | Інкрементальне розгортання та зворотний зв'язок від реальних користувачів                       |
| Сценарії використання           | Ідеально підходить для середовищ з вимогами до безпеки та відповідності | Підходить для додатків, де важливий інкрементальний реліз та зворотний зв'язок від користувачів |
| Зворотний зв'язок               | Швидкий зворотний зв'язок щодо безпеки та відповідності                 | Забезпечує реальний зворотний зв'язок від користувачів та їхньої взаємодії з новою версією      |
| Використання трафіку            | Не залежить від обсягу трафіку                                          | Ефективне тільки при наявності достатнього обсягу трафіку для отримання значущих результатів    |
| Інтеграція з Tekton і Terraform | Інтеграція через Terraform для тестування IaC                           | Можлива інтеграція з Tekton та Terraform для реалізації canary тестування                       |

Головна відмінність цих двох інструментів у тому, що вони переслідують різну мету. Chef InSpec і Canary тестування відрізняються в основних цілях і областях застосування. Chef InSpec спеціалізується на тестуванні безпеки та відповідності, зорієнтований на перевірку інфраструктури згідно з безпечними стандартами.

З іншого боку, Canary тестування створене для оцінки впливу нових функцій в реальному середовищі через поступовий їх реліз для обмеженої групи користувачів (канарейок). Вибір між ними залежить від основних цілей тестування: Chef InSpec підходить для інфраструктурного тестування безпеки, тоді як Canary тестування використовується для ефективного впровадження нового функціоналу з оцінкою його впливу в реальному часі.

Оскільки, наша мета, це тестування розгортання інфраструктури, наш вибір зупиняється на інструменті Chef InSpec.

Отже, ми провели дослідження методів та інструментів тестування системи, згідно з рекомендаціями офіційної документації з тестування застосунків від розробників Google Cloud Platform.

### **4.3 Тестування системи**

Для подальшого проведення дослідження тестування використаємо платформу з відкритим вихідним кодом для тестування та аудиту програм та інфраструктури Chef InSpec. Використовувати Chef InSpec рекомендовано саме офіційним розробником Google Cloud Platform.

Тестування розгортання застосунку є важливим етапом у процесі розробки та розгортання застосунків. Важливо розробити тестові сценарії, які будуть перевіряти всі важливі аспекти розгортання застосунку. Також важливо автоматизувати тестування, щоб гарантувати, що тести будуть виконуватися регулярно та у стабільному середовищі.

Етапи тестування розгортання нового застосунку передбачає створення Chef InSpec тестів, які перевіряють, чи встановлені та налаштовані всі компоненти системи коректно [42]. При тестуванні оновлення існуючого застосунку необхідно враховувати сумісність нової версії з іншими компонентами системи, а також виконати всі необхідні кроки для оновлення, визначені в конфігурації. Тестування відновлення застосунку після збоїв включає створення тестів Chef InSpec для перевірки автоматичного відновлення системи та її здатності повертатися до працездатного стану після збоїв. А на завершення, етап інтеграції тестів у CI/CD процес, це дозволяє автоматизовано виконувати тести при кожному внесенні змін, забезпечуючи стабільність та надійність системи протягом її життєвого циклу.

Початком дослідження, варто відобразити алгоритм тестування, який наведено на рисунку 4.4.

