Міністерство освіти і науки України
Харківський національний університет радіоелектроніки

Факультет комп’ютерних наук (або центр післядипломної освіти, або навчально-науковий центр заочної форми навчання)
(повна назва)

Кафедра програмної інженерії
(повна назва)

# КВАЛІФІКАЦІЙНА РОБОТА
# Пояснювальна записка

рівень вищої освіти другий (магістерський)

Дослідження методів та технологій розгортання та управління інфраструктурою хмарних застосунків
(тема)

Виконав:
студент (ка) 2 курсу, групи ІПЗм-22-1

Литовченко В.Ю
(прізвище, ініціали)

Спеціальність 121 – Інженерія програмного забезпечення
(код і повна назва спеціальності)

Тип програми освітньо-наукова

Керівник доц. Бабій А.С.
(посада, прізвище, ініціали)

Допускається до захисту
Зав. кафедри ____________ З.В.Дудар
(підпис) (прізвище, ініціали)

2024 р.

Харківський національний університет радіоелектроніки

Факультет комп'ютерних наук (або центр післядипломної освіти, або навчально-науковий центр заочної форми навчання)
Кафедра програмної інженерії
Рівень вищої освіти другий (магістерський)
Спеціальність 121 – Інженерія програмного забезпечення
Тип програми освітньо-наукова програма
Освітня програма Інженерія програмного забезпечення
(шифр і назва)

ЗАТВЕРДЖУЮ:
Зав. кафедри ________
(підпис)
«___» ________ 2024 р.

**ЗАВДАННЯ**
НА КВАЛІФІКАЦІЙНУ РОБОТУ

студентові Литовченку Владиславу Юрійовичу
(прізвище, ім'я, по батькові)

1. Тема роботи «Дослідження методів та технологій розгортання та управління інфраструктурою хмарних застосунків»

Затверджена наказом по університету від 29.03.2024р. № 250 Ст

2. Термін подання студентом роботи до екзаменаційної комісії 17.06.2024 р.

3. Вихідні дані до роботи порівняльна характеристика методів та інструментів створення хмарної інфраструктури, пояснювальна записка, практична реалізація для управління хмарною інфраструктурою на базі Crossplane

4. Перелік питань, що потрібно опрацювати в роботі

історія та тенденції в сфері хмарних обчислень, роль та проблеми в сфері створення хмарної інфраструктури, аналіз та порівняння існуючих методів автоматизації процесу, визначення найкращого методу та інструменту, практична реалізація на реальному прикладі

# КАЛЕНДАРНИЙ ПЛАН

| № | Назва етапів роботи | Термін виконання етапів роботи | Примітка |
|---|---|---|---|
| 1 | Аналіз предметної галузі та постановка задачі | 23.03 – 18.04.24 | *виконано* |
| 2 | Аналіз та порівняння існуючих методів та принципів | 18.04 – 07.05.24 | *виконано* |
| 3 | Теоретичне дослідження та порівняння IaC інструментів | 07.05 – 20.05.24 | *виконано* |
| 4 | Проведення експериментів та практичне порівняння IaC інструментів | 20.05 – 25.02.24 | *виконано* |
| 5 | Програмна реалізація хмарного провайдера на прикладі Crossplane | 25.05 – 02.06.24 | *виконано* |
| 6 | Написання пояснювальної записки | 23.03 – 04.06.24 | *виконано* |
| 7 | Підготовка презентації та доповіді | 03.06 – 11.06.24 | *виконано* |
| 8 | Нормоконтроль | 08.06.24 | *виконано* |
| 9 | Рецензування | 13.06.24 | *виконано* |
| 10 | Занесення диплома в електронний архів | 15.06.24 | *виконано* |
| 11 | Попередній захист | 15.06.24 | *виконано* |
| 12 | Допуск до захисту у зав. кафедри | 17.06.24 | *виконано* |

Дата видачі завдання __________р.

Студент (ка) __________________ Литовченко В.Ю.
(підпис)

Керівник роботи __________________ доц. Бабій А.С
(підпис) (посада, прізвище, ініціали)

**РЕФЕРАТ / ABSTRACT**

Пояснювальна записка містить: 105 с., 24 рис., 5 табл., 28 джерел.

АВТОМАТИЗАЦІЯ, ІНФРАСТРУКТУРА, КОНФІГУРАЦІЯ, ХМАРНІ ОБЧИСЛЕННЯ, CI/CD, CROSSPLANE, DEVOPS, INFRASTRUCTURE AS A CODE, TERRAFORM.

Метою роботи є дослідження методів та інструментів для автоматизації створення хмарної інфраструктури.

В ході роботи було проаналізовано сучасні проблеми, що виникають при створенні та управлінні інфраструктурою, методи вирішення цих проблем, проведено аналітику цих методів, теоретично та практично порівняно конкретні інструменти згідно кращого методу)

AUTOMATION, INFRASTRUCTURE, CONFIGURATION, CLOUD COMPUTING, CI/CD, CROSSPLANE, DEVOPS, INFRASTRUCTURE AS A CODE, TERRAFORM.

The aim of the work is to study methods and tools for automating the creation of cloud infrastructure

In the course of the work, we analyzed the current problems that arise when creating and managing infrastructure, methods for solving these problems, analyzed these methods, and compared specific tools theoretically and practically according to the best method.

Заява щодо самостійного виконання кваліфікаційної роботи та можливості її публікації в електронному архіві відкритого доступу ElArKhNURE.

Я, Литовченко Владислав Юрійович, студент гр. ІПЗм-21-1, здобувач вищої освіти на другому (магістерському) рівні кафедри «Програмна інженерія», заявляю: моя кваліфікаційна робота на тему «Дослідження методів та технологій розгортання та управління інфраструктурою хмарних застосунків», що буде представлена в екзаменаційну комісію для публічного захисту, виконана самостійно, в ній не містяться елементи плагіату і вона може бути опублікована в електронному архіві відкритого доступу ElArKhNURE. Всі запозичення з друкованих та електронних джерел мають відповідні посилання.

Я ознайомлений(на) з діючим положенням «Про протидію академічному плагіату в ХНУРЕ», згідно з яким виявлення плагіату є підставою для відмови в допуску кваліфікаційної роботи до захисту та застосування дисциплінарних заходів.

# ЗМІСТ


Вступ ........ 8
1 Аналіз предметної галузі ........ 9
1.1 Історія хмарних обчислень ........ 9
1.2 Сучасні тенденції в розвитку хмарних обчислень ........ 11
1.3 Принцип роботи хмари та надання хмарних послуг ........ 14
1.4 Типи надання хмарних послуг ........ 17
2 Постановка задачі ........ 22
2.1 З чого складається хмарна інфраструктура ........ 22
2.2 Проблеми при створенні хмарної інфраструктури ........ 25
2.3 Задача дослідження ........ 28
3 Порівняння методів та принципів створення інфраструктури ........ 28
3.1 Infrastucture as a code ........ 29
3.2 Декларативні та імперативні підходи ........ 30
3.3 Скрипти та інтерфейс командного рядка ........ 32
3.4 Source Development Kits ........ 33
3.5 Cloud Development Kits ........ 36
3.6 IaC vendor-neutral інструменти ........ 40
3.7 Порівняння методів ........ 45
4 Порівняння vendor-neutral external IaC рішень ........ 49
4.1 Terraform ........ 49
4.2 Crossplane ........ 51
4.3 Порівняння Crossplane і Terraform ........ 54
4.3.1 Теоретичне порівняння ........ 54
4.3.2 Практичне порівняння ........ 56
5 Практична реалізація провайдера ........ 62
5.1 Вибір хмарного постачальника для дослідження ........ 62
5.2 Процес реалізації ........ 63

5.3 Інтеграція DevOps та GitOps ........ 67
5.3.1 Приклад розгортання за допомогою розробленого рішення ........ 68
5.3.2 Висновки за проведеною реалізацією ........ 72
Висновки ........ 73
Перелік джерел посилання ........ 75
Додаток А. Перелік джерел посилання науковими напрямами керівника та науковців кафедри Програмної інженерії ........ 79
Додаток Б. Слайди презентації ........ 80
Додаток В. Апробація у вигляді тез ........ 95
Додаток Г. Фрагменти коду ........ 101
Додаток Д. Звіт результатів перевірки на унікальність тексту в базі ХНУРЕ ..... 104
Додаток Е. Експертний висновок результатів перевірки кваліфікаційної роботи на відповідність оформлення вимогам ДСТУ 3008:2015 ........ 105

# ВСТУП

Дослідження методів та технологій розгортання та управління інфраструктурою хмарних застосунків є вкрай актуальним у сучасному цифровому світі. Глобальний перехід від традиційних віддалених серверів до хмарних рішень відбувається швидкими темпами, а це означає, що потреба в розумінні і контролі над цими новими технологіями росте.

Зростання популярності хмарних сервісів та застосунків створює необхідність у вдосконаленні методів їх розгортання і управління для забезпечення ефективності, безпеки та надійності інфраструктури.

Однією з головних проблем є необхідність постійно пристосовувати та оптимізовувати інфраструктуру під змінні потреби бізнесу. Застосунки повинні бути легко масштабовані та гнучкими, а їх управління повинно забезпечувати швидке впровадження нових функцій та забезпечення високої доступності.

Нарешті, в контексті розгортання хмарних застосунків важливо розвивати інструменти автоматизації для швидкого та ефективного управління ресурсами. Це включає в себе розгортання, моніторинг, оновлення та масштабування ресурсів, щоб забезпечити їх оптимальне використання.

Отже, проблематика цієї теми стосується не лише технічних аспектів розгортання та управління інфраструктурою, а й включає в себе питання безпеки, масштабованості та ефективного використання ресурсів, що стає все більш критичним у сучасному інформаційному середовищі.

Важливо вивчати нові стратегії розгортання, спрямовані на швидкість та ефективність, а також вирішення завдань забезпечення безпеки в умовах зростаючого обсягу обробки даних в хмарних сервісах. Дослідження автоматизованих методів управління інфраструктурою сприяє розробці рішень для швидкого реагування на зміни та оптимізації використання ресурсів.

Узагальнюючи, дослідження в даній темі спрямоване на аналіз інструментів та методів, які дозволяють організаціям максимально використовувати переваги ресурсів хмарної інфраструктури для розгортання своїх систем, забезпечуючи при цьому, ефективність та гнучкість управління інфраструктурою.

# 1 АНАЛІЗ ПРЕДМЕТНОЇ ГАЛУЗІ

## 1.1 Історія хмарних обчислень

Для того, щоб детально зрозуміти концепт і проблематику будь-якої сфери, найкращим рішенням більш детально познайомитися з її історією та повернутися до витоків. Історія дозволяє нам проникнутися контекстом, з'ясувати, які фактори та події сприяли формуванню даної концепції, і зрозуміти її еволюцію з часом. Тому, щоб провести достовірне дослідження, спочатку коротко розглянемо етапи становлення такого поняття як «хмарні обчислення» з самих витоків.

На початку 1960-х років необхідність спільного використання одного мейнфрейму була актуальною темою. У цей момент такі потужні обчислювальні машини стають все більш популярними. Але в той же час вони були непомірно дорогими для покупки і підтримки, тому виникла необхідність в загальному використанні, щоб ділити та зменшувати витрати.

В той час дуже популярною стала технологія «time-sharing», розроблена компанією IBM. Вона дозволяла спільно використовувати один комп’ютер для десятків користувачів одночасно. Вони використовували термінали для входу через телефонні лінії. Машини з за такою технологією не з’єднувалися один з одним, але забезпечували різні функції, наприклад, писати та компілювати програми в мейнфреймі, переглядати та редагувати набори даних тощо.

У 1961 році Джон Маккарті був першим, хто публічно заявив, що технології на кшталт «time-sharing» ведуть до майбутнього, в якому обчислювальні потужності і навіть певні програми можна буде продавати через бізнес-модель публічного доступу або “комунальних послуг”. Це виглядало як цікава ідея, але, як і всі геніальні ідеї, вона була попереду часу, оскільки протягом наступних десятків років люди та технології елементарно ще не були готові до неї.

У 1972 році IBM починає популяризувати ідею віртуалізації та віртуальних машин. У віртуалізованих операційних системах одна фізична машина може запускати кілька віртуальних машин, кожна з яких обробляє різні обчислювальні завдання незалежно. Спільне використання обчислювальних ресурсів у такий

спосіб стало набагато ефективніше та дешевше, ніж їх минула технологія для розподілення мейнфреймів – «time-sharing» [1].

У 1999 році компанія Salesforce стала одним із перших прикладів вдалого використання хмарних обчислень. Вони очолили процес використання Інтернету для безпосередньої доставки програмного забезпечення кінцевим користувачам. Ці програми, стали доступними для завантаження будь-якому користувачу з Інтернет-підключенням.

Також ключову роль в розвитку хмарних технологій відіграла компанія Amazon. Після 1999 року Amazon вирішив розширити свій вплив та продавати в інтернеті не тільки книги, а й інші речі, наприклад, одяг. Але, на відміну від книг, одяг міг бути різного кольору, розміру та підходити різним покупцям. Розробникам приходилось постійно створювати новий функціонал, щоб покрити будь-які конфігуровані параметри для речей, які вони продавали. Також вони почали додавати функціонал для полегшення процесу онлайн покупок. Під цим мається на увазі списки улюблених товарів, списки товарів схожих на ті, які знаходяться в кошиках користувачів або були ними придбані, тощо. Під час цього процесу виникла головна проблема – монолітна архітектура. Поточна архітектура заважала компанії розвивати розбробку програмного забезпечення такими ж швидкими темпами, як приходили нові ідеї. Тому Amazon довелося розбити свою монолітну архітектуру на мікросервіси і придумати інфраструктуру, щоб легко створювати потужності та розгортатаи на них мікросервіси.

В результаті до 2006 року вони винайшли хмарну інфраструктуру для орендування, де користувачі могли сплачувати щомісячну або щорічну плату, щоб уникнути проблем з покупкою, встановленням і цілодобовою підтримкою власних дата-центрів. Замість цього, інженери Amazon Web Services (AWS) робили б це у фоновому режимі, ознаменовуючи початок ери Інфраструктури-як-послуги (IaaS). На той час користувачам були доступні такі сервіси: Amazon S3, Amazon EC2, Amazon RDS.

У тих же роках Google створив свої служби Google Docs. Користувачі могли зберігати, змінювати та передавати документи за допомогою платформи блогів

Writely, яку нещодавно придбала Google. Веб-додаток під назвою Google Spreadsheets (придбаний у компанії 2Web Technologies у 2005 році) дозволяв користувачам створювати, редагувати та ділитися електронними таблицями з іншими в Інтернеті. Він використовував програмне забезпечення на основі Ajax, яке працює з Microsoft Excel.

У 2007 році IBM і Google спільно з кількома університетами створили серверну ферму, призначену для дослідницьких проектів, яким були необхідні потужні процесори і колосальні обсяги даних. Університет Вашингтона став першим піддослідним, використовуючи ресурси, запропоновані Google і IBM. Потім його прикладу послідували MIT, Університет Карнегі-Меллона, Стенфордський університет, Каліфорнійський університет та інші. Ці академічні заклади швидко зрозуміли, що комп’ютерні експерименти можна проводити швидше з меншими витратами за підтримки Google і IBM [2]. Рік також ознаменував появу Netflix, що пропонував послуги розповсюдження відео за допомогою застосування потужностей хмар.

## 1.2 Сучасні тенденції в розвитку хмарних обчислень

Останні декілька років стали свідками безпрецедентного зростання хмарних технологій. Пандемія COVID-19 лише прискорила цей процес, адже компанії по всьому світу були змушені перейти на віддалену роботу та онлайн-торгівлю. Це призвело до різкого зростання попиту на хмарні сервіси, такі як зберігання даних, обчислювальні потужності та програмне забезпечення як послуга.

Одним з основних чинників, що спровокував такий ріст – популяризація віддаленої роботи. Перехід на «remote» змусив багато компаній шукати рішення для безпечного та надійного доступу до даних та програмного забезпечення для своїх співробітників. Хмарні технології стали ідеальним рішенням цієї проблеми, надаючи доступ до ресурсів з будь-якої точки світу.

Іншою причиною, що надала прискорення розповсюдження використання хмар, стало збільшення обсягів даних. У сучасному data-driven світі, кількість інформації збільшується з кожним днем. Такий ріст провокує необхідність в

постійному збільшенні потужностей компаній. Зберігання та обробка такої кількості інформації на локальних серверах стає все складнішим і дорожчим. Кожний бізнес хоче зменшувати витрати та збільшувати дохід, і хмарні технології як раз і пропонують масштабовані та економічно вигідні рішення для цього.

Дослідження, проведене платформою баз даних Couchbase, показало, що підприємства витрачають в середньому 28,91 мільйона доларів на хмарні сервіси. Така сума є більшою приблизно на 6,5 мільйона, ніж очікувалось. “Постачальникам послуг більше необхідно пропонувати високобезпечні, масштабовані рішення та гнучкі варіанти розгортання, які забезпечать компаніям оптимальне співвідношення ціни та якості ресурсів, що дозволить їм зосередитися на покращенні та прискоренні розвитку свого бізнесу”, - говорить Рахул Прадхан, віце-президент із продуктів і стратегії Couchbase [3]. Варто зазначити, що 53% компаній переводять свої витрати на хмарні обчислення з капітальних витрат на операційні витрати, в середньому ця сума дорівнює 25% капітальних витрат. Це означає, що чверть їхніх початкових інвестицій у хмарні сервіси тепер буде оплачуватися як поточні витрати. Крім того, до 2026 року компанії очікують, що 31% їхніх загальних витрат на IT буде припадати на публічну хмару.

Якщо порівняти суми витрат, неозброєним оком можна побачити стрімкий ріст витрат на всі види сервісів, які забезпечуються хмарними провайдерами. В таблиці наведені зміни за кожен рік і можна перебачити, що цей ріст буде кожен рік буде тільки збільшуватися (див. рис. 1.1).

| | 2021 | 2022 | 2023 |
|---|---|---|---|
| Cloud Business Process Services (BPaaS) | 54,952 | 60,127 | 65,145 |
| Cloud Application Infrastructure Services (PaaS) | 89,910 | 110,677 | 136,408 |
| Cloud Application Services (SaaS) | 146,326 | 167,107 | 195,208 |
| Cloud Management and Security Services | 28,489 | 34,143 | 41,675 |
| Cloud System Infrastructure Services (IaaS) | 90,894 | 115,740 | 150,254 |
| Desktop-as-a-Service (DaaS) | 2,059 | 2,539 | 3,104 |
| **Total Market** | **412,632** | **490,333** | **591,794** |

Рисунок 1.1 – Ріст витрат на хмарні обчислення (за даними [4])

На даний час найбільш розповсюдженими хмарними гігантами серед клієнтів вважаються Amazon Web Services, Microsoft Azure, Google Cloud Platform.