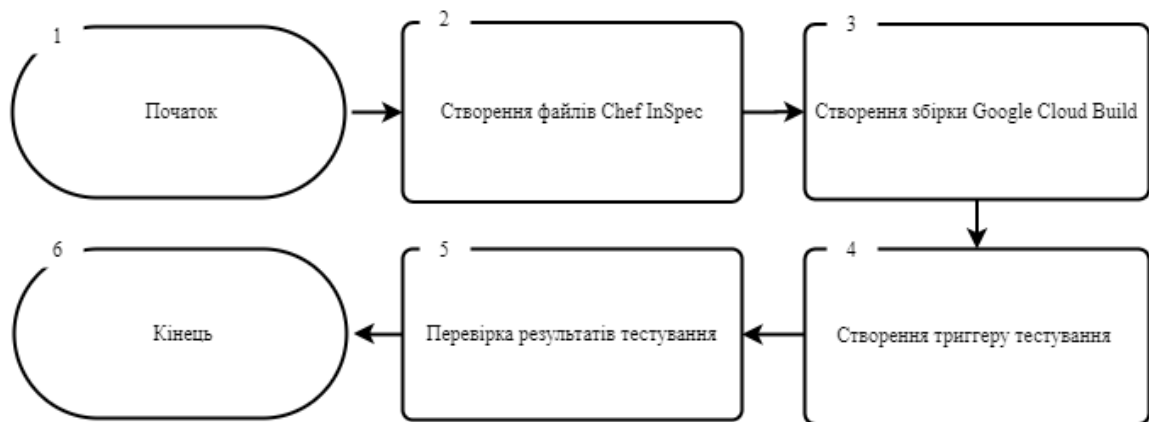


Рисунок 4.4 – Алгоритм тестування з Chef InSpec

Метою наступного етапу є процес створення файлів Chef InSpec, які будуть використовуватися для тестування системи. Ці файли повинні містити набори перевірок, які перевіряють, чи відповідає система очікуванням. Для ефективного створення файлів Chef InSpec та реалізації тестування системи, важливо докладно описати очікування щодо функціональності та безпеки. Спершу рекомендується провести аналіз вимог до системи та визначити ключові аспекти, які потрібно перевірити. Наприклад, це може включати в себе правильність конфігурації мережі, налаштування безпеки, аутентифікацію користувачів та інші аспекти, важливі для відпрацювання коректної роботи системи, але нашим пріоритетом є тестування саме розгортання.

Створення нового каталогу для файлів Chef InSpec. Цей каталог можна створити в будь-якому місці, але зазвичай його створюють у кореневому каталозі проекту. У новому каталозі створюється файл з назвою `spec.rb`. Цей файл буде містити основні перевірки для тестування системи. У файлі `spec.rb` лістингу 4.1, необхідно додати перевірки, які потрібно виконати.

#### Лістинг 4.1 – Конфігурація файлу `spec.rb`

```
describe docker_container('my-app') do
```

```

    its('image') { should eq 'my-app:latest' }
    its('ports') { should include(80) }
end

```

Перед запуском тестування необхідно перевірити файли Chef InSpec, щоб переконатися, що вони скомпільовані правильно. Для цього можна використовувати команду `chef-inspec exec spec.rb`. Якщо команда не виводить жодних помилок, це означає, що файли Chef InSpec скомпільовані правильно і готові до запуску.

Далі необхідно створити збірку Google Cloud Build, яка буде запускати файли Chef InSpec. Збірка Google Cloud Build - це автоматизований процес, який може запускатися за розкладом або за подією. Створення збірки відбувається переходячи ручним способом до інтерфейсу Google Cloud, так клікаючи "Створити збірку", де нам необхідно вибрати тип збірки "Виконання сценарію".

Наступний етап, створення тригера тестування. Тригер тестування - це механізм, який дозволяє запускати збірку Google Cloud Build за певною подією. При створенні нам необхідно вказати репозиторій, у якому зберігаються файли Terraform.

Для запуску збірки Google Cloud Build після кожного розгортання застосунку, який зберігається в репозиторії Git. Результати тригера можна переглянути у інтерфейсі на сторінці тригери, а сам тригер можна запустити вручну, натиснувши кнопку запустити, але також, тригер також можна запустити за розкладом або за подією.

Результати тестування відображаються у консолі Google Cloud Build. Якщо результати тестування негативні, необхідно провести додаткове дослідження для виявлення причини несправності.

Необхідно знайти збірку у інтерфейсі Google Cloud, за фільтрацією на сторінці знайти файл який запускав файли Chef InSpec. На цій вкладці результатів можна переглянути результати тестування, які відображаються у вигляді таблиці.

Наприклад, якщо збирання завершилося успішно, то у таблиці результатів тестування може бути відображено наступне (табл. 4.3).