AWS пропонує найширший спектр хмарних послуг, включаючи обчислення, сховище, мережі, бази даних, аналітику, машинне навчання та штучний інтелект. AWS відомий своїми інноваціями, масштабованістю та надійністю, що робить його популярним вибором для компаній будь-якого розміру.

Azure пропонує широкий спектр послуг, подібних до AWS, і особливо сильний у гібридних хмарних рішеннях, що робить його привабливим вибором для компаній із наявними інвестиціями Microsoft.

А Google Cloud відомий своїми інноваційними технологіями, такими як машинне навчання та штучний інтелект. GCP стає все більш популярним для компаній, які шукають гнучкі та масштабовані хмарні рішення.

Крім того, існує багато інших провайдерів, які можуть запропонувати свої переваги в надані хмарних послуг, наприклад, Oracle Cloud, Digital Ocean, IBM cloud тощо. Нові провайдери на будь який смак з’являються дуже часто, тому клієнти по всьому світу можуть обрати свого відповідно до потреб їх бізнесу.

## 1.3 Принцип роботи хмари та надання хмарних послуг

Варто почати з загального визначення, що сам по собі означає термін «хмара». Хмарні обчислення – модель забезпечення повсюдного та зручного доступу на вимогу через мережу до спільного пулу обчислювальних ресурсів, що підлягають налаштуванню (наприклад, до комунікаційних мереж, серверів, засобів збереження даних, прикладних програм та сервісів), і які можуть бути оперативно надані та звільнені з мінімальними управлінськими затратами та зверненнями до провайдера [5].

Є дві важливі переваги, пов’язані з використанням компаніями хмар - гнучкість і зменшення витрат. Гнучкість – це здатність організації реагувати на раптові проблемі на ринку та галузі. Вона дозволяє компанії швидко використовувати нові можливості та збільшувати прибутки. Нездатність швидко реагувати на зміни в ІТ сфері та адаптуватися до постійних змін на ринку призведе до втрати конкурентної переваги. І хоча гнучкість може призвести до збільшення початкових витрат, хмарні обчислення допомагають знизити вартість внесення потенційних змін. Як ми розглянемо пізніше, основні характеристики хмарних сервісів дозволяють швидко впроваджувати нові рішення та масштабувати їх за потребою. При цьому користувачі сплачуючують лише за фактичне використання послуг.

Відповідно до національного інституту стандартів і технологій (NIST), хмарний сервіс повинен мати п'ять характеристик, щоб вважатися справжнім хмарним рішенням:

- загальний пул ресурсів;
- швидку еластичність;
- самообслуговування за потребою;
- широкий мережевий доступ;
- вимірюване обслуговування.

Використання загального пулу ресурсів дає змогу кільком організаціям-споживачам спільно використовувати певні ресурси хмарного постачальника. Це

явище відомо як “multi-tenancy” модель. Завдяки їй споживач може повною мірою використовувати всі ресурси без витрат на придбання та обслуговування.

Об'єднання ресурсів також забезпечує швидку еластичність. Наприклад, якщо у споживача відбувається велика подія, під час якої навантаження може різко зрости в декілька разів, ресурси можна буде швидко збільшити на певний проміжок часу, коли проходить подія. Коли додаткові ресурси вже більше не потрібні, їх потрібно видалити. Швидка еластичність дозволяє додавати або видаляти ресурси протягом декількох хвилин, тоді як такий процес з традиційним підходом з власними дата центрами може зайняти дні або тижні. Більше того, розширення інфраструктури власного дата центру для таких разових подій може призвести до того, що більша частина ресурсів буде просто простоювати, коли вони вже перестануть бути необхідні.

Залежно від принципу використання хмарних обчислень, зміни у використанні ресурсів можуть відбуватися автоматично на основі “вимірюваного використання”. Оплата лише за те, що ви використовуєте, потребує точних вимірів тої кількості ресурсів, які фактично використовує споживач. У нашому попередньому прикладі споживач платитиме за збільшення ресурсів лише протягом необхідного часу під час проведення тимчасової події зі збільшенням навантаження. Знову ж таки, це знижує витрати на загальні витрати датацентрів, що пов'язані із впровадженням тимчасових резервних систем. Якщо споживач укладає угоду з хмарним постачальниокм щодо автоматичного коригування виділених ресурсів, споживачу можуть надсилатися сповіщення, коли ресурси додаються або витрати ростуть.

Самообслуговування за потребою тісно пов'язане зі швидкою еластичністю. Споживач може швидко запускати необхідні ресурси або вивести їх з експлуатації самостійно в будь-який момент часу, як тільки виникає необхідність. І знову ж таки для порівняння можна привести дні або тижні, які можуть знадобитися для придбання, отримання, налаштування та впровадження ресурсів в дата центри, які підтримуються користувачем самостійно. Крім того, широкий мережевий доступ

дозволяє споживачам, клієнтам, співробітникам, партнерам і постачальникам отримувати доступ до потрібних ресурсів будь-де та будь-коли.

Додатковою перевагою є зберігання та відновлення даних. Хмарні обчислення забезпечують усі необхідні інструменти для збереження даних, а також план резервного копіювання. У хмарі дані ніколи не будуть втрачені завдяки резервуванню [6].

Тепер розглянемо моделі хмарного розгортання. Споживачі можуть використовувати одну з моделей розгортання - приватна, публічна або гібридна (див. рис. 1.2). Бажана модель в основному залежить від необхідної безпеки даних та послуг. Вибір між ними може вплинути на ефективність, безпеку та витрати на IT-інфраструктуру. Це означає, що дуже важливо ретельно оцінювати всі можливості перед прийняттям такого рішення. Кожна з моделей має свої переваги і недоліки, і важливо підібрати ту, яка найкраще відповідатиме цілям та стратегії організації [7].

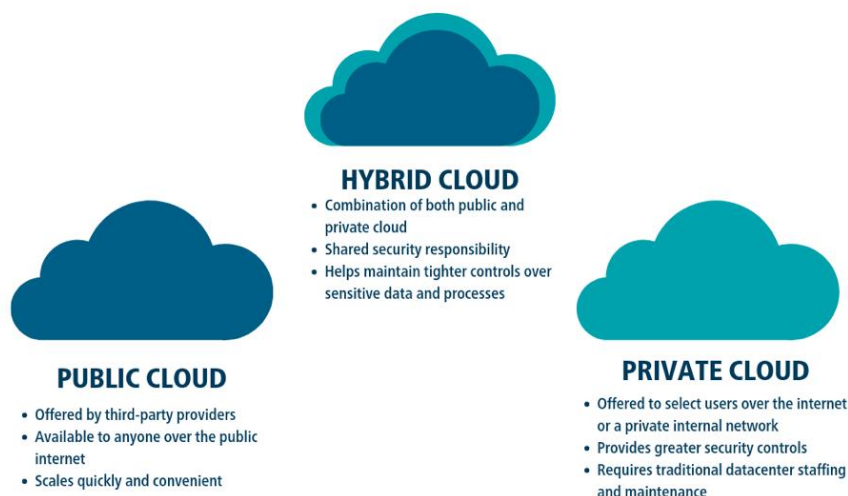



Рисунок 1.2 – Типи постачання хмар (за даними [8])

Приватна хмара використовується виключно однією організацією. Доступ суворо контролюється, а фізична інфраструктура розміщується в окремому дата центрі хмарного постачальника або інших готових дата центрах. Такі реалізації моделі хмарних обчислень є досить актуальними для компаній, які вже мають

певну локальну інфраструктуру та повинні належним чином управляти ризиками, що пов'язані з дуже конфіденційною інформацію.

Крім того, приватні хмари забезпечують більшу гнучкість, оскільки ресурси не надаються іншим споживачам. Відсутність використання загального пулу ресурсів на відміну від публічних хмар також збільшує витрати.

В той же час публічна хмара - це модель, де обчислювальні послуги та інфраструктура керуються стороннім постачальником і надаються спільно багатьом організаціям. Іншими словами, загальнодоступна хмара - це пул обчислювальних, інформаційних та мережевих ресурсів, розташованих і повністю керованих хмарними постачальниками. Ресурси спільно використовуються будь-якою кількістю споживачів за вже відомими нам підходом – “multi-tenancy”.

Основними перевагами публічної хмари є висока масштабованість, гнучкість і доступність. Користувачі можуть швидко і легко збільшувати або зменшувати обсяги послуг за потреби, що забезпечує велику гнучкість у використанні ресурсів. Використання такого типу значно зменшує витрати, оскільки користувачі платять тільки за використані ресурси за моделлю "оплата за використання".

Однак публічна хмара також має певні недоліки. В умовах жорсткої конкуренції, найбільше компанії бояться витоків даних з мережі хмарного провайдера внаслідок перехоплення інформації, втрати контролю над даними і додатками, неможливості знищення даних, дій інсайдера на стороні провайдера або інших користувачів хмари [9]. Крім того, існує пряма залежність від стабільності та надійності обраного провайдера, що впливає на можливі ризики, пов'язані зі змінами в політиках або ціноутворенні.

Окремий споживач може мати потребу як у приватній, так і в публічній хмарі одночасно. Реалізація такої моделі є найскладнішою для створення та управління. Найбільш правильне визначення їй дає Національний інститути технологічних стандартів: “Інфраструктура гібридної хмари – це сукупність двох або більше окремих хмарних інфраструктур (приватних та публічних), які

залишаються окремими сутностями, але пов'язані разом стандартизованою технологією, яка підтримує переносимість даних та додатків” [10].

## 1.4 Типи надання хмарних послуг

У хмарних обчисленнях існує три основні типи надання послуг:

– інфраструктура як послуга (IaaS);
– платформа як послуга (PaaS);
– програмне забезпечення як послуга (SaaS).

Кожна з них забезпечує різний рівень абстракції та управління базовою IT-інфраструктурою.

Інфраструктура як послуга (IaaS) - це модель хмарних обчислень, яка надає віртуалізовані обчислювальні ресурси через Інтернет. Користувачі орендують IT-інфраструктуру на умовах оплати використаних ресурсів, включаючи віртуальні машини, сховища та мережі. При цьому необхідність керувати цим обладнанням відпадає [11].

Найвідоміші провайдери IaaS включають Amazon Web Services (AWS) EC2, віртуальні машини Microsoft Azure та Google Cloud Compute Engine.

Таким чином, IaaS пропонує віртуалізовані обчислювальні ресурси, що дозволяє користувачам масштабувати ресурси за потреби без управління фізичною інфраструктурою. Використання інфраструктури як сервісу дозволяє компаніям скоротити свої витрати, підвищити гнучкість і зосередити увагу на інноваціях, а не на управлінні апаратним забезпеченням.

Платформа як послуга (PaaS) - це модель хмарних обчислень, яка забезпечує комплексне середовище для розробки, запуску та управління додатками. При цьому такий процес не вимагає від користувача роботи з базовою інфраструктурою. Вона включає в себе ряд інструментів, таких як інтегровані середовища розробки (IDE), засоби контролю версій і фреймворки для тестування, які призначені для оптимізації процесу розробки. PaaS також пропонує послуги проміжного програмного забезпечення, автоматичне

масштабування та системи управління базами даних, що дозволяє розробникам зосередитися на імплементації та тестуванні своїх додатків та сервісів [12].

Програмне забезпечення як послуга (SaaS) - це модель хмарних обчислень, яка надає програмні додатки через Інтернет на основі підписки. Користувачі отримують доступ до цих програм через веб-браузер. Постачальники SaaS відповідають за управління всім, від інфраструктури до логіки роботи програми, забезпечуючи користувачам доступ до найновіших функцій та оновлень.

Додатки SaaS охоплюють широкий спектр функціональних можливостей, включаючи офісні пакети, управління взаємовідносинами з клієнтами, планування ресурсів підприємства та інструменти для спільної роботи.

На рисунку 1.3 детально зображено, які конкретно компоненти включає в себе певний вид надання хмарних послуг.

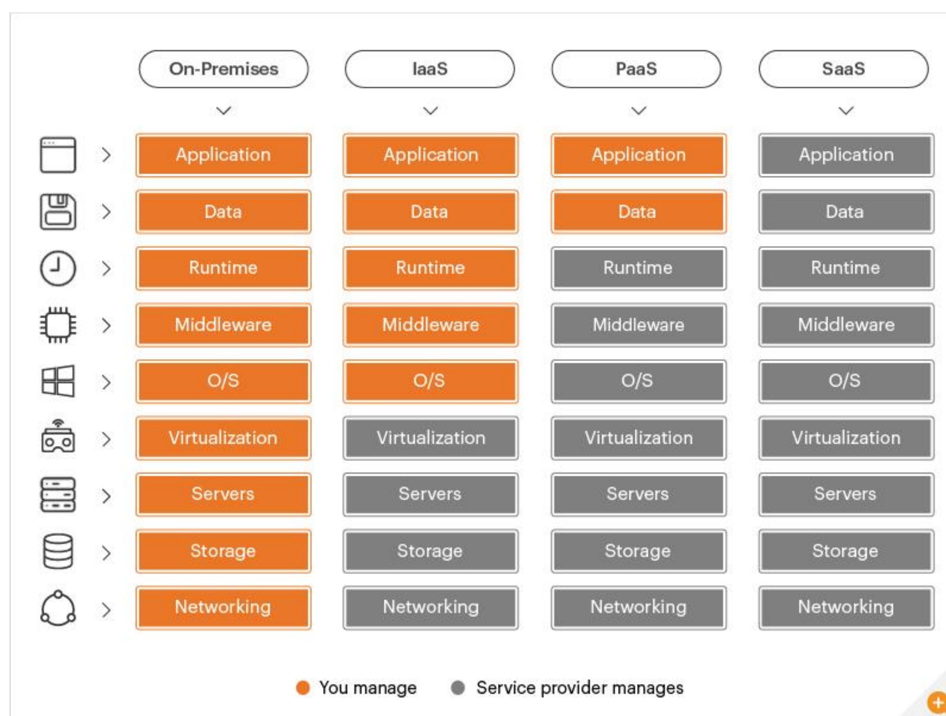



Рисунок 1.3 – Рівні відповідальності видів хмарних послуг (за даними[13])

On-premises - це будь-які IT-сервіси або інфраструктура, які розміщуються, управляються та експлуатуються у фізичних приміщеннях або центрах обробки даних організації, а не передаються на аутсорсинг сторонньому постачальнику хмарних послуг. В такому випадку всі частини системи компанія повинна

створювати самостійно. Це означає, що потрібно придбати все - починаючи від фізичних складових, закінчуючи створенням віртуальної платформи, розробкою та розгортанням додатку. Як видно на зображенні, підтримка фізичних складових означає, що компанія повинна придбати фізичні машини, що будуть слугувати серверами, роутери та кабелі для створення мережі, машини для зберігання даних тощо. Витрати на придбання та обслуговування можуть бути непідйомними і нераціональними для більшості компаній. Саме через це користувачі почали мігрувати в хмарні середовища та арендувати потрібний вид ресурсів.

У ситуації з рівнем IaaS помітно, що його повноваження перепадають тільки на фізичні складові, що є "наріжним каменем" системи. Користувачі, які обирають цю модель отримують доступ до фізичних ресурсів, які знаходяться в дата центрах хмарного провайдера, тому така аренда коштує в сотні або навіть тисячі раз дешевше. Розробка, тестування додатку, а також створення всіх більш абстрактних та віртуальних компонентів, наприклад, кластерів для розгортання, засобів безпеки та моніторингу, бере на себе сам користувач.

Модель PaaS є дуже схожою на IaaS, тому що користувачеві надаються базові ресурси та середовище для розробки та тестування сервісу. Але, як вже зазначалося вище, на відміну від IaaS, PaaS гарантує замовнику не тільки базові обчислювальні пристрої, а й віртуальне середовище. Працюючи в ньому розробникам не потрібно задумуватися про те як і куди деплоїти додаток, налаштовувати слідкування за станом системи, безпекою та як реалізувати масштабування сервісу тощо. Незважаючи на це, користувачу все ще потрібно думати про сам процес розробки сервісу та які дані будуть в ньому оброблюватися.

SaaS є найабстрактнішою моделлю, тому що користувачу надається кінцевий результат роботи сервісу та інтерфейс для його використання. Підписуючись на такий тип, ви не повинні думати про процес розробки та підтримки цього додатку. Всі низькорівневі процеси перебігають на задньому плані і виконуються тим, хто надає вам цю послугу. Такий тип моделі передбачає

покриття усіх складнощів від початку до кінця створення продукту, а замовник отримує готовий кінцевий результат для використання.

Опираючись на вищезазначені факти, стає зрозуміло, що низькорівнева інфраструктура для сервісу є чи не найголовнішою складовою частиною в розробці систем, тому якість реалізація процесу її створення та управління є критично важливим критерієм.

# 2 ПОСТАНОВКА ЗАДАЧІ

## 2.1 З чого складається хмарна інфраструктура

Хмарна інфраструктура – це сукупність апаратних та програмних ресурсів, що становлять хмару. Постачальники хмарних послуг обслуговують глобальні центри обробки даних із тисячами компонентів IT-інфраструктури, таких як сервери, фізичні пристрої зберігання даних та мережеве обладнання. Вони налаштовують фізичні устрою з використанням всіх типів конфігурацій операційної системи. Вони також встановлюють інші типи програмного забезпечення, необхідного для запуску програми. Ваша організація може орендувати хмарну інфраструктуру з оплатою за фактом використання, що дозволяє значно заощадити придбання та обслуговування окремих компонентів [14].

Звичайно, як і будь-яку інфраструктуру або архітектуру, її можна поділити на певні основні компоненти, з яких вона складається (див. рис 2.1).

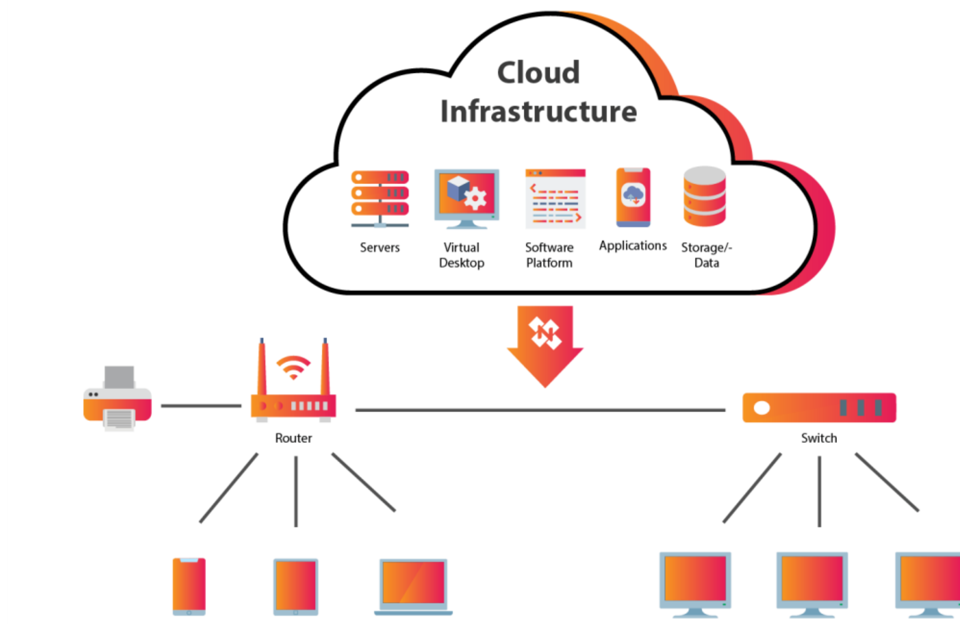



Рисунок 2.1 – Схема компонентів хмари (за даними [15])

По-перше, базовою частиною, яка формує інфраструктуру є сервери і вони є базовим робочим блоком в хмарі. Хмарна інфраструктура ґрунтується на обширній мережі потужних серверів, які слугують базовим фундаментом для всіх сервісів доступних з хмари. Такі сервери можуть бути тільки фізичними пристроями (такими як ви маєте в себе вдома, тільки набагато більшими та потужнішими). Вони спеціально розроблюються для цілі обробки та збереження величезної кількості даних.

Але найчастіше за все хмарні постачальники використовують технологію віртуалізації. Це дозволяє їм створювати одночасно декілька віртуальних серверів на одній фізичній машині, яку ми описали в минулому абзаці. Це виглядає приблизно як будинок, що поділений на квартири, кожна з яких живе своїм незалежним життям. Кожен віртуальний сервер має власну операційну систему, окремі програми, спеціально виділені фізичні ресурси з пристроя, які не можуть бути доступними іншим віртуальним процесам. Завдяки віртуалізації, процес створення сервісів у хмарах стає більш економічно вигідним, безпечним, краще масштабованим. Це також забезпечується тим, що такий віртуальний процес можна легко створити або видалити.

Підсумовуючи, сервери діють як “робітники”, що забезпечують обчислювальну потужність та тимчасове сховище, необхідні для запуску різноманітних програм і сервісів. В інфраструктурі AWS, наприклад, такими робітниками є EC2 екземпляри.

Крім серверів, важливим компонентом в хмарі є мережа. Хмарна мережа це складний ланцюг мережевих з'єднань, що дозволяє різноманітним незалежним компонентам хмарної інфраструктури взаємодіяти між собою. Мережа виступає в ролі чогось схожого на “нервову систему”, що постійно пересилає певну інформацію між поєднаними серверами, сховищами або навіть географічно віддаленими точками.

Незважаючи на те, що в більшості випадків мережа включає в себе використання звичайних фізичних каналів, комутаторів, маршрутизаторів тощо, основою саме хмарною мережі також стала віртуалізація. Користувачі можуть

створювати ізольовані virtual private clouds, subnets, virtual gateways тощо, налаштовуючи зв'язки так, як вони хочуть. Це привносить деякі плюси на відміну від тільки фізичної мережі. Прикладом може слугувати масштабованість, що означає легку акомодацію такої мережі до збільшення навантажень або змін в архітектурі. Крім того, такий підхід може забезпечувати безпеку, використовуючи фаєрволи або інші служби для фільтрації трафіку.

Також надзвичайно важливим компонентом хмарної інфраструктури є сховище. Хмарне сховище - це модель хмарних обчислень, яка дає змогу зберігати дані та файли в Інтернеті через постачальника хмарних обчислень, до якого ви отримуєте доступ через інтернет або приватне підключення до мережі [16]. Постачальник безпечно зберігає, керує та обслуговує сервери зберігання, інфраструктуру та мережу, щоб забезпечити вам доступ до даних, коли вони вам потрібні, у практично необмеженому масштабі та з еластичною ємністю. Хмарне сховище усуває необхідність купувати власну інфраструктуру зберігання даних і керувати нею, надаючи вам гнучкість, масштабованість і довговічність із доступом до даних у будь-який час і будь-де.

Прикладом такого сховища в AWS є Elastic Block Storage. Його можнва розглядати як високопродуктивний масштабований віртуальний жорсткий диск, який можна під'єднати до вже вищезгаданих запущених екземплярів EC2. На відміну від тимчасового сховища, яке постачається з екземплярами EC2, томи EBS є постійними, що означає, що ваші дані залишаються надійно збереженими, навіть якщо ви вимкнете віртуальні машини. Це робить їх ідеальними для зберігання критично важливих програм, баз даних і будь-яких даних, які мають бути доступними після перезапуску екземплярів.

Окрім вищеописаних компонентів сучасної хмарної інфраструктури, існує велика кількість інших не менш важливих сервісів, які стають невід'ємною частиною хмарних провайдерів і їх список постійно поповнюється. На прикладі AWS приведемо декілька прикладів.

Під час створення хмарної інфраструктури для сучасних додатків часто виникає потреба в створенні кластерів оркестрації. В AWS це Elastic Kubernetes

Service – Kubernetes кластер, який автоматично управляється та розгортається AWS провайдером. Вам потрібно тільки визначити його характеристики.

Також зараз набирає популярності “message brokers”. Це один із шаблонів обміну інформації між певними сервісами, який може замінити REST протокол або протокол віддаленого виклику процедур(RPC). Під час створення вашої інфраструктури в певних хмарах ви також можете додати такий компонент до вашої системи. З точки зору AWS таким компонентом є AWS SQS. В додаток до зазначених можливих компонентів, провайдери мають десятки інших, які користувач може додавати при створенні власної інфраструктури для свого хмарного додатку.

## 2.2 Проблеми при створені хмарної інфраструктури

Кожна IT організація при розробці своїх проектів намагається максимально пришвидшити процес доставки своїх систем кінцевим користувачам. В сучасному світі кількість та масштаб веб сервісів збільшується з кожним днем, тому збільшується кількість користувачів і в результаті кількість даних для обробки. Це змушує компанії по розробці програмного забезпечення створювати та розширювати свої середовища, де розгортання додатків буде швидким і безперебійним.

На жаль, звичайні методи ручного забезпечення та налаштування інфраструктури не дають гарного результату в наш час. Мало того, що вони не є ефективними з точки зору витрат часу, але вони також більш схильні до помилок. Такі помилки можуть призвести до критичних збоїв у роботі додатків, що може мати вкрай негативні наслідки для користувачів та розробників. Саме тут автоматизація інфраструктури стає рятівником. Інтеграція автоматизації в цьому процесі дозволяє IT організаціям перейти від ручних та трудомістких кроків до передачі відповідальності за це іншим автоматичним інструментам та механізмам.

Створення інфраструктури вручну було громіздким процесом, який вимагав значних людських зусиль. Працівники ретельно налаштовували кожен сервер, кожну мережу та сховище даних окремо, переміщуючись по складних меню та

екранах, що забирало багато часу і нагадувало збірку комп'ютера вручну. Такий підхід мав кілька обмежень.

По-перше, налаштування нового сервера передбачало довгу серію кліків і конфігурацій для кожної машини, що робило його повільним і неефективним. По-друге, ручний характер процесу був дуже схильний до помилок. Одна помилка в налаштуванні або забутий крок могли призвести до падіння сервера і простою додатків. Усунення цих помилок часто включало в себе шалений пошук, щоб визначити та виправити джерело проблеми.

В решті решт масштабування інфраструктури стало серйозною проблемою, коли додавання нових компонентів на хмару означало повторення кожного кроку десятки або сотні разів (див. рис. 2.2). Такий підхід став так названим “bottleneck”, що призводив до значного уповільнювання процесу розробки та роботи веб-систем.

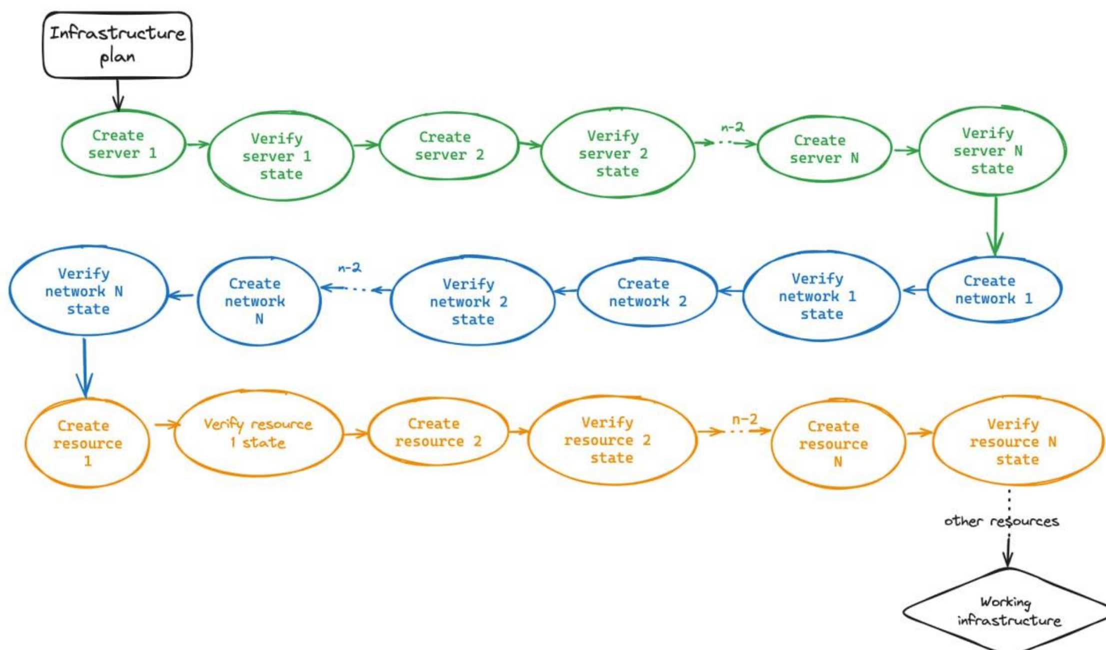



Рисунок 2.2 – Схема процесу створення інфраструктури(виконано самостійно)

Незважаючи на це, навіть у наш час досі бувають випадки, коли ручне створення компонентів хмари є більш доцільним і правильним варіантом.

Наприклад, таким випадком може бути ситуація, коли налаштування інфраструктури передбачає складні конфігурації або унікальні вимоги, які неможливо врахувати в автоматизованих скриптах або шаблонах.

Крім того, у навчальних цілях або під час експериментів з новими технологіями ручне налаштування може дозволити краще розуміти процес налаштування певного компонента на найнижчому рівні. Автоматизація ж додає певний рівень абстракції і може опускати певні характеристики налаштувань, що може призвести до використання компонента хмари в неповному обсязі.

Ще одною ситуацією, коли ручне налаштування може бути більш актуальним варіантом – робота зі старими системами. Такі системи можуть не підтримувати інструменти автоматизації або не надають API, який можна використати в таких інструментах для виконання запитів до хмари. Тому для них ручне налаштування стає єдиним варіантом.

Останнім, але не менш важливим випадком для мануального створення та управління є невеликі проекти або разові завдання. Під час роботи над такими завданнями часові або навіть грошові витрати можуть перевищувати користь від автоматизації. Тоді ручне налаштування може бути швидшим і ефективнішим.

Тим не менш, в реальних бізнес-кейсах ручний спосіб розглядається більш як виключення, ніж правило.

## 2.3 Задача дослідження

Проаналізувавши головні проблеми, які існують в сфері управління інфраструктурою хмарних застосунків було визначено декілька підзадач, що мають бути розглянуті в ході подальшого дослідження:

- розглянути та порівняти існуючі методи автоматизації створення та управління інфраструктурою;
- визначити найперспективніший метод;
- визначити декілька найкращих інструментів, що входять до метода, практично та теоретично порівняти;
- на практиці детально розглянути принцип роботи переможця.

# 3 ПОРІВНЯННЯ МЕТОДІВ ТА ПРИНЦИПІВ СТВОРЕННЯ ХМАРНОЇ ІНФРАСТРУКТУРИ

## 3.1 Infrastructure as a code

На заміну ручному створенню інфраструктури прийшов нових підхід – Infrastructure as a Code(IaC). Infrastructure as a Code це підхід для управління та опису інфраструктури через конфігураційні файли, а не через ручне редагування конфігурацій. IaC поводиться зі скриптами та конфігураційними файлами так само, як розробники поводяться з кодовою базою проекту [17].

Інфраструктурний код має проходити коміти, CI/CD, merge requests, code review. IaC використовує високорівневу мову опису коду для автоматизації надання IT-інфраструктури. Ця автоматизація позбавляє розробників необхідності вручну виділяти та керувати операційними системами, серверами, підключеннями до сховищ, баз даних та інших елементів інфраструктури щоразу, коли потрібно написати, протестувати або розгорнути програмний додаток (див. рис. 3.1).

До переваг IaC можна віднести:

- Управління версіями. Опис інфраструктури як і звичайний код може підтримуватися системами контроля версій. Тому інженери зможуть побачити хто, коли робив зміни, який зараз очікуваний стан системи та відкочувати їх до певної версії.
- Можливість повторного використання. Як і звичайний програмний код IaC надає можливість використовувати базові модулі, які можна використовувати багато разів в різних місцях, а не просто робити одні й ті ж зміни кожен раз.
- Узгодженність. Завдяки тому, що опис інфраструктури як і звичайна кодова база повинна проходити перевірку та отримувати схвалення, кількість потенційних помилок зменшується, тому збільшується надійність системи.

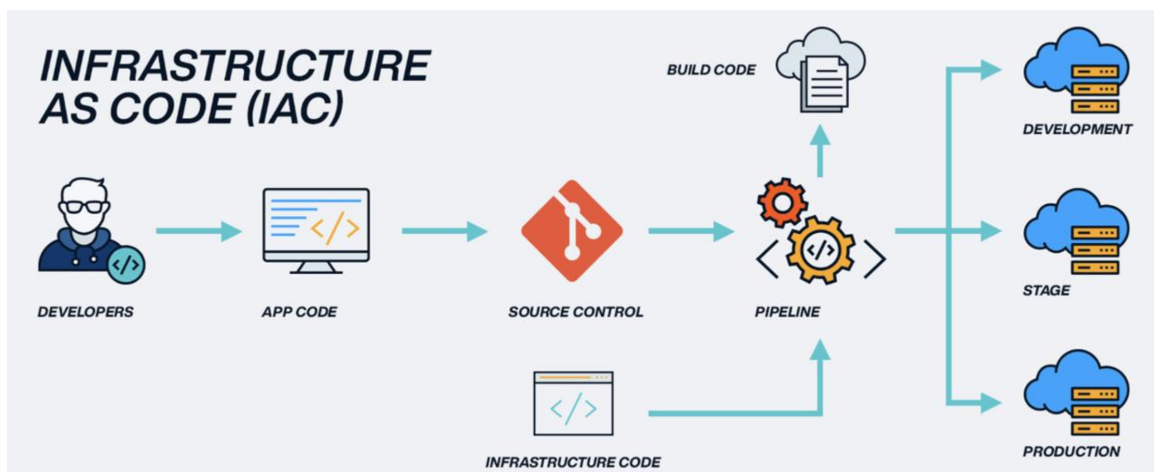



Рисунок 3.1 – Схема роботи IaC принципу (за даними [18])

3.2 Декларативні та імперативні підходи

Програмування та управління системами охоплюють два різні підходи, а саме імперативну та декларативну парадигми, кожна з яких характеризується унікальними принципами та методологіями (див. рис. 3.2).

Рисунок 3.2 – Переваги і недоліки підходів (за даними [19])

Основний акцент імперативної парадигми полягає в точному окресленні серії операцій, які комп’ютер повинен виконати, щоб досягти бажаного результату. Ця методологія характеризується поетапною процедурою, у якій

розробник чітко визначає конкретну послідовність дій та які саме дії повинні виконуватися. В цій парадігмі велику роль відіграють різні конструкції на кшталт цикли, умови та інші. Саме вони дозволяють розробнику, що вирішує використовувати такий підхід, точно контролювати дії програми та передбачати всі потенціальні ситуації.

В цей час декларативна парадигма робить акцент саме на формулюванні бажаного результату та його визначенні, а не на конкретних засобах його досягнення. Ця методологія передбачає абстрагування від складних процедурних інструкцій, що дозволяє розробникам визначати бажаний кінцевий стан або мету, а базова система самостійно визначає необхідну послідовності дій для досягнення цього стану.

Коли справа доходить до побудови інфраструктури, імперативний підхід передбачає чітке розмежування кожної необхідної дії для встановлення та налаштування ресурсів. У разі використання імперативних інструментів користувач створює сценарії, які описують необхідні кроки та команди для створення, видалення, оновлення кожного ресурсу. Це дозволяє проводити ретельне налаштування, що є важливим у складних середовищах, які вимагають точної унікальної послідовності дій. Однак виникає недолік, коли ці сценарії стають довшими та ускладнюються, що робить контроль та підтримку неможливим у міру розширення інфраструктури.

З іншого боку, коли йдеться про створення інфраструктури, може використовуватися декларативний підхід, який передбачає чітке формулювання бажаного кінцевого результату інфраструктури та надання системі можливості виконувати конкретні кроки самостійно. Наприклад, замість того, щоб вказувати всі кроки для створення певного ресурсу, наприклад, сервера, можна просто вказати, які саме характеристики та властивості він повинен мати. В цей час декларативна реалізація сама повинна розуміти, які імперативні кроки потрібно зробити, щоб в кінці отримати заданий стан. Цей підхід дозволяє більше абстрагуватися, не фокусуватися на низькорівневих деталях, що спрощує та прискорює розгортання. Також, з точки зору інфраструктури, декларативний

підхід гарантує певну узгодженість, тому що система автоматично обробляє зміни та робить так, щоб інфраструктура завжди відповідала тому стану, який від неї чекають.