Таблиця 4.3 – Позитивний результат тестування

| Статус | Перевірка                                           | Результат                              |
|--------|-----------------------------------------------------|----------------------------------------|
| Успіх  | Контейнер Docker застосунку повинен слухати порт 80 | Docker container my-app слухає порт 80 |

Якщо збирання завершилося з помилкою, у таблиці результатів тестування відображено наступне (табл. 4.4).

Таблиця 4.4 – Негативний результат тестування

| Статус  | Перевірка                                           | Результат                                         |
|---------|-----------------------------------------------------|---------------------------------------------------|
| Помилка | Контейнер Docker застосунку повинен слухати порт 80 | Docker container my-app не відповідає очікуванням |

У разі помилки необхідно провести додаткове дослідження для виявлення причини несправності. Наприклад, можна перевірити, чи правильно написані перевірки Chef InSpec, чи правильно розгорнуто застосунок, або чи є інші проблеми в системі.

Отже, ми провели дослідження етапів автоматизованого тестування застосунку у GKE за допомогою Chef InSpec та Google Cloud Build. На першому етапі ми створили файли Chef InSpec, який може містити набори перевірок для тестування нашого застосунку. Ці перевірки перевіряють, чи відповідає застосунок нашим вимогам, наприклад, чи використовується правильний образ Docker, чи працює застосунок на правильному порті. На другому етапі ми створили збірку Google Cloud Build, яка буде запускати файли Chef InSpec. Ця збірка запускається автоматично після кожного розгортання застосунку, що

дозволяє нам швидко і легко перевіряти якість застосунку після будь-яких змін. На третьому етапі ми дослідили створення тригера тестування, який запускає збірку Google Cloud Build після кожного розгортання застосунку. Цей тригер дозволяє нам автоматизувати процес тестування і гарантувати, що застосунок буде тестуватися після кожного розгортання. І на останок ми перевірили результати тестування, які відображаються у консолі Google Cloud Build. В цілому, такий підхід дозволяє нам забезпечити якість застосунку та скоротити час розробки та впровадження.

## ВИСНОВКИ

В процесі виконання дослідження поставлені задачі виконані. Дослідили принципи та методи для системи розгортання застосунків у хмарних платформах за методом контейнеризації. При аналізі та дослідженні враховані особливості конкретних хмарних платформ, таких як Amazon Web Services, Microsoft Azure, Google Cloud Platform, з використанням допоміжних інструментів, як Kubernetes, Terraform by HashiCorp, Tekton CI/CD Pipelines, Docker Image. Такі рішення дозволять скоротити час на проведення розгортання, чим би це відбувалось методом віртуалізації на віддаленому суперкомп'ютері без провайдеру, та дозволяє скоротити витрати на обслуговування такої системи.

Хмарні платформи ми розглянули з точки зору оптимальності та вартості, як початкового періоду роботи на платформі хмарного провайдеру, так і у подальшому обслуговуванні системи.

Також ми провели дослідження та аналіз інструментів Terraform, Docker, Habitat та Ansible для автоматизації розгортання образів застосунку у контейнері та Tekton CI/CD, Jenkins, Bitbucket Pipelines та GitHub Actions для розробки та доставлення коду у робоче середовище. Завдяки автоматизації ми можемо налаштувати автоматизовану систему розгортання, розробки та супроводження програм у системі, завдяки підходу контейнеризації, ми маємо змогу розгорнути мікросервісні частини програми у кожному контейнері окремо, а контейнери у свою чергу у кластерах. Такий підхід забезпечує працезданість, протиаварійність та підтримуванність. Але має недолік у тому, що чим більше система, тим більш складніший процес обслуговування застосунку, оскільки велика кількість модулів може призвести до конфліктів пакетів, чим більше пакетів, тим більше існує можливість не сумісності.

В роботі проведено дослідження принципів розгортання застосунку у контейнері кластера на основі Kubernetes на базі платформи Google Kubernetes Engine. Також розглянуто можливість впровадження тестування системи

розгортання та інтеграції на базі хмарного провайдеру, а саме у системі Google Cloud Build з інтеграцією платформи Chef InSpec для автоматизації процесів тестування.