### 3.3 Скрипти та інтерфейс командного рядка

Так як створення інфраструктури почало вимагати швидкої автоматизації, як і будь які IT процеси, першим етапом стало написання скриптів. Як і в будь-якому процесі автоматизації, невеликі і прості скрипти стали першим інструментом через свою простоту та швидкість написання.

Інтерфейс командного рядка (CLI) в цей період відігравав важливу роль в автоматизації процесів створення хмарної інфраструктури. Ці інтерфейси надають розробникам і системним адміністраторам потужні інструменти для програмної взаємодії з хмарними ресурсами, дозволяючи автоматизувати різні завдання і спростити управління хмарною інфраструктурою.

Основна мета CLI для автоматизації хмарних обчислень - спростити виконання повторюваних і складних операцій. Вони дають змогу запускати команди безпосередньо з командного рядка, що дає змогу швидше розгортати, налаштовувати та керувати хмарними ресурсами без будь-якого ручного втручання. Наприклад, створення віртуальних машин за допомогою сценарію порівняно з ручним введенням одних і тих самих команд щоразу може заощадити значну кількість часу для адміністратора, якому потрібно створити кілька машин. Дії, що виконуються в CLI, також можуть бути включені в інші інструменти автоматизації або сценарії, які ви використовуєте на даний момент.

Ці сценарії можуть автоматизувати весь процес розгортання - від створення інфраструктури до розгортання застосунків і масштабування ресурсів або навіть виконання подальших дій з обслуговування.

Крім того, розробники можуть використовувати скрипти написані на Bash, що дасть змогу краще контролювати бажані операції та додавати додаткову логіку. Приклад невеликого bash скрипта для створення мінімально простої AWS інфраструктури наведений в наступному коді.

```
#!/bin/bash
REGION="us-west-2"

INSTANCE_ID=$(aws ec2 run-instances --image-id "ami-0abcdef1234567890"  --count 1 --instance-type "t2.micro" --key-name $KEY_NAME --subnet-id $SUBNET_ID --region $REGION --query 'Instances[0].InstanceId' --output text)
echo "EC2 Instance launched with ID: $INSTANCE_ID"
aws ec2 wait instance-running --instance-ids $INSTANCE_ID --region $REGION
echo "EC2 Instance is running"
PUBLIC_IP=$(aws ec2 describe-instances --instance-ids $INSTANCE_ID --region $REGION --query 'Reservations[0].Instances[0].PublicIpAddress' --output text)
echo "EC2 Instance Public IP: $PUBLIC_IP"
echo "Infrastructure setup complete."
```

Як бачимо, цей код виконує свою основну функцію – підняти інфраструктуру з нуля. Але при цьому ж очевидно, що такий код не є гарно читабельним. А при ситуації, якщо потреба збільшиться, складно уявити, в що перетвориться такий файл. Окрім того, при необхідності виконання змін в існуючій інфраструктурі, створеній за допомогою цих інструкцій, цей код буде повністю непридатним. Для цього потрібно буде робити велику кількість змін, додавати умовні оператори тощо. Це призведе не тільки до ще більшої нечитабельності, але й до потенційно критичних помилок.

Як не складно помітити, CLI скрипт визначає чітку послідовність дій, яку розробник хоче застосувати до хмарного середовища. Це означає, що перед нами типовий представник імперативної методології.

## 3.4 Source Development Kits

Використання Source Development Kit для створення хмарної інфраструктури дозволяє розробникам автоматизувати та програмно керувати хмарними ресурсами. SDK являть собою певні бібліотеки на різних популярних мовах програмування, які хмарні постачальники надають користувачам. Розробники можуть писати власний код для задоволення специфічних вимог, які можуть бути неможливими за допомогою інструментів командного рядка або консолей управління.

Одна з головних переваг, що є характерною для SDK є можливість писати більш складні конструкції. Розробники можуть писати код, щоб виконати специфічні вимоги, що можуть бути нереальними для реалізації за допомогою CLI або веб-консолі. Рівень деталізації, що надається SDK, може бути дуже корисним компаніям з незвичними потребами або складною інфраструктурою. Крім того, SDK надають схожий інтерфейс для різних мов програмування. Така характеристика є дуже важливою, щоб підтримувати уніфікованость в проектах з багатьма різноманітними мовами програмування.

Однак використання SDK для управління хмарною інфраструктурою має й певні недоліки. Однією з головних проблем є початкова крива навчання. Розробники повинні ознайомитися з конкретним SDK, його API та найефективнішими методами його використання. Цей процес навчання може зайняти багато часу і вимагати значних зусиль, особливо для новачків у хмарних сервісах або програмуванні.

Іншим потенційним недоліком є залежність від певного SDK. Сильна залежність від певного SDK може створити проблеми, якщо він зміниться або застаріє. Ця залежність також може ускладнити міграцію до іншого хмарного провайдера або платформи, оскільки SDK є специфічним для певного провайдера і код потрібно буде повністю переписувати.

У деяких випадках проблемою може бути і продуктивність. Хоча SDK зазвичай оптимізовані, додаткові рівні абстракції іноді можуть впливати на продуктивність. Це особливо помітно в сценаріях, що передбачають велику кількість ресурсів для розгортання та складність схеми інфраструктури. Крім того, налагодження та усунення помилок та багів в автоматизації на основі SDK може бути складнішим, ніж використання веб-інтерфейсу хмарного провайдера. Пошук помилок може бути дуже складним процесом. Ця складність може збільшити час і зусилля, необхідні для підтримки та оновлення коду інфраструктури.

Схожою проблемою методів з використанням SDK та CLI є відсутність підтримки та зберігання стану системи. Тобто, при виконанні такої програми ні

вона, ні ми не знаємо про ресурси, які є в зовнішній системі. Через це, наприклад, якщо ресурси, що описані в коді, вже існують, програма впаде. Щоб це виправити, потрібно вручну покривати кожну можливість стану для кожного ресурсу. Це призводить до потенційних критичних помилок та неможливості подальшої підтримки коду.

Наведемо приклад коду на Python, що створює певну систему в середовищі AWS.

```
import boto3
session = boto3.Session(region_name='us-west-2')
ec2 = session.resource('ec2')
instances = ec2.create_instances(
    ImageId='ami-0abcdef1234567890',
    InstanceType='t2.micro',
    KeyName='your-key-pair',  # Replace with your key pair name
    NetworkInterfaces=[{
        'SubnetId': subnet.id,
        'DeviceIndex': 0,
        'AssociatePublicIpAddress': True,
        'Groups': [security_group.id]
    }]
)
instance = instances[0]
instance.wait_until_running()
print(f"EC2 Instance launched")
```

Порівнявши цей код з кодом скрипту на Bash помітно, що такі інструкції є більш читабельні для розробників та інтуїтивно зрозумілі. Також варто помітити ще декілька переваг над CLI. По-перше, так як це код на мові програмування, він легко тестується оскільки будь-яка мова програмування підтримує просте написання unit та integration тестів. По-друге, IDE зараз мають дуже широкі можливості, тому при написанні такого коду вони будуть дуже допомагати, вказувати на потенційні помилки і так далі. Крім того, на відміну від CLI, мови програмування мають зручні та читабельні синтаксичні конструкції по типу умов та циклів. Це допоможе більш гнучко описувати потреби, але все ще недостатньо.

## 3.5 Cloud Development Kits

AWS Cloud Development Kit дозволяє розробникам описувати бажану хмарну інфраструктуру за допомогою популярних мов програмування, таких як TypeScript, Python, Java тощо, або декларативних форматів серіалізації – yaml, json та інші. CDK полегшує створення, розгортання та управління хмарною інфраструктурою у більш простий спосіб. CDK є представником декларативної парадигми, тому при його використані користувач не описує конкретні дії, а лише бажані параметри об’єкту.

Зазвичай CDK системи “під капотом» використовують існуючі сервіси в відповідному хмарному середовищі, які надають можливість легше управляти ресурсами в ньому. Для AWS таким сервісом є Cloud Formation (див. рис. 3.3). AWS CloudFormation - це сервіс, який допомагає моделювати та налаштовувати ресурси AWS, щоб користувач міг витрачати менше часу на управління цими ресурсами і більше часу зосереджуватися на своїх додатках, які працюють в AWS. Ви створюєте шаблон, який описує всі потрібні вам ресурси AWS (наприклад, екземпляри Amazon EC2 або Amazon RDS DB), а CloudFormation подбає про забезпечення та налаштування цих ресурсів для вас [20]. В той же час CDK використовує API цього сервісу, надаючи можливість реалізувати ці шаблони на зручних мовах програмування.

Однією з ключових переваг CDK є його можливості моніторингу, порівняння та управління станом. CDK керує станом вашої хмарної інфраструктури та обчислює відмінності під час розгортання, зберігаючи цей стан у вигляді шаблону стека і пов'язаних з ним метаданих.

Після модифікації програми та її перерозгортання CDK виконує операцію diff. Ця операція синтезує код програми у певний шаблон, який представляє новий стан інфраструктури. Потім цей шаблон порівнюється з тим, який був розгорнутий раніше. Це порівняння визначає відмінності між бажаним станом (як визначено в коді CDK) і поточним станом (як зараз знаходиться в хмарі). На основі виявлених відмінностей CDK генерує набір конкретних кроків, необхідних для приведення поточного стану у відповідність до бажаного. Таким чином він

гарантує, що фактичний стан ресурсів відповідає бажаному стану, визначеному в конфігурації CDK.

У разі виникнення помилки в процесі розгортання, інструменти CDK здатні автоматично повернути зміни до останнього відомого "здорового" стану. Ця функція існує для підтримки стабільності та цілісності інфраструктури, тим самим зменшуючи ризик часткового розгортання, яке може призвести до непередбачуваних результатів.

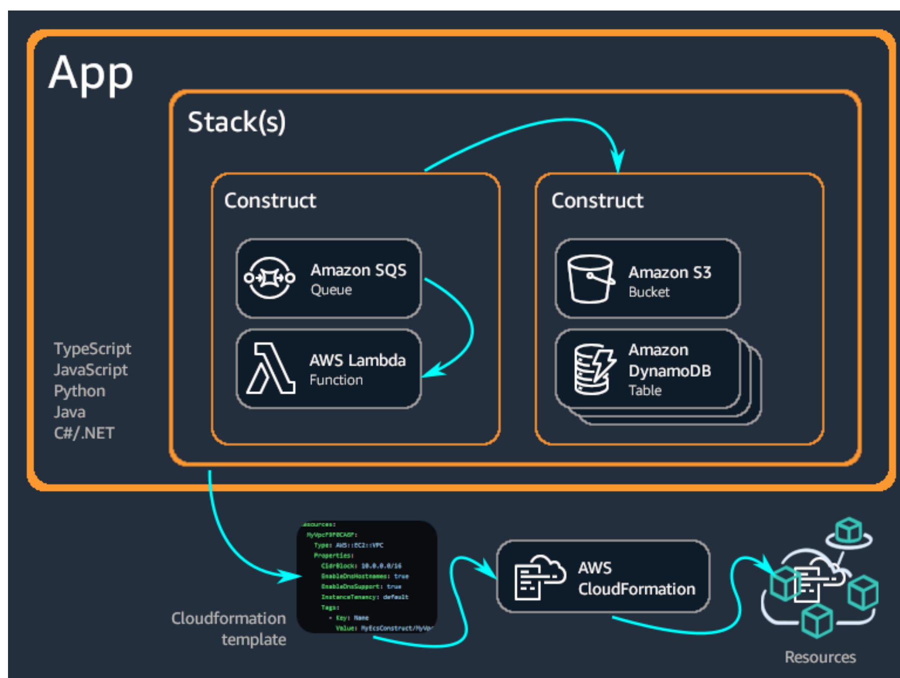



Рисунок 3.3 – Схема роботи AWS CDK та Cloud Formation (за даними [21])

Крім того, CDK легко інтегрується з існуючими робочими процесами розробки, дозволяючи розробникам використовувати свої улюблені IDE, системи контролю версій і фреймворки для тестування. Ця інтеграція полегшує уніфікований процес розробки, в якому інфраструктура та код додатків управляються разом. CDK також підтримує контроль версій, що дозволяє ефективно керувати змінами та повертати конфігурації, коли це необхідно.

Ще одним плюсом є те, що CDK використовує «конструктивні бібліотеки». Це заздалегідь зібрані модулі, які містять найкращі практики та загальні шаблони. Такі бібліотеки охоплюють широкий спектр сервісів, що дозволяє швидко збирати

складні архітектури без необхідності глибокого знання кожного сервісу хмари. Така модульність прискорює розробку і зменшує кількість помилок конфігурації, тим самим покращуючи ручні, схильні до помилок процеси налаштування, які покладаються на CLI і SDK.

Деякі CDK вводить поняття «стеків», які є одиницями розгортання, що пов'язують ресурси в одну логічну групу. Кожен стек можна розгортати і керувати ним незалежно, що спрощує управління великими і складними інфраструктурними рішенням. Стек являє собою набір ресурсів, які CDK розгортає та керує як єдиним цілим.

Незважаючи на свої переваги, CDK має свої недоліки. Як і SDK, він є специфічним для кожного окремого хмарного провайдера, а це означає, що код інфраструктури не можна переносити між різними провайдерами. У разі його зміни значна частина коду або весь код CDK проекту має бути переписана. Крім того, існує тенденція, яка передбачає, що нові функції та сервіси, які з'являються в провайдері, не завжди або з дуже великою затримкою з'являються в CDK репозиторіях. Хмарні компанії більше надають перевагу доповненню функцій веб-консолі та Command Line Interface, бо вони використовуються користувачами частіше. Через це є шанс, що деякі нові компоненти необхідно імплементувати самому або використовувати альтернативний підхід, наприклад, частину інфраструктури реалізувати на SDK.

CDK також може не тільки спрощувати, але й ускладнювати великі проекти. Управління залежностями та розуміння загальної архітектури може стати складним завданням у міру розширення визначень інфраструктури. Тому досить важливо впровадити ефективну документацію, організацію коду та дотримуватися найкращих практик, щоб ефективно керувати цією складністю.

Наведемо приклад використання AWS CDK на мові Python для створення тієї ж інфраструктури, яку ми описували за допомогою CLI та SDK.

```
from aws_cdk import core
import aws_cdk.aws_ec2 as ec2

class MyCDKStack(core.Stack):
```

```
        def __init__(self, scope: core.Construct, id: str, **kwargs) -> 
None:

            instance = ec2.Instance(self, "MyInstance",
              instance_type=ec2.InstanceType.of(ec2.InstanceClass.T2, 
ec2.InstanceSize.MICRO),
                machine_image=ec2.MachineImage.generic_linux({
                    "us-west-2": "ami-0abcdef1234567890"
                }),
                key_name="your-key-pair",
                vpc=vpc,
                vpc_subnets={"subnet_type": ec2.SubnetType.PUBLIC}
            )
            core.CfnOutput(self,"InstanceID", instance.instance_id)
    app = core.App()
    MyCDKStack(app, "MyCDKStack", env={'region': 'us-west-2'})
    app.synth(
```

Як можна помітити, такий опис дійсно є більш декларативним, тому що описує який ресурс ми хочемо мати та з якими характеристиками. На відміну від минулих методів, ту немає інструкцій, як саме це потрібно робити. Ми просто абстрактно описуємо систему, а CDK інтерпретатор вже сам розуміє, як йому це зробити. Це передбачає той факт, що CDK самостійно вираховує різницю між бажаним станом та реальним в хмарі. Тобто нам не треба додавати купу умовних операторів для покриття кейсів з додаванням, видаленням і редагуванням як в імперативних підходах. Розробнику потрібно просто описати, яким він хоче бачити ресурс або групу ресурсів, а інструмент визначить, які маніпуляції будуть проведені.

Крім того, багато хмарних постачальників мають зручний веб-інтерфейс для управління створеними стеками через CDK. Для прикладу знову візьмемо AWS CDK та пов'язаний Cloud Formation сервіс (див. рис. 3.4). В ньому, наприклад, можна моніторити процес створення або оновлення ресурсів, що були описані в стеці. Після створення також там відображається стан кожного ресурсу. Тобто користувачу не потрібно вручну робити запит для кожного компонента або шукати його в веб-консолі, щоб знайти помилку або переглядати відповідність до бажаного стану. Список всіх компонентів зібрано в одному місці. Це, очевидно, надає великої надійності системі та допомагає виявляти проблеми на ранньому етапі.

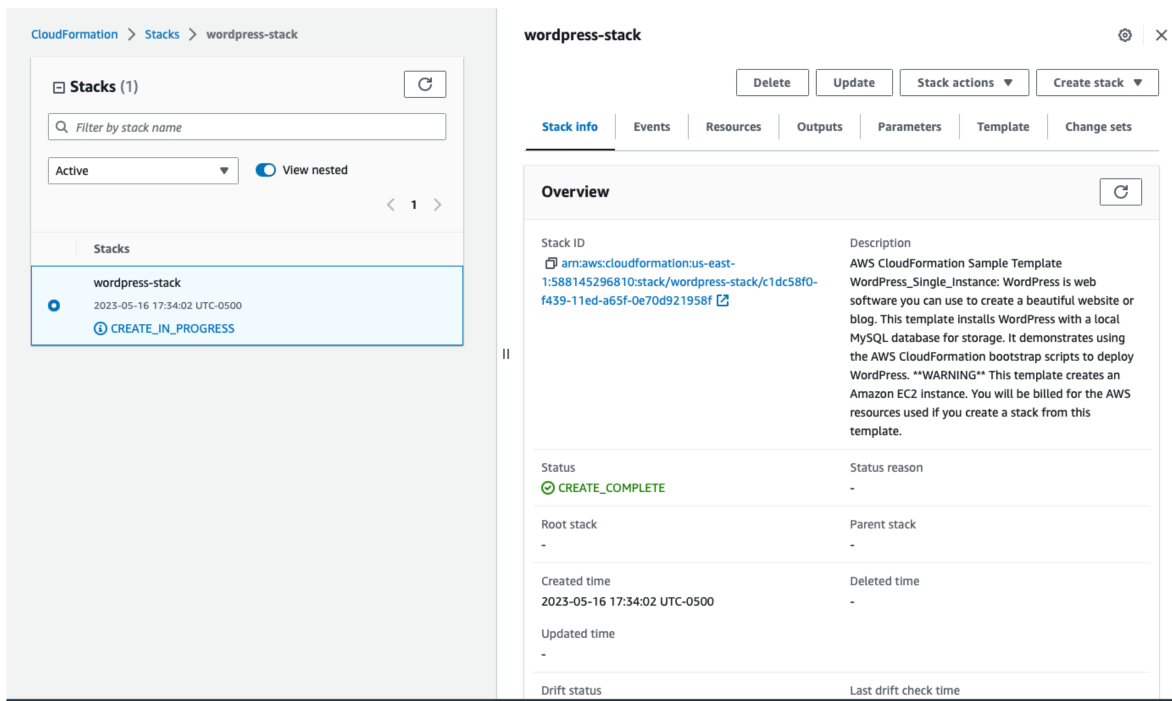



Рисунок 3.4 – Веб консоль AWS та Cloud Formation сервіс (виконано самостійно)

## 3.6 IaC vendor-neutral інструменти

Існує цілий ряд інструментів, які працюють подібно до підходу CDK. Їх методологія полягає в тому, що вони, на відміну від CDK, не прив’язуються до певного провайдера. Це означає, що в той час, коли кожний CDK інструмент має свою окрему імплементацію тісно пов’язану з інтерфейсом, що надає провайдер, зовнішні інструменти мають свою загальну абстрактну реалізацію процесу розгортання без прив’язки до певних хмарних ресурсів. Це означає, що вони містять алгоритми для управління станом, валідацію, вирахування потрібних операцій для розгортання, моніторингу тощо. І згідно з цим вони надають певний “контракт” для реалізації хмарним постачальникам, щоб ці відповідні алгоритми застосовувалися для потрібних видів ресурсів.

Наприклад, компанія, створиши свою власну хмару, замість реалізації власного CDK може взяти готове IaC рішення та, дотримуючись його вимог, швидко і безболісно додати власні типи хмарних інфраструктурних ресурсів, якими потрібно управляти. При цьому провайдеру не потрібно задумуватися, який саме низькорівневий код і яка логіка розгортання лежать в основі процесу. Вони вже є написаними розробниками цієї IaC системи, а користувачам необхідно

тільки підлаштуватись під правила такого програмного забезпечення. Очевидно, що такі системи створюються та підтримуються зовнішньою спільнотою, а не хмарними провайдерами. Найбільш розповсюдженими прикладами таких інструментів є Terraform, Crossplane, Pulumi та інші.

Такі інструменти стають все більш і більш розповсюдженими серед великих і маленьких компаній. Така популярність обґрунтовується цілім рядом переваг перед вищезазначеними методологіями та інструментами.

Перш за все, варто позначити, що більшість переваг є загальними та однаковими з вищеописаним принципом CDK.

Як сторонні IaC, так і CDK інструменти мають властивість абстрагуватися від складності низькорівневих хмарних операцій, надаючи високорівневі конструкції. Замість того, щоб безпосередньо взаємодіяти з API хмарного провайдера або низькорівневими конфігураційними файлами, користувачі можуть визначати компоненти інфраструктури за допомогою більш інтуїтивно зрозумілих і виразних конструкцій.

По-друге, звичайно, не можна не зазначити головну характеристику, що визначає ці дві методології – автоматичне управління станом. Як вже і було описано, для принципу, що є загальним для всіх IaC систем, є характерним відсутність необхідності для користувача слідкувати за реальним поточним станом інфраструктури, перед тим, як оновити певні ресурси. Підтримка стану автоматично визначить точні CRUD інструкції, необхідні для виконання, щоб привести систему в кінцеве положення.

Також варто відзначити високу відмовостійкість та гарантування надійності інфраструктури, що створюється за таким методом. Завдяки версіонуванню та можливості відкочування до певної версії, користувач завжди може бути впевнений, що розгортання не призведе до неочікуваних наслідків.

Окрім загальних переваг, якими володіють всі інструменти, що відносяться до IaC парадигми, є інші, специфічні для таких сторонніх мультихмарних систем. Серед таких можна визначити:

– підтримка декількох провайдерів одним інструментом;

– швидкість і легкість додавання підтримки нового провайдера;
– можливість використання не в хмарному контексті;
– стрімкий розвиток та додавання нового функціоналу;
– велика спільнота та підтримка;
– open-source;
– легка та вбудована інтеграція з існуючим CI/CD програмним забезпеченням;
– безкоштовність.

Головною перевагою перед всіма іншими принципами є хмарна універсальність. Така властивість передбачає той факт, що використовуючи тільки один інструмент, користувач може забезпечувати інфраструктуру для багатьох непов'язаних хмарних провайдерів. Це гарантує, що при терміновій необхідності міграції з одного провайдера на іншого, користувачам буде необхідно зробити мінімальну кількість дій для адаптування. Їм не буде потрібно переписувати весь проект через те, що реалізація для кожної хмари відрізняється на базовому рівні та рівні API. Так як vendor-neutral інструменти передбачають загальну реалізацію та інтерфейс однаковий для всіх, при міграції єдине, що потрібно зробити, це змінити провайдера, його облікові дані та провести деякі зміни в маніфестах ресурсів, якщо їх специфікація відрізняється від минулої хмари.

Всі популярні системи вже підтримують десятки й сотні існуючих провайдерів, тому проблем з різноманіттям немає. Незважаючи на це, в ситуаціях, коли виникає потреба використовувати певні IaC інструменти, але провайдер в ньому відсутній, розробники надають можливість розширити список. Зазвичай вони надають спеціальну інструкцію, згідно з якою будь-хто може додати власну реалізацію провайдера на вже готові кодовій базі. На відміну від створення CDK з нуля, такий спосіб є достатньо швидким та простим. Крім того, такий функціонал є дуже корисним при розробці власної невеликої хмари. Для того, щоб створити комфортний спосіб для управління такою хмарою, все що потрібно від її

розробників – мати будь-який API (REST, CLI тощо) для управління ресурсами та імплементувати наданий IaC інструментом інтерфейс.

Варто відзначити, що інструменти, такі як Terraform, Crossplane, Pulumi та інші, на початку створювалися для управління хмарною інфраструктурою, але завдяки їхній гнучкості та розширюваності вони знайшли широке застосування за межами простого управління хмарними ресурсами. Ці інструменти дають змогу описувати і управляти будь-якими ресурсами, що робить їх корисними не тільки для хмарних середовищ, а й для інших сфер діяльності. Ось кілька таких "не хмарних" прикладів, що присутні в реєстрах та для управління ресурсами яких можна використовувати такі сторонні IaC системи: GitHub, Jenkins, Datadog, Okta та інші.

Наступні три переваги дуже пов'язані між собою. Найбільш популярні представники "multicloud" методології мають величезну аудиторію користувачів. Завдяки тому, що для їх використання потрібно мати навички, більшість користувачів це розробники та DevOps інженери. Крім того, майже всі відповідні репозиторії є open-source. Це означає, що код системи є у відкритому доступі та кожний бажаючий може його переглядати та привносити щось нове. Поєднавши вищеописані факти про технологічно розвинену аудиторію та відкритий доступ, стає очевидно, що функціональність таких IaC програм розширюється надзвичайно швидкими темпами. В цей же час, CDK розробляється самим хмарним постачальником і нові функції додаються дуже рідко через інші пріоритети компаній.

Легка інтеграція з CI/CD передбачає те, що сторонні IaC системи часто мають певні вбудовані особливості, які дозволяють легко взаємодіяти з іншими сервісами для швидкого розгортання. Наприклад, у Jenkins користувачі можуть використовувати плагіни, розроблені спеціально для таких інструментів, як Terraform. Ці плагіни дозволяють пайплайнам Jenkins виконувати сценарії Terraform, керувати файлами стану та застосовувати конфігурації безпосередньо з середовища CI/CD.

Якщо компанія вирішує використовувати хмарні рішення в своїх проектах, то це означає, що для них грошове питання стоїть не на останньому місці. А наведені інструменти дозволяють зекономити хоча і невелику, але все ж таки певну суму. Це обґрунтовується тим фактом, що CDK інструменти та методи часто потребують певних витрат. Наприклад, як зазначалося, AWS CDK використовує AWS Cloud Formation сервіс для управління стеками та станом. А Cloud Formation, як і будь-який сервіс AWS вимагає певних витрат для його використання. В цей же час сторонні багатохмарні IaC інструменти не потребують нічого.

Звичайно, окрім переваг також завжди існують і недоліки.

По-перше, використання будь-якої хмари передбачає надавання певних приватних даних(облікові дані, публічні та приватні ключі тощо). І той факт, що зовнішній програмний додаток має інформацію про ці дані, звичайно додає певні безпекові сумніви. Але завдяки тому, що такі інструменти мають відкритий код та відповідну репутацію, зменшують цю проблему до мінімального рівня. Для повної гарантії, користувачі повинні обирати тільки популярні системи з великою спільнотою.

По-друге, додавання нових рівнів абстракції та патернів призводить до збільшення часу розгортання інфраструктури та зменшення продуктивності. Хоча ці інструменти спрямовані на спрощення операцій, вони можуть додавати певні затримки, особливо при великомасштабних розгортаннях. Це невід’ємний недолік складних систем, в яких заради надійності та простоти завжди потрібно чимось жертвувати.

Також один із недоліків, що може мати вирішальне значення у виборі користувачами інших методів розгортання інфраструктури є затримка в додаванні функцій, які підтримує хмарний постачальник. Тобто, якщо провайдер випускає новий сервіс або функціональність сервісу, вони з’являються у CLI, SDK, CDK значно швидше. Це обумовлюється тим фактом, що розробникам vendor-neutral IaC систем потрібен час, щоб розібратися в новому сервісі, реалізувати його та протестувати.

## 3.7 Порівняння методів

Для розуміння переваг будь-яких сервісів найкращим способом завжди є збір відгуків про їх використання в реальних користувачів. Для порівняння буде використана статистика з сервісу G2, що збирає рецензії про програмне забезпечення [22]. Для аналізу було обрано інструменти AWS як найбільш популярні та розповсюджені для більш точної статистики та Terraform як найбільш поширеного представника multicloud IaaS рішення. Але дані є пропроційними і для інструментів інших хмарних постачальників. Опитування було проведено для декількох груп користувачів в залежності від масштабу та складності їх проекту.

Першою групою є рецензенти з малого бізнесу (див. табл 3.1).

Таблиця 3.1 – результати оцінювання використання хмарних інструментів для малого бізнесу (за даними [22])

| Критерій/Продукт | AWS CLI | AWS CDK | Terraform | Web-console |
|---|---|---|---|---|
| Задовольняє потреби | **9.3** | 8.6 | 8.9 | 8.9 |
| Легкість використання | **9.0** | 8.5 | 8.5 | 8.3 |
| Легкість створення | N/A | 7.5 | 8.3 | **8.8** |
| Легкість адміністрування | N/A | 7.7 | **8.3** | 7.9 |
| Якість підтримки | 8.3 | **8.5** | 8.2 | 7.3 |
| Чи був продукт корисний для бізнесу? | N/A | 8.0 | N/A | **8.3** |
| Продуктовий напрям(% позитивності) | **10.0** | **10.0** | 9.2 | 8.4 |

Виходячи з наданих чисел, аналіз демонструє, що CLI є привабливим для користувачів, яким потрібна простота та ефективність в невеликих проектах завдяки високим оцінкам «задовольняє потреби» та «легкість використання». Також, орієнтуючись на інші високі оцінки, очевидно, що користувачі задоволені технічною підтримкою та напрямом, в який прагне йти AWS CLI.

Оцінки для AWS CDK демонструють, що, хоча користувачі і задоволені інструментом та його потенціалом, вони знаходять його значно складнішим у використанні ніж командний рядок.

Terraform в цілому має дуже схоже оцінювання на CDK. Незважаючи на це, легко помітно, що критерій складності використання відрізняється. Складність адміністрування та створення ресурсів за допомогою CDK рецензенти знаходять більшою ніж Terraform. Підсумовуючи ці оцінки, зрозуміло, що використання Terraform має ряд переваг перед CDK.

Крім того, цікаво подивитися, як користувачі оцінюють використання UI web console для управління хмарними ресурсами. Задоволення від легкості створення ресурсів важко не помітити – воно є вищим за інші методи. Така думка не є несподіваною, тому що графічний інтерфейс завжди надає кращий UX та інтуїтивну зрозумілість ніж скрипти та мови програмування. Але згідно з інших критеріїв використання консолі особливо не виділяється або навпаки відстає від інших.

Отже, проаналізувавши інформацію наведену вище, можна зробити підсумок, що користувачі малого бізнесу частіше за все знаходять CLI та скрипти найбільш зручним та часто використовуваним інструментом. Це зумовлюється тим фактором, що невеликі проекти зазвичай не мають складних схем інфраструктур з великою кількістю ресурсів. Завдяки цьому в них немає критичної потреби витрачати час на налаштування непростого DevOps процесу та вивчення IaC принципів.

Іншою групою рецензентів є підприємства з великого бізнесу (див. табл. 3.2).

Таблиця 3.2 – результати оцінювання використання хмарних інструментів для складних проектів (за даними[22])

| Критерій/Продукт | AWS CLI | AWS CDK | Terraform | Web-console |
|---|---|---|---|---|
| Задовольняє потреби | 8.6 | 9.0 | **9.2** | 8.5 |
| Легкість використання | 8.6 | 8.1 | **8.7** | 8.5 |
| Легкість створення | N/A | 8.1 | **9.2** | **N/A** |
| Легкість адміністрування | N/A | 8.1 | **9.5** | N/A |
| Якість підтримки | 8.3 | 8.3 | **8.5** | 8.3 |
| Чи був продукт корисний для бізнесу? | N/A | 8.5 | **9.5** | **N/A** |
| Продуктовий напрям(% позитивності) | 8.3 | 9.2 | **10.0** | 6.1 |

Для корпорацій великого бізнесу з відповідно великими та складними проектами, статистика значно відрізняється від наведеної для малого.

Перш за все варто зазначити, що оцінки CLI та веб-консолі стали дуже схожими між собою і в цей час нижчими ніж в минулій статистиці. З цього випливає висновок, що такі прості інструменти вже не можуть задовільнити незвичайні та складні потреби систем з точки зору управління хмарною інфраструктурою. В той же час складно не помітити, що рейтинг CDK інструменту хоч і не значно, але покращився. Особливо це помітно в оцінюванні корисності для бізнесу. Оцінка виросла на 0.5 пунктів. Завдяки паралельному спаду балів для веб консолі та CLI і їх росту для CDK, можна визначити, що пріоритетність для користувачів різко змінилася. Тепер з збільшенням складності

проектів, пріоритетом стає надійність, масштабованість, гнучкість та адаптивність, яку надає IaC принцип.

Якщо говорити про статистику для Terraform, то навіть без детального аналізу видно, що він перемагає всі інші методи та інструменти по кожному з критеріїв. Ця інформація насправді не є неочікуваною. Як вже було зазначено, помітно, що великі проекти надають перевагу CDK як IaC інструменту. В той же час, згадуючи аналіз, який було проведено в даній роботі раніше, було зазначено, що multicloud IaC рішення мають всі плюси CDK принципу, але ще й додають ряд своїх перед ним. Підсумовуючи цю статистику, очевидно, що великі користувачі з складними проектами для створення ресурсів хмарної інфраструктури нададуть перевагу стороннім vendor-neutral IaC рішенням в більшості випадків.

Якщо говорити про загальні пріоритети в світі хмарної інфраструктури, то стає зрозуміло, що не можна чітко визначити переможця, якого б обирала абсолютна кількість рецензентів. Вибір методу та інструменту залежить від конкретних потреб клієнта в конкретній ситуації. Наприклад, для компаній, в яких інфраструктура полягає в управлінні парою серверів та базою даних, вибір паде на простий спосіб, такий як скрипт з використанням командного рядку. В той же час великі корпорації з величезною кількістю ресурсів та потребами віддадуть перевагу складному сторонньому IaC програмному забезпеченню.

# 4 ПОРІВНЯННЯ VENDOR-NEUTRAL EXTERNAL IaC РІШЕНЬ

Для дослідження та порівняння IaC рішень було обрано Terraform як найбільш розповсюджену систему в поточний час та Crossplane як більш нову систему, що стрімко набирає популярність серед всіх категорій користувачів IaaS сфери.

## 4.1 Terraform

Terraform являє собою типовий IaC інструмент, що використовує декларативну парадигму. Він був розроблений компанією HashiCorp та був одним з перших реалізацій інфраструктури як коду. Завдяки цьому він завоював ринок та займає першість й у наш час.

Як джерело конфігурації для хмарної інфраструктури Terraform використовує файли з розширенням “.tf”. Такі файли пишуться на спеціальній мові HashiCorp Configuration Language, яка була розроблена HashiCorp та використовується для її програмного забезпечення. HCL є дуже схожим на JSON, тому легко читається людьми. Приклад такого коду наведений нижче.

```
provider "aws" {
  region = "us-west-2"
}
resource "aws_instance" "example" {
  ami           = "ami-0c55b159cbfafe1f0"
  instance_type = "t2.micro"
  tags = {
    Name = "ExampleInstance"
  }
}
```

Завдяки своїй розповсюдженості Terraform вже має сотні написаних реалізацій популярних і не тільки хмарних провайдерів. Найбільш використовувані можна знайти в спеціальному публічному реєстрі. Завдяки цьому при обранні популярного постачальника для своєї інфраструктури, користувачам не потрібно реалізовувати Terraform провайдера для них. Але у випадку, якщо користувач хоче обрати хмару, якої ще немає в цій системі, то розробники надали можливість створювати власних провайдерів. В мережі існує велика кількість

інструкцій для їх створення, тому така задача не є надзвичайно складною, хоча можуть виникати питання.

Terraform надає велику кількість інтуїтивно зрозумілих команд командного рядка, що дозволяють легко виконувати потрібні операції. Наприклад, використовуючи команду «terraform plan» користувач може детально переглянути поточний стан інфраструктури та які зміни в неї планує вносити система. Команда «terraform apply» вже фактично вираховує різницю в фактичному і бажаному стані в поточну секунду та починає процес розгортання. Для запобігання внесення випадкових змін, програма просить користувача підтвердити намір своїх дій. В той же час команда «terraform apply» навпаки дозволяє повністю видалити всі ресурси, що є описаними в конфігураційних файлах.