Таким чином, впроваджено систему розгортання застосунків у хмарі з використанням інструментів автоматизації. Реалізована система має переваги, в порівнянні з існуючим методом розгортання у віртуалізованій машині, що означає що така система може бути запроваджена для широкого використання.

## ПЕРЕЛІК ДЖЕРЕЛ ПОСИЛАННЯ

- 1 Rudkovskyi O.R., Kirichek G.G. Interaction support system of network applications, CEUR-WS, Vol-2832, November 27, 2020. P. 11–23.
- 2 Дем'яненко В.В. Моделювання процесів побудови та розгортання хмарних сервісів, 2021. 3 – 11 с.
- 3 Aaqib R., Amit C. Cloud Computing Characteristics and Services: A Brief Review, IJCSE, 2019. 421-426 p.
- 4 Киричек Г.Г., Гаркуша В.Ю. Віртуалізація хостів на основі Proxmox VE в умовах надлишкового використання ресурсів. Вчені записки ТНУ імені В.І. Вернадського. Серія: Технічні науки, 2021. Т. 32 (71) № 1. 78-84 с.
- 5 Rashid A., Chaturvedi A. Cloud computing characteristics and services: a brief review. International Journal of Computer Sciences and Engineering. 2019. Т. 7. №. 2. 421–426 p.
- 6 A tale of two systems: Using containers to deploy HPC applications on supercomputers and clouds / Andrew J., Pedretti K., Ryan E., Brightwell R. (2017, December). In 2017 IEEE International Conference on Cloud Computing Technology and Science (CloudCom), 2017. 74-81 p.
- 7 Zhou N. Container orchestration on HPC systems through Kubernetes. Journal of Cloud Computing, 2021. 1-14 p.
- 8 Petrosyan D., Astsatryan H. Serverless High-Performance Computing over Cloud. Cybernetics and Information Technologies, 2022. 82-92 p.
- 9 Ben-Kiki O., Evans C., Ingerson B. Yaml ain't markup language (yaml™) version 1.1. Working Draft 2008. 2009. 11 p.
- 10 Glushkova D., Jovanovic P., Abelló A. Mapreduce performance model for Hadoop 2.x. Information systems. 2019. 32-43 p.
- 11 Salloum S. Big data analytics on Apache Spark. International Journal of Data Science and Analytics. 2016. 145-164 p.
- 12 Смірнов В.В., Киричек Г.Г. Дослідження застосування контейнерних

технологій для розгортання програм на суперкомп'ютерах. Тиждень науки-2023. Факультет комп'ютерних наук і технологій. Тези доповідей науково-практичної конференції, Запоріжжя, 24-28 квітня 2023 р. Запоріжжя: НУ «Запорізька політехніка», 2023. 87-88 с.

13 Киричек Г.Г., Смірнов В.В., Тягунова М.Ю. Дослідження застосування контейнерних технологій для розгортання програм на суперкомп'ютерах. Вісник Кременчуцького національного університету імені Михайла Остроградського. Кременчук: КрНУ, 2023. Випуск 3 (140). 29-35 с. <https://doi.org/10.32782/1995-0519.2023.3>.

14 Allison R. The Ideal Versus the Real: Revisiting the History of Virtual Machines and Containers. 2020. 12-15 p.

15 Rodrigo G. Towards understanding HPC users and systems: a NERSC case study. Journal of Parallel and Distributed Computing. 2018. 206-221 p.

16 Palos-Sanchez P., Arenas-Marquez F., Aguayo-Camacho M. Cloud computing (SaaS) adoption as a strategic technology: Results of an empirical study. Mobile Information Systems. 2017. 2-11 p.

17 Bharany S. Efficient Middleware for the Portability of PaaS Services Consuming Applications among Heterogeneous Clouds. Sensors. 2022. 1-16 p.

18 Adhikari M., Amgoth T. Heuristic-based load-balancing algorithm for IaaS cloud. Future Generation Computer Systems. 2018. 156-165 p.