Поточний стан системи зберігається в спеціальному файлі. Такий файл зберігає стан ресурсів в існуючій інфраструктурі, що дозволяє системі вираховувати, які саме зміни потрібно вносити до того або іншого ресурсу для досягнення бажаного положення.

Основні етапи, які відбуваються в Terraform:

– застосування конфігураційних файлів, в яких описуються хмарний провайдер, вхідні дані та ресурси;
– ініціалізація модуля під час якого система завантажує необхідні плагіни та провайдера згідно з конфігураційними файлами;
– створення файлу планування, в якому система записує вирахувані зміни, необхідні для оновлення або створення ресурсів хмари;
– застосування змін отриманих під час минулого кроку планування та очікування поки ресурси перейдуть в готовий стан;
– збереження нового стану хмари в файл стану після розгортання.
– повторення всіх минулих дій при необхідності застосування нових вимог до інфраструктури.

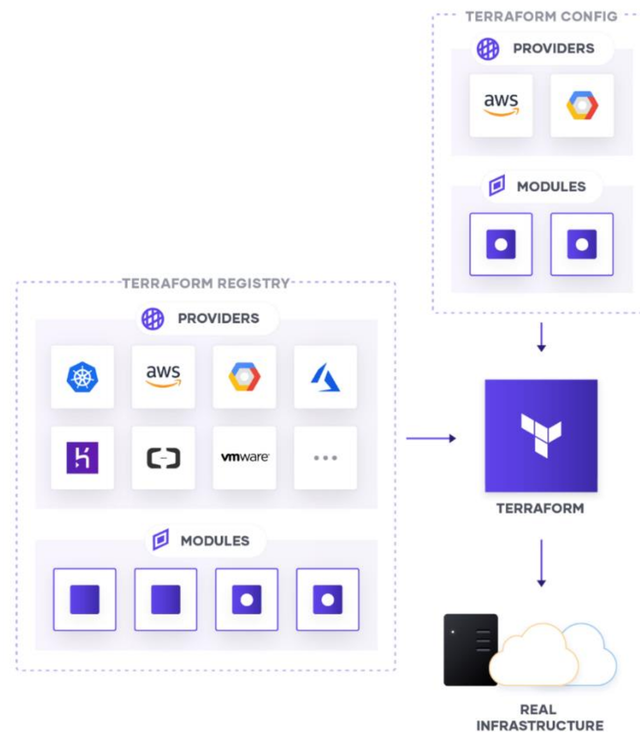



Рисунок 4.1 – Схема роботи Terraform (за даними [23])

## 4.2 Crossplane

Як і Terraform, Crossplane є представником декларативного принципу.

Crossplane являє собою платформу, що дозволяє управляти інфраструктурою, використовуючи кластер Kubernetes. Весь процес розгортання відбувається з Kubernetes кластеру через його власний API.

Через те що ця система відносно нова, вона не є такою розповсюдженою, але різко набирає популярність та входить в топ IaC інструментів.

Як джерело конфігурації Crossplane використовує типовий тип даних для Kubernetes – yaml. YAML - це людиночитаємий формат даних схожий на JSON, який використовується для опису та передачі текстових даних.

Прикладом такої конфігурації є наступний код.

```yaml
apiVersion: ec2.aws.crossplane.io/v1alpha1
kind: Instance
metadata:
  name: example-instance
spec:
  forProvider:
    region: us-west-2
```

```
      ami: ami-0c55b159cbfafe1f0
      instanceType: t2.micro
      tags:
        - key: Name
          value: ExampleInstance
    providerConfigRef:
      name: default
```

Crossplane також має реєстр з різноманітними готовими провайдерами, які користувачі можуть використовувати для своїх потреб. Також, як і Terraform, розробники Crossplane залишили можливість розширювати список провайдерів своїм власним.

На відміну від Terraform, який запускається з терміналу, як звичайна програма, Crossplane працює як розширений плагін для Kubernetes кластеру. Kubernetes початково розроблювалась платформа для оркестрації та управління контейнерами [24]. Але окрім управління контейнерами, вона має велику кількість іншого функціоналу, що допомагає в розгортанні сервісів. Kubernetes надає ряд типів ресурсів, які користувач може створювати для управління кластером. Такими, наприклад, є Deployment, ReplicaSet, StorageClass, Service та багато інших. Кожний тип ресурсу має свій контролер. Контролери Kubernetes безперервно порівнюють бажаний стан, зазначений кожним ресурсом, з реальним станом кластера і вносять необхідні коригування. Цей процес називають «циклом узгодження», оскільки він весь час повторюється у спробі узгодити поточний стан із бажаним.

Окрім базових видів ресурсів, що надає Kubernetes, він також надає функціонал для додавання своїх типів ресурсів та логіки для їх обробки - Custom Resource Definitions. CRD – це опис об'єкту певного ресурсу певної доменної області, що є схожим на визначення класів в ООП. Користувачі можуть писати власні визначення певних ресурсів і робити так, щоб спеціальні сервіси «оператори» виконували певну логіку над ними (див. рис. 4.2) [25].

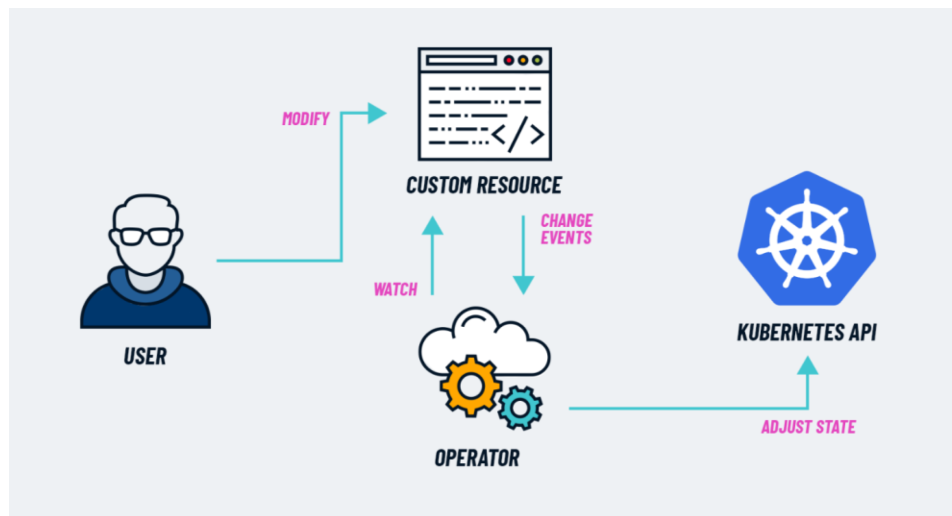



Рисунок 4.2 – Kubernetes operator pattern (за даними [25])

Цю функціональність і застосовує Crossplane, використовуючи безліч переваг, які надають оператори. При встановлені провайдера на кластер, користувач автоматично встановлює CRDs, що відповідають хмарним ресурсам. Також встановлюються оператори – сервіси, які управляють цими ресурсами. Кожний оператор слідкує за списком ресурсів свого типу. Коли до ресурсів приходять сторонні зміни, оператор помічає це і виконує певну логіку, яка в нього закладена. Цей алгоритм повинен бути таким, щоб ці зміни дійсно відображались в тому об'єкті, який характеризує CRD. Також, окрім слідкування за подіями, що відбуваються з ресурсами, такі оператори періодично перевіряють, чи відповідає стан ресурса на кластері, тому реальному об'єкту, які він відображає. У негативному випадку, оператор проводить ту ж логіку, що й під час надходження подій.

Crossplane також зберігає поточний реальний стан існуючих ресурсів, але робить він це не в файлі, а в специфічній базі даних для Kubernetes – etcd.

Основні етапи, що відбуваються в Crossplane під час внесення змін:

– застосування користувачем yaml файлів конфігурації на кластер за допомогою спеціального API;
– реагування оператором на зміни в певному ресурсі;
– порівняння стану застосовного Kubernetes ресурсу та стану відповідного об’єкту в інфраструктурі;

– виконання необхідної логіки для синхронізації та очікування успішного статусу;
– оновлення нових параметрів та статусу ресурсу на кластері, щоб він відображав реальний стан;
– продовження циклу слідкування та періодичне порівняння бажаного та реального положення.

## 4.3 Порівняння Crossplane і Terraform

### 4.3.1 Теоретичне порівняння

Але для визначення кращого IaC рішення поміж Terraform та Crossplane, визначимо основні переваги та недоліки кожного з них (див. табл. 4.1).

Перш за все, Terraform виділяється тим, що він є найбільш зрілою та стабільною системою. Вона існує вже досить довгий час, тому має найбільше ком’юніті в цій сфері. Це призводить до того, що більша кількість людей довіряє такому програмному забезпеченню, вносить більший вклад та допомагає новим користувачам. Також це означає, що Terraform має більшу кількість матеріалів, щоб швидко освоїти цю технологію. Крім того, завдяки великій підтримці на даний час він підтримує більшу кількість хмарних провайдерів.

Наступною невеликою перевагою є той факт, що цей інструмент надає можливість спочатку побачити заплановані зміни, і в певній ситуації скасувати розгортання. В ситуації з Crossplane після застосування маніфестів ресурсів, ви не можете скасувати розгортання та побачити точні зміни – тільки поточний та майбутній стан.

Крім того, Terraform зберігає поточний стан в окремому файлі, який можна зберігати, передавати та управляти за допомогою систем контролю версій. Crossplane же зберігає стан в самих ресурсах Kubernetes, які в свою чергу зберігаються в базі даних etcd. Міграція etcd можлива, але це не є такою простою задачею, як передача файлу.

Якщо говорити про переваги Crossplane, то їх кількість більша, ніж можна очікувати від відносно нового інструмента.

По-перше, сам факт того, що ця платформа є інтегрованою в Kubernetes екосистему надає велику кількість переваг. Kubernetes в поточний час є однією з найбільш затребуваних та важливих інструментів в світі cloud-native сфери. Вона вважається "золотим стандартом" в світі контейнерів та оркестрації. Рідко який великий проект в наш час існує без використання цієї системи оркестрації. Завдяки цьому більшість DevOps інженерів вже знайома з її екосистемою та механізмами, тому час навчання буде мінімальним.

По-друге, завдяки Kubernetes Crossplane використовує шаблон оператора та CRDs. Цей шаблон забезпечує унікальні можливості, яких немає в Terraform.

Найбільш важливою характеристикою цього паттерну є той факт, що оператори постійно порівнюють стан ресурсів в хмарі та стан ресурсів на кластері. Це гарантує уніфікованість та надійність. Якщо певний користувач робить сторонні зміни в ресурсі на хмарі без внесення змін в конфігурацію на кластері, це називається "drift changes", Crossplane автоматично бачить такі зміни та відкачує їх зміни, щоб бажана конфігурація ресурсів в маніфестах точно відповідали тим, які вже задеплоєні. В цей же час Terraform не підтримує слідкування за "drift changes" не дізнається про віддаленні маніпуляції над ресурсами, доки користувач вручну не запустить відповідну команду.

Terraform може розвалитися, коли для управління інфраструктурою організації необхідно залучити більше інженерів. Terraform покладається на монолітний файл стану для зіставлення бажаної конфігурації з фактичною інфраструктурою, що працює. Під час застосування конфігурації цей файл стану повинен бути заблокований, а застосування конфігурації Terraform - це процес блокування, який може зайняти хвилини. Протягом цього часу жодна інша організація, жоден інший інженер - не може вносити зміни в конфігурацію. Аналогічно, Terraform використовує монолітний процес «застосування» - не рекомендується змінювати лише один елемент інфраструктури в конфігурації. Якщо ви використовуєте одну і ту ж конфігурацію для управління кешами і

базами даних, ви завжди повинні оновлювати обидві - ви не можете оновлювати тільки кеші [26].

Крім того, якщо під час процесу синхронізації станів об’єкту трапляється помилка, оператори будуть нескінченно намагатися повторити операцію раз в деяких період часу, а Terraform просто поверне помилку і закінчить виконання.

У доповнення до минулої функції, кожен оператор відповідає за слідкування та управління тільки своїм видом ресурсу, тому при розгортанні декількох видів одночасно, кожен оператор буде виконуватися паралельно, що очевидно додає швидкості. В цей час Terrform, хоч і використовує асинхронність для очікування на успішну відповідь від створеного чи зміненого ресурсу, але операції ініціювання операції синхронізації виконуються по черзі одна за одною.

В додаток до всього Terraform переймає всі плюси поєднання CI/CD інструментів з Kubernetes платформою. Одним із прикладів є можливість використання Helm для шаблонізації маніфестів і ArgoCD для автоматичної синхронізації Kubernetes ресурсів з різними видами репозиторіїв. Наприклад, це дозволяє маніфестам автоматично з’являтися на кластері, якщо вони додаються до GitHub репозиторію користувача без додаткових інтеграцій та налаштувань.

На відміну від Terraform, який не має ніяких механізмів авторизації та аутентифікації, Crossplane наслідує механізм RBAC від Kubernetes. Такий механізм дозволяє налаштовувати права, хто і які ресурси може створювати, змінювати та видаляти. Також виникає питання це зберігати облікові дані для авторизації до хмари від лиця IaaS інструменту. Для Terraform вони кладуться в звичайний файл, а Crossplane використовує спеціальні зашифрований вид ресурсів “Secret”, на який посилаються інші ресурси. Це додає йому ще кілька пунктів з точки зору безпеки і надійності.

Завдяки Kubernetes, Crossplane також може отримати перевагу в вигляді автоматичного масштабування. Платформа оркестрації надає різні інструменти для цієї задачі. Наприклад, вбудований Vertical Pod Autoscaler дозволяє подам динамічно та автоматично при необхідності збільшувати кількість віртуальних ресурсів виділених під певний под. Така функція є дуже корисною під час

розгортання великих інфраструктур, коли навантаження та використання ресурсів перевищує очікування.

Таблиця 4.1 – результати оцінювання використання хмарних інструментів для складних проектів (виконано самостійно)

| Критерій | Terraform | Crossplane |
|---|---|---|
| Велика спільнота | + | +- |
| Планування змін перед розгортанням | + | - |
| Легке перенесення стану в інші місця | + | - |
| Можливість паралельної роботи | - | + |
| Повторні спроби розгортання при помилці | - | + |
| Автоматичне виявлення та відкат “drift changes” | - | + |
| Паралелізм | Асинхронність | Паралелізм за типом ресурсу |
| Зручна інтеграція з CI/CD | +- | + |
| Вбудований RBAC | - | + |
| Вбудовані можливості масштабування | - | + |

4.3.2 Практичне порівняння

Однією з ключових характеристик будь-якого IaaS інструменту є швидкість розгортання. Це означає час, починаючи з процесу внесення змін в конфігураційний файл, закінчуючи моментом, коли всі ресурси переходять в готове положення. Дуже важко теоретично оцінити час для кожного інструменту і особливо порівняти їх через абсолютно різні підходи до кожного кроку в процесі розгортання.

Для визначення кращого кандидата проведемо практичне дослідження.

В процесі дослідження було реалізовано декілька програм та підготовано середовище для тестування. В якості хмарного провайдера було обрано AWS

через його популярність та ряд переваг перед іншими постачальниками. Також було створено локальний Kubernetes кластер за допомогою утиліти Kind. Перед початком тестування також були завантаженні та встановлені відповідні модулі AWS провайдера для Terraform та Crossplane. Для доступу до хмари було сгенеровано облікові дані та збережено в відповідному місці для кожного інструменту.

Програмна реалізації для вимірів реалізувалися на мові Python. Конфігураційні файли писались на відповідних для кожного інструмента мовах – HCL та yaml.

Час виконання Terraform замірювався від початку виклику операції до її закінчення. Це гарантувало, що ресурси перейшли в здоровий стан, або були повністю видалені. Нижче наведений код основних методів програми.

```
def run_command(command):
    start_time = time.time()
    subprocess.run(command, shell=True, check=True)
    end_time = time.time()
    return end_time - start_time

def terraform_apply():
    print("Applying Terraform configuration...")
    time_taken = run_command("terraform apply -auto-approve")
    print(f"Terraform provisioning time: {time_taken:.2f} seconds")
    return time_taken

def terraform_destroy():
    print("Deleting Terraform configuration...")
    time_taken = run_command("terraform destroy -auto-approve")
    print(f"Terraform destroy time: {time_taken:.2f} seconds")
    return time_taken
```

Для дослідження було обрано певний ресурс(DynamoDB Table) та його кількість. В наступному коді наведний приклад опису конфігураційного .tf файлу.

```
resource "aws_dynamodb_table" "dynamodb-testdb" {
  count          = 250
  name           = "db-${count.index}"
  billing_mode   = "PAY_PER_REQUEST"
  hash_key       = "PriKey"
  range_key      = "Quantity"
  attribute {
    name = "PriKey"
```

```
      type = "S"
    }
    attribute {
      name = "Quantity"
      type = "N"
    }
  }
```

За допомогою тестової програми він декілька разів розгортався на хмарі, поки кількість тестів збільшувалась та проводились заміри (див. рис 4.3, 4.4).

```
Plan: 0 to add, 0 to change, 30 to destroy.
aws_dynamodb_table.dynamodb-testdb[27]: Destroying... [id=db-27]
aws_dynamodb_table.dynamodb-testdb[25]: Destroying... [id=db-25]
aws_dynamodb_table.dynamodb-testdb[1]: Destroying... [id=db-1]
aws_dynamodb_table.dynamodb-testdb[3]: Destroying... [id=db-3]
aws_dynamodb_table.dynamodb-testdb[15]: Destroying... [id=db-15]
aws_dynamodb_table.dynamodb-testdb[22]: Destroying... [id=db-22]
aws_dynamodb_table.dynamodb-testdb[7]: Destroying... [id=db-7]
aws_dynamodb_table.dynamodb-testdb[8]: Destroying... [id=db-8]
aws_dynamodb_table.dynamodb-testdb[14]: Destroying... [id=db-14]
aws_dynamodb_table.dynamodb-testdb[13]: Destroying... [id=db-13]
aws_dynamodb_table.dynamodb-testdb[15]: Destruction complete after 1s
```

Рисунок 4.3 – Хід виконання програми для тестування (виконано самостійно)

```
Destroy complete! Resources: 10 destroyed.
Terraform destroy time: 63.14 seconds
```

Рисунок 4.4 – Приклад виводу результату тестування (виконано самостійно)

Виконавши ряд тестів з різними вхідними даними, створимо підсумкову таблицю з результатами тестування для Terraform (див. табл. 4.1).

Таблиця 4.1 – Результати замірів швидкості виконання для Terraform (виконано самостійно)

| Кількість\Операція | Створення | Видалення |
| --- | --- | --- |
| 1 | 15.87 с | 14.03 с |
| 30 | 38.12 с | 27.27 с |
| 100 | 109.75 с | 75.97 с |
| 250 | 224.27 с | 176.63 с |

| 500 | 434.72 с | 402.25 с |
|---|---|---|

В таблиці відображено кореляцію часу виконання всіх операцій в секундах від кількості ресурсів, що були розгорнуті паралельно. Як можна помітити, час збільшується пропорційно до збільшення числа об’єктів, що створюються. З першого погляду можна припустити, що така залежність є лінійною. Для впевненості створимо рівняння лінійної регресії та побудуємо графіки з результатами (див. рис. 4.5).

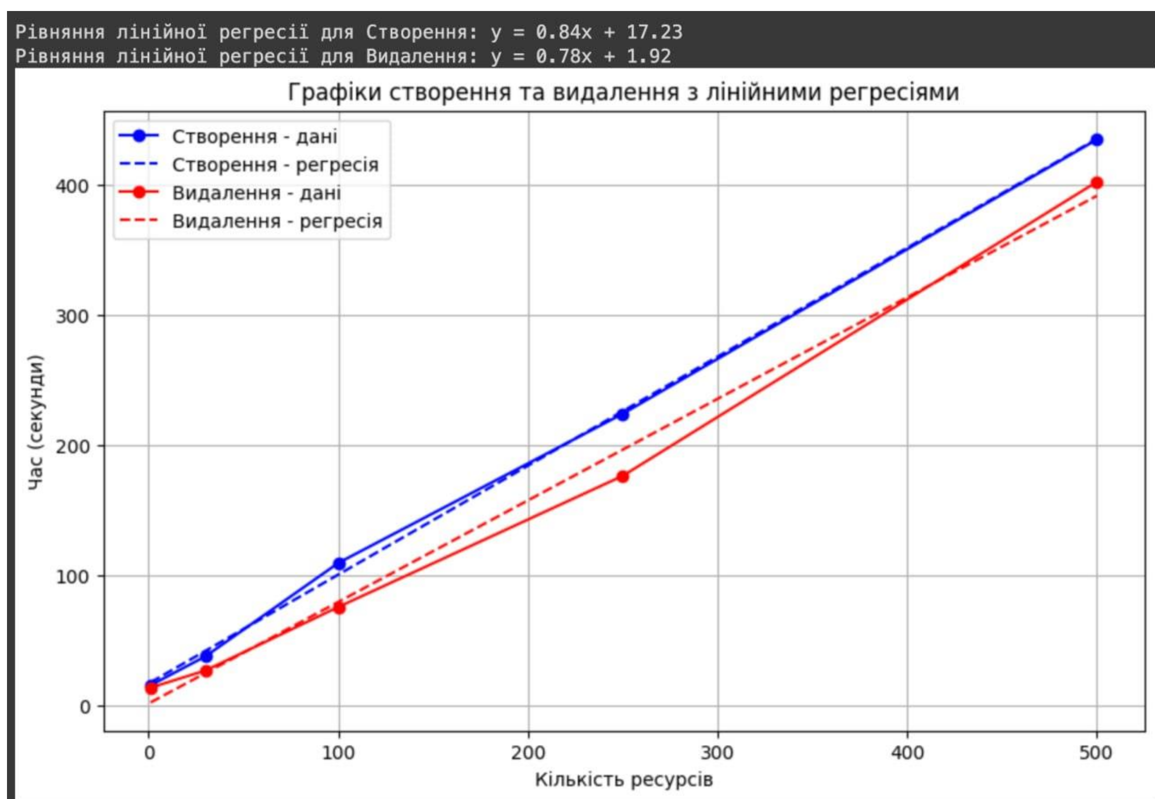



Рисунок 4.5 – Графік регресії для вимірів швидкості Terraform (виконано самостійно)

Зображення містить окремі рівняння лінійної регресії для кожної операції та відповідні графіки залежності з значеннями, по яких розраховувалась регресія. Як можна помітити на графіку відсутні точки аномалій, тому можна стверджувати, що регресія досить точно відображає кореляцію вхідних та вихідних даних. Завдяки цим рівнянням тепер можна приблизно передбачити очікуваний час

розгортання для будь-якої кількості цього виду ресурсів, що може допомогти в процесах оптимізації.

Тепер створимо таку ж таблицю з результатами Crossplane тестування з такими ж вхідними даними (див. табл. 4.2).

Таблиця 4.2 – Результати замірів швидкості виконання для Crossplane (виконано самостійно)

| Кількість/Операція | Створення | Видалення |
|---|---|---|
| 1 | 11.74 с | 24.40 с |
| 30 | 21.07 с | 27.98 с |
| 100 | 35.98 с | 43.04 с |
| 250 | 61.74 с | 77.38 с |
| 500 | 116.40 с | 150.55 с |

Результати для Crossplane також відображають лінійну залежність, тому для них також можна побудувати лінійну регресію (див. рис. 4.6).

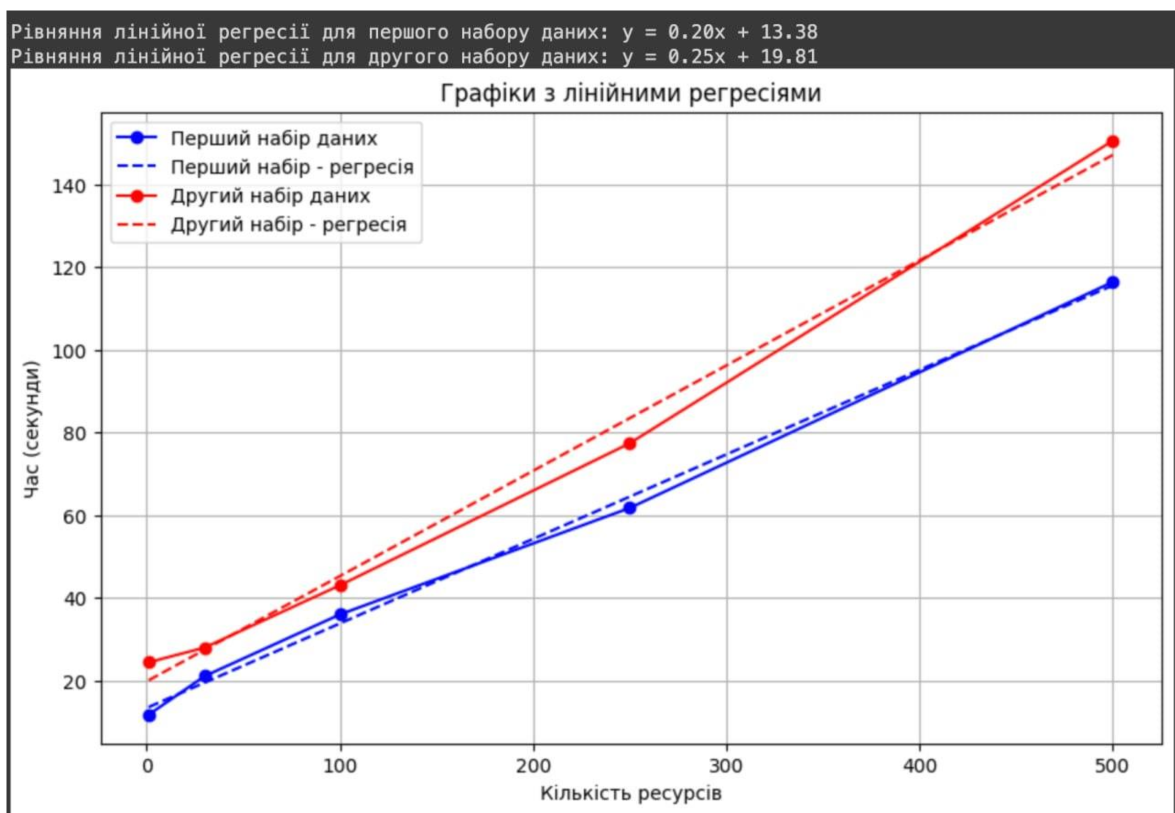


Рисунок 4.6 – Графік регресії для вимірів швидкості Crossplane (виконано самостійно)

Опираючись на ці графіки, як і для підсумків Terraform тестів, можна також стверджувати, що отримані рівняння досить точно відображають кореляцію. Точки аномалій є відсутніми.

Якщо порівнювати результати видалення та створення, можна помітити певну закономірність. В тестуванні Terraform видно, що швидкість створення є більшою за швидкість видалення, а в Crossplane рівно навпаки.

Порівнявши отримані результати не складно зробити висновок, що швидкість роботи Crossplane та Terraform на зовсім малих об’ємах дуже близька. Але збільшуючи об’єми об’єктів та ускладнюючи інфраструктуру, Terraform починає програвати в декілька разів, і ця різниця росте в геометричній прогресії. Наприклад, для 30 ресурсів різниця полягає в 2 рази, а для 500 вона збільшується до 4 разів.

Поєднавши результати цього дослідження з теоретичним порівнянням наведеним в цій роботі раніше, можна зробити певний висновок. Незважаючи на велику спільноту Terraform та його репутацію, Crossplane переважає його по великій кількості характеристик. Орієнтуючись на стрімкий розвиток Crossplane, можна очікувати, що нагорода за першість з великою вірогідністю може перейти до нього.

# 5 ПРАКТИЧНА РЕАЛІЗАЦІЯ ПРОВАЙДЕРА

Відповідно до отриманих результатів проведеного аналізу було виявлено, що Crossplane можна визнати одним з найкращих vendor-neutral IaC інструментів по багатьом критеріям. Щоб краще на практиці зрозуміти, як саме працює Crossplane та які процеси проходять під час його роботи, було вирішено виконати мінімально працюючу реалізацію для хмарного провайдера, якого ще немає в реєстрі Crossplane.

## 5.1 Вибір хмарного постачальника для дослідження

Для вибору піддослідного постачальника було проаналізовано IaaS ринок та обрано того, який має зручні для імплементації ресурси IaaS та має демонстраційний період без плати за використання. Вибір впав на провайдера під назвою Kamatera [27]. Він є глобальним та відносно популярним постачальником платформи хмарних послуг, який пропонує IaaS продукти організаціям будь-яких типів та розмірів. Kamatera має реалізацію в деяких IaaS рішеннях, але в Crossplane вона відсутня, тому ця платформа буде гарним піддослідним в нашому дослідженні (див. рис. 5.1).

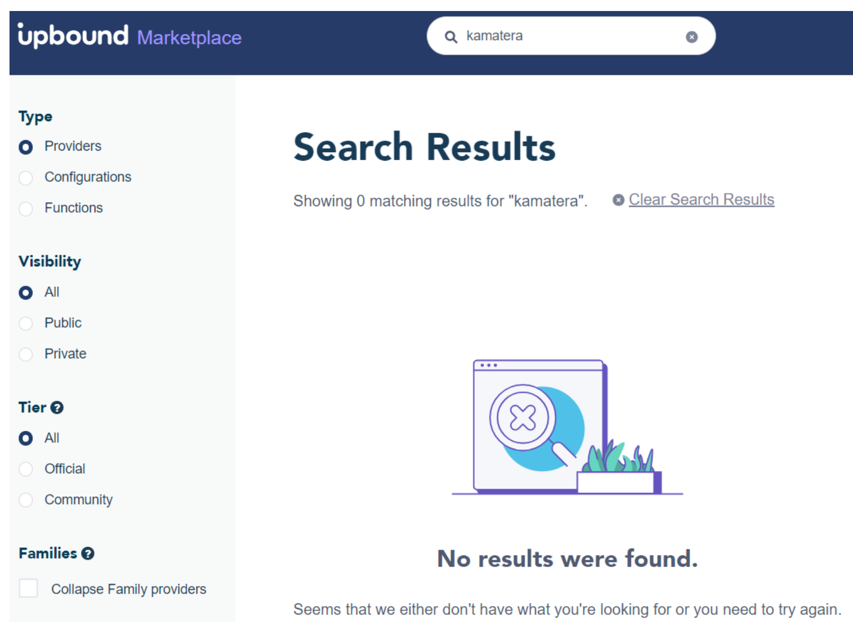



Рисунок 5.1 – Демонстрація відсутності Kamatera імплементації в реєстрі Crossplane (виконано самостійно)

## 5.2 Процес реалізації

Так як Kubernetes оператори пишуться на Golang, імплементація провайдера також була виконана на цій мові. Розглянемо структуру проекту та схему роботи більш детально.

Перш за все, найважливішою частиною роботи Crossplane як частини Kubernetes платформи є визначення користувацьких ресурсів, які будуть управлятися системою. В кожному ресурсі повинно бути присутнім поле Parameters. Цей об’єкт буде містити в собі всю можливу конфігурацію, яку може задавати користувач. Тобто з IaC точки зору цей об’єкт описує бажаний стан ресурсу. В наведеному коді наводиться структура, яку буде задавати користувач при створенні ресурсу на кластері в якості бажаного стану створюваного об’єкту.

```
type NetworkParameters struct {
  Name   string    json:"name"    // Required
  Subnet []*Subnet json:"subnets" // Optional

  // +crossplane:generate:reference:type=topology/v1.Datacenter
  // +crossplane:generate:reference:refFieldName=DatacenterIDRef
  // +optional
  DatacenterID string json:"datacenter"
  DatacenterIDRef *xpv1.Reference json:"datacenterIdRef,omitempty"
  DatacenterIDSelector *xpv1.Selector
json:"datacenterIdSelector,omitempty"
}

type Subnet struct {
  IP      string  json:"ip"                // Required
  Bit     int     json:"bit"               // Required
  Gateway *string json:"gateway,omitempty" // Optional
  // +optional
  DNS1 *string json:"dns1,omitempty" // Optional
  // +optional
  DNS2 *string json:"dns2,omitempty" // Optional
  Description string json:"description"
}
```

Ці параметри містять, як користувач хочу бачити мережу. В це входять опис, список підмереж та датацентр, в якому мережа буде існувати. Datacenter буде імплементовано окремим CRD, тому Crossplane надає можливість посилатися на іншу структуру за допомогою референсів або селекторів.

Наступною структурою, яка буде частиною Network ресурсу є Observation. В цій частині розробник постачальника визначає характеристики ресурсу, якими зараз володіє реальний об'єкт на хмарі після його створення. Тобто він відображає вже не бажаний, а реальний стан "snapshot" в певний момент часу. Зазвичай такі поля використовуються для порівняння та визначення того, чи відповідає реальне положення тому, якого від нього очікують. Ця структура під час роботи системи заповнюється не користувачем, а даними, що повертаються з API, який надає хмарний постачальник.

```
type NetworkObservation struct {
  Subnets []*SubnetStatus json:"subnets,omitempty"
  Id      int             json:"id"
}

type SubnetStatus struct {
  Datacenter        string  json:"datacenter"
  VlanID            int     json:"vlanId"
  SubnetID          int     json:"subnetId"
  StartRange        string  json:"startRange"
  EndRange          string  json:"endRange"
  SubnetIp          string  json:"subnetIp"
  SubnetBit         int     json:"subnetBit"
  SubnetDescription string  json:"subnetDescription"
  DNS1              *string json:"dns1,omitempty"
  DNS2              *string json:"dns2,omitempty"
  Gateway           *string json:"gateway,omitempty"
  InUse             int     json:"inUse"
}
```

Після виконання певного набору команд, генерується yaml файл, що містить схему ресурсу з переліком параметрів, полів та валідацій відповідно до Golang структур описаних вище. Не застосувавши такий файл до Kubernetes кластеру, він не буде знати про існування такого типу об'єкту, тому користувачі не зможуть застосовувати свої yaml файли з описом бажаного стану Network об'єкту.

Після розробки схеми для ресурсу наступною не менш важливою частиною є реалізація операторів – сервісів, що моніторять стан ресурсів та гарантують відповідність стану ресурсів у Kubernetes їхньому стану в зовнішній системі.

Crossplane надає готовий інтерфейс, який розробник повинен самостійно реалізувати відповідно до своїх потреб і вимог. Йому потрібно реалізувати 4 основних метода – observe, create, update, delete.

Метод Observe відповідає за перевірку поточного стану керованого ресурсу в зовнішній системі та синхронізацію цього стану з Kubernetes. Цей метод використовується для визначення, чи є розбіжності між бажаним станом ресурсу, зазначеним у Kubernetes, і фактичним станом ресурсу в зовнішній системі. Тобто реалізація цього метода повинна реалізовувати такі задачі:

- виконання запиту до зовнішньої системи для отримання інформації про стан ресурсу;
- аналіз отриманої інформації та порівняння з бажаним станом ресурсу, зазначеному в об'єкті Kubernetes;
- у випадку різниці очікуваного і фактичного результату, викликати потрібну операцію модифікації об'єкта зі списку API постачальника;
- оновлення статусу ресурсу фактичними значеннями з зовнішньої системи.

Реалізація цього методу для Kamatera Network об'єкту містить виклики до REST інтерфейсу, що надає Kamatera постачальник, порівняння зі значеннями полів об'єкту NetworkParameters, що був описаний та повертання відповідного значення – Network існує, Network не існує, Network змінився, Network синхронізований.

Відповідно до цього результату викликається один з наступних методів, які також були реалізовані з використанням Kamatera REST API:

- Create – створює новий ресурс у зовнішній системі та оновлює статус ресурсу в Kubernetes;
- Update - оновлює ресурс у зовнішній системі, щоб він відповідав новим параметрам, зазначеним у Kubernetes;
- Delete - видаляє відповідний ресурс у зовнішній системі та виконує необхідні операції очищення.

Коли оператор з цією реалізацією розгортається на кластері, він починає виконувати 2 операції одночасно. Перша операція це прослуховування та реагування на Kubernetes події, які генеруються при додаванні, змінні та видаленні ресурсу. При появі такої події оператор викликає відповідний Observe метод, що аналізує модифікації та при необхідності викликає потрібну реалізацію зі списка CRUD методів для відповідного CRD. Друга операція це просто періодичний виклик Observe методу для всіх об'єктів, навіть якщо модифікацій проведено не було. Це зроблено для того, щоб виявляти неочікувані зміни в зовнішній системі та синхронізувати ресурс в такому випадку. З цього стає зрозуміло, що найважливішим місцем в реалізації провайдера для Crossplane є Observe метод, який як раз і виконує основну логіку.