19 Chnar M., Subhi R. Zeebaree Sufficient Comparison Among Cloud Computing Services: IaaS, PaaS, and SaaS: A Review. 2021 20-24 p.

20 Ramesh N. A High-Availability and Integrity Layer for Cloud Storage, Cloud Computing Security: From Single to Multi-Clouds. 2021. 3-10 p.

21 Junlong L., Chenghong G. Edge-cloud Computing Systems for Smart Grid: State-of-the-art, Architecture, and Applications. 2022. 4-10 p.

22 Vijaya S., Udaya S. Key Decisions to Design a Secure Cluster in GKE Cloud. 2020. 1-4 p.

23 Pallavi W., Minaiy T., Rutuja C. A comparative study of cloud platforms - Microsoft Azure, Google Cloud Platform, and Amazon EC2. 2020. 61-63 p.

24 Yining O. Maximizing Resource Utilization in GKE: A Case Study on IP Address Management and Budget Allocation in Google Kubernetes Engine Config Page. 2023. 1-3 p.

25 The essential deployment metamodel: a systematic review of deployment automation technologies / Michael W., Uwe B., Michael F., Christoph K., Frank L., Karoline S., Jacopo S. 2021. 2-9 p.

26 He L., Lu Y., Wu H. The Impact of GDPR on Global Technology Development. Journal of Global Information Technology Management. 2019. 1-4 p.

27 Wilnellys M., Sarah F. Review of HIPAA, Part 1: History, Protected Health Information, and Privacy and Security Rules. 2019. 1-4 p.

28 Parul D., Arvind K. Amazon Web Services: the Definitive Guide for Beginners and Advanced Users. 2023. 3-8 p.

29 Jusuf Q. Using Cloud Computing as an Infrastructure Case Study- Microsoft Azure. 2020. 2-6 p.

30 Khalil M., Adadurov S., Mahmood M. Mastering Google cloud: building the platform that serves your needs. 2020. 41-44 p.

31 Prassanna J., Neelanarayanan V. A review of existing cloud automation tools. 2017.

32 Russ M. Learn Ansible: Automate cloud, security, and network infrastructure using Ansible 2.x. 2018. 9- 14 p.

33 Janani K., Anuhya K., Manaswini V., Likitha V., Suneetha B., Vignesh T. Analysis of CI/CD Application in Kubernetes Architecture. 2022. 1-5 p.

34 Tekton Overview. URL: <https://tekton.dev/docs/concepts/overview/> (date of access: 29.10.2023).

35 Head F. Патерни проєктування / Е. Фрімен та ін., пер. з англ. Г. Якубовська. Харків : Фабула, 2022. 672 с.

36 Катренко А. В., Пасічник В. В. Прийняття рішень: теорія та практика : підруч. Львів : Новий світ – 2000, 2023. 447 с.

37 Joni V. Comparing Different CI/CD Pipelines. 2021. 21-51 p.

38 Andreas Faanes Slagsvold. Exploring CI/CD Pipelines for Cloud

Infrastructure Deployment Can one increase efficiency through amalgamation? 2023. 35-44 c.

39 Application deployment and testing strategies. URL: <https://cloud.google.com/architecture/application-deployment-and-testing-strategies> (date of access: 10.11.2023).

40 Patterns for automated compliance testing using Chef InSpec. URL: <https://cloud.google.com/architecture/patterns-automated-compliance-testing-chef-inspec> (date of access: 18.11.2023).

41 Implementing Cloud Run canary deployments with Git branches and Cloud Build. URL: [https://cloud.google.com/architecture/implementing-cloud-run-canary-deployments-git-branches-cloud-build#automate\\_canary\\_testing](https://cloud.google.com/architecture/implementing-cloud-run-canary-deployments-git-branches-cloud-build#automate_canary_testing) (date of access: 19.11.2023).

42 Daniel S., Matthias V., Maneendra P., Klaus T. Exploring the Test Driven Development of a Fraud Detection Application using the Google Cloud Platform. 2022. 6-9 p.