Діаграму схеми роботи імплементованої системи для Kamatera постачальника наведено на наступному зображенні (див. рис. 5.2).

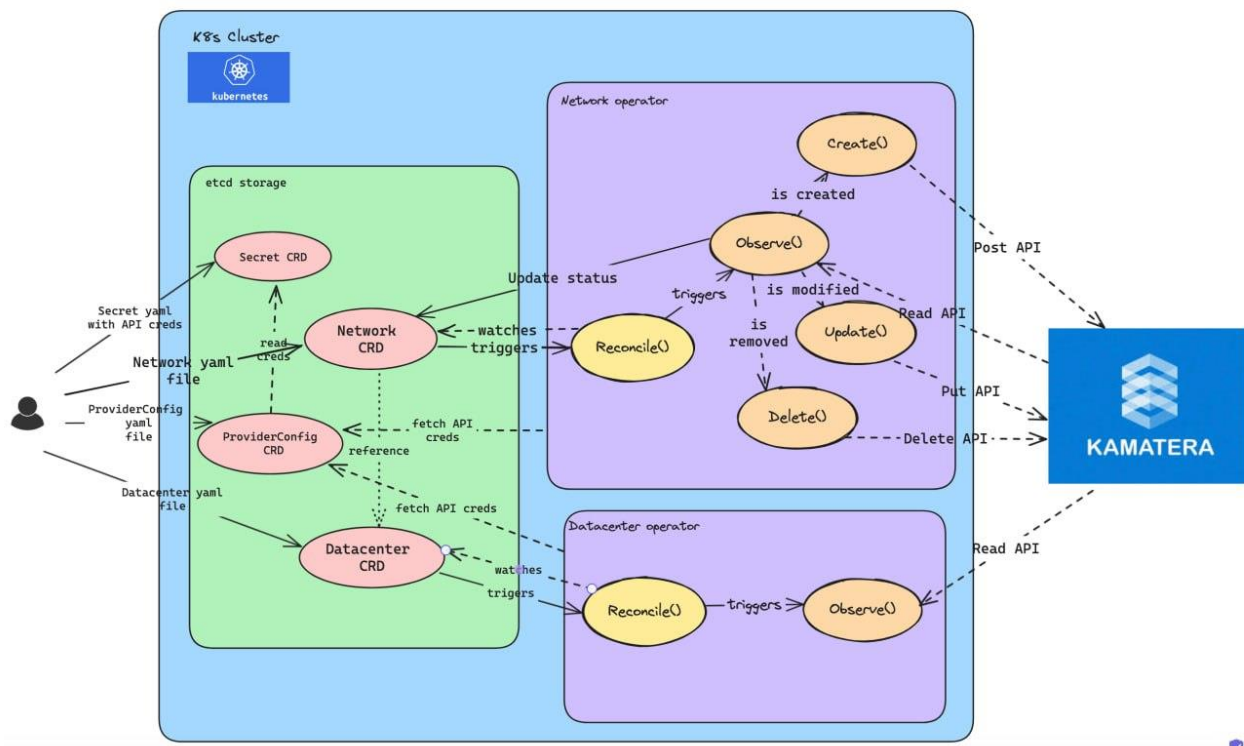



Рисунок 5.2 – Діаграма реалізованого провайдера для управління Kamatera Network (виконано самостійно)

Опишемо схему, зображену на діаграмі. Основним джерелом даних є база даних etcd. Коли до Kubernetes API надходить запит з певним CRD, він зберігає його в своєму сховищі. Паралельно з цим спеціальні вбудовані сервіси генерують подію, яка розсилається всім операторам, що прослуховують цей канал. Кожна реалізація оператора CRD в Kubernetes платформі повинна імплементувати спеціальний метод Reconcile, що приймає таку подію, та вирішує чи співпадає тип оператора з типом ресурсу, який її сгенерував або ні. В Crossplane операторах метод Reconcile вже є реалізованим розробниками за замовчуванням. Цей метод має різноманітні налаштування обробки подій, наприклад, він контролює максимальний час виконання операції обробки, інтервал перевірки відповідності ресурсу бажаному стану, інтервал до наступної спроби обробки при виникненні помилки тощо. Після виконання всіх валідацій, він викликає Observe метод, що був описаний раніше.

В поточній реалізації для створення ресурсу типу «Мережа», було створено 3 типи CRD – Network, Datacenter, KamateraProviderConfig. Network ресурс слугує для задання параметрів та характеристик, які повинні відображати параметри мережі в Kamatera провайдері. Оператор для цього ресурсу реалізує всі 4 методи – Observe, Create, Update, Delete з відповідними викликами REST API, що надає постачальник. Також Kamatera має розділення на датацентри, тому при створенні мережі потрібно вказувати, до якого саме вона буде відноситися. Для посилання на датацентр потрібно вказувати унікальний ідентифікатор. Цей ідентифікатор повинен бути одним зі списку, який надається REST API. Для того, щоб в користувача не було необхідності вручну витрачати час на пошук ідентифіктора потрібного центру, було вирішено створити нову CRD “Datacenter”. Особливістю цього CRD є ObserveOnly принцип. Він полягає в тому, що єдиним реалізованим методом в операторі є Observe без інших CRUD методів. Це дозволяє ресурсу містити в собі актуальний стан зовнішнього ресурсу, але при цьому унеможливлює для користувача робити зміни. Коротко кажучи, метод Observe для Datacenter об’єкту отримує його ідентифікатор та зберігає в статусі. Після цього Network ресурси можуть посилатися на нього для виконання власних операцій.

Крім того, перед початком виконання операцій деплойменту сховище etcd повинно мати ресурси для конфігурації зв’язку з хмарним API. Для цього потрібно створити ресурс типу “Secret”, що буде містити вхідні дані, та в ProviderConfig CRD визначити посилання на цей секрет. В майбутньому всі методи операторів будуть використовувати цей ProviderConfig, щоб отримувати облікові дані для кожного виклику інтерфейсу Kamatera.

### 5.3 Інтеграція з DevOps та GitOps

Для автоматизації GitOps процесу було використано інструмент під назвою ArgoCD. ArgoCD слідує шаблону GitOps, використовуючи репозиторії Git як джерело істини для визначення бажаного стану програми. Argo CD реалізовано як контролер Kubernetes, який безперервно відстежує запущені додатки і порівнює поточний стан з бажаним цільовим станом (як зазначено в репозиторії Git). Розгорнутий додаток, поточний стан якого відхиляється від цільового, вважається OutOfSync [28].

Інтеграція ArgoCD в Kubernetes кластер, позбавляє користувачів від мануальної роботи деплоя маніфестів, що описують інфраструктурні ресурси. Це дозволяє розробникам напряму додавати необхідні yaml файли конфігурацій в GitHub репозиторій, в той час як ArgoCD контролер на кластері в автоматичному режимі моніторить зміни в репозиторії, стягує та застосовує їх до Kubernetes API.

### 5.4 Приклад розгортання за допомогою розробленого рішення

Перед початком розгортання, потрібно додати створені нами Custom Resource Definitions на демонстраційний Kubernetes кластер, для того щоб API кластеру знав про ці типи ресурсів і користувач міг деплоїти нові екземпляри (див. рис. 5.3).

```
vlytovch@v-lytovchenko-16-5DBMD aws % kubectl get crds | grep kamatera
datacenters.topology.kamatera.crossplane.io          2024-06-02T17:53:32Z
networks.network.kamatera.crossplane.io              2024-05-30T00:02:46Z
providerconfigs.kamatera.crossplane.io               2024-05-30T00:02:45Z
```

Рисунок 5.3 – Застосовані CRDs на кластері (виконано самостійно)

Після цього потрібно додати оператор, який буде управляти цими ресурсами та проводити необхідні операції згідно подій. На рисунку відображені логи оператора, в який можна помітити початок роботи Reconcile методів для кожного виду CRD (див. рис. 5.4).

```
2024-06-08T17:51:53+03:00	INFO	Starting EventSource	{"controller": "managed/network.network.kamatera.crossplane.io", "controllerGroup": "network.kamatera.crossplane.io", "controllerKind": "Network", "sour
ce": "kind source: *v1.Network"}
2024-06-08T17:51:53+03:00	INFO	Starting EventSource	{"controller": "managed/datacenter.topology.kamatera.crossplane.io", "controllerGroup": "topology.kamatera.crossplane.io", "controllerKind": "Datacenter
", "source": "kind source: *v1.Datacenter"}
2024-06-08T17:51:53+03:00	INFO	Starting Controller	{"controller": "managed/network.network.kamatera.crossplane.io", "controllerGroup": "network.kamatera.crossplane.io", "controllerKind": "Network"}
2024-06-08T17:51:53+03:00	INFO	Starting EventSource	{"controller": "providerconfig/providerconfig.kamatera.crossplane.io", "controllerGroup": "kamatera.crossplane.io", "controllerKind": "ProviderConfig",
"source": "kind source: *v1alpha1.ProviderConfig"}
2024-06-08T17:51:53+03:00	INFO	Starting Controller	{"controller": "managed/datacenter.topology.kamatera.crossplane.io", "controllerGroup": "topology.kamatera.crossplane.io", "controllerKind": "Datacenter
"}
2024-06-08T17:51:53+03:00	INFO	Starting EventSource	{"controller": "managed/server.core.kamatera.crossplane.io", "controllerGroup": "core.kamatera.crossplane.io", "controllerKind": "Server", "source": "ki
nd source: *v1.Server"}
2024-06-08T17:51:53+03:00	INFO	Starting Controller	{"controller": "managed/server.core.kamatera.crossplane.io", "controllerGroup": "core.kamatera.crossplane.io", "controllerKind": "Server"}
2024-06-08T17:51:53+03:00	INFO	Starting EventSource	{"controller": "providerconfig/providerconfig.kamatera.crossplane.io", "controllerGroup": "kamatera.crossplane.io", "controllerKind": "ProviderConfig",
"source": "kind source: *v1alpha1.ProviderConfigUsage"}
2024-06-08T17:51:53+03:00	INFO	Starting Controller	{"controller": "providerconfig/providerconfig.kamatera.crossplane.io", "controllerGroup": "kamatera.crossplane.io", "controllerKind": "ProviderConfig"}
2024-06-08T17:51:53+03:00	INFO	Starting workers	{"controller": "managed/datacenter.topology.kamatera.crossplane.io", "controllerGroup": "topology.kamatera.crossplane.io", "controllerKind": "Datacenter
", "worker count": 10}
2024-06-08T17:51:53+03:00	INFO	Starting workers	{"controller": "managed/network.network.kamatera.crossplane.io", "controllerGroup": "network.kamatera.crossplane.io", "controllerKind": "Network", "work
er count": 10}
2024-06-08T17:51:53+03:00	INFO	Starting workers	{"controller": "providerconfig/providerconfig.kamatera.crossplane.io", "controllerGroup": "kamatera.crossplane.io", "controllerKind": "ProviderConfig",
```

Рисунок 5.4 – Логи з Kamatera Kubernetes оператора (виконано самостійно)

Наступним кроком є встановлення ArgoCD на кластер та додавання Kamatera ресурсу, який буде управлятися кластером (див. рис 5.5).

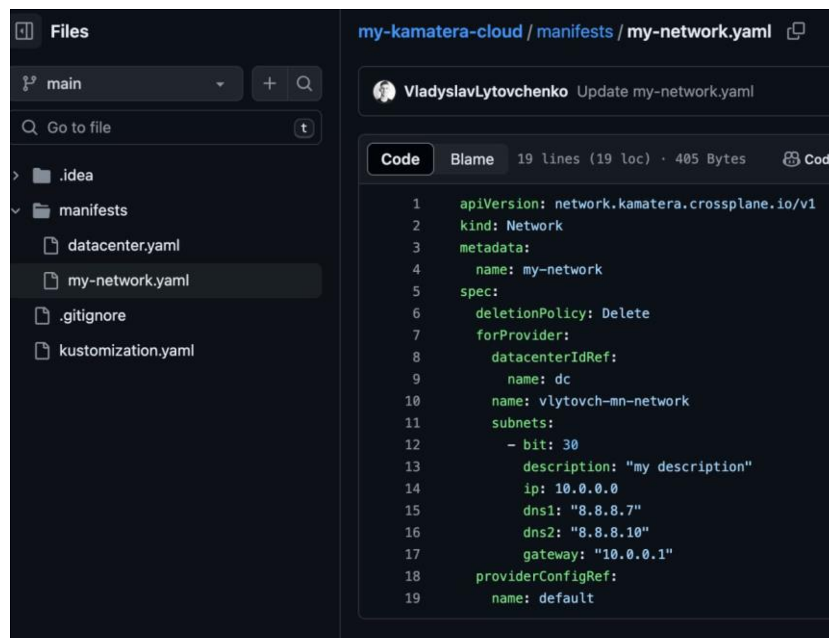



Рисунок 5.5 – Network маніфест в GitHub репозиторії (виконано самостійно)

Видно, що Network має посилання на Datacenter об’єкт, тому для успішного розгортання потрібно створити і цей ресурс, щоб наш оператор отримав ідентифікатор датацентру по назві та країні (див. рис. 5.6).

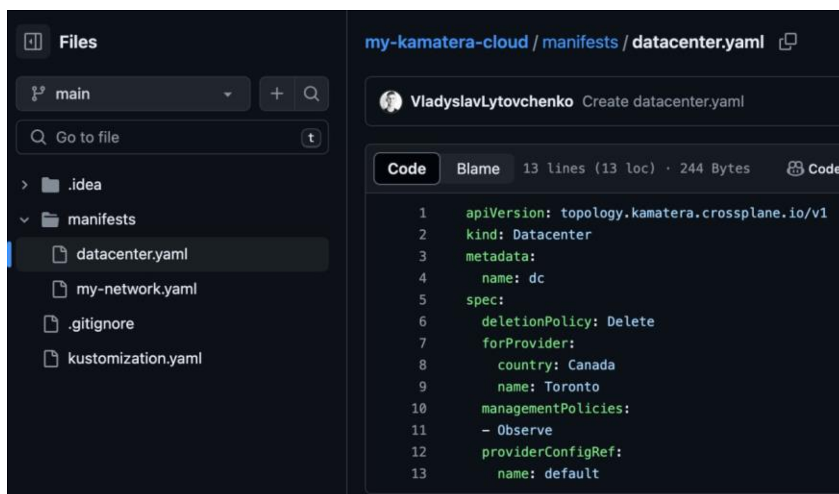



Рисунок 5.6 – Datacenter маніфест в GitHub репозиторії (виконано самостійно)

Після внесення змін в репозиторій, ArgoCD автоматично знаходить ці маніфести та починає процес створення їх на кластері. На рисунку наведений графічний інтерфейс, що дозволяє детально переглядати процес управління CRDs (див. рис. 5.7).

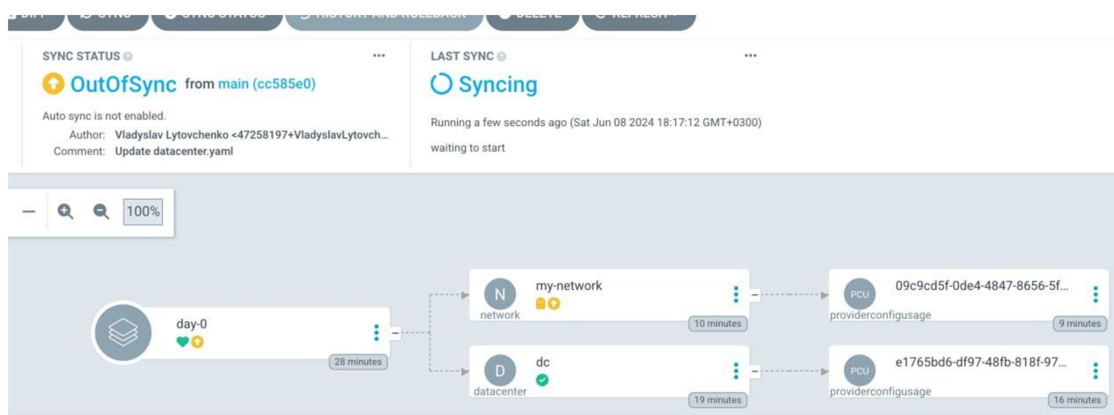
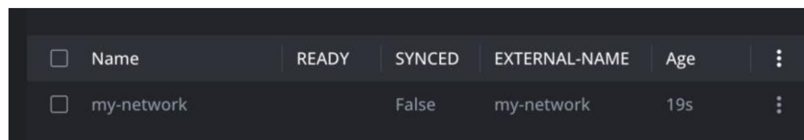



Рисунок 5.7 – Графічний інтерфейс ArgoCD (виконано самостійно)

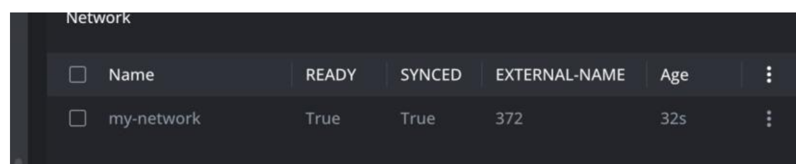
В перші секунди після створення, статус ресурсу відображає те, що бажаний стан, який визначений в маніфесті ще не застосований. За це відображення відповідають статуси Ready та Synced (див. рис. 5.8).



| Name | READY | SYNCED | EXTERNAL-NAME | Age |
|---|---|---|---|---|
| my-network | | False | my-network | 19s |

Рисунок 5.8 –Стан CRD після створення (виконано самостійно)

Після очікування протягом певного проміжку часу, статус ресурсу переходить в “здоровий” стан, що означає успішний результат останньої синхронізації (див. рис. 5.9).



Network

| Name | READY | SYNCED | EXTERNAL-NAME | Age |
|---|---|---|---|---|
| my-network | True | True | 372 | 32s |

Рисунок 5.9 – Стан ресурсу на кластері після синхронізації (виконано самостійно)

Відкривши веб консоль Kamatera постачальника, помітно, що створилась нова мережа з параметрами та характеристиками, які зазначенні в yaml маніфестах (див. рис. 5.10).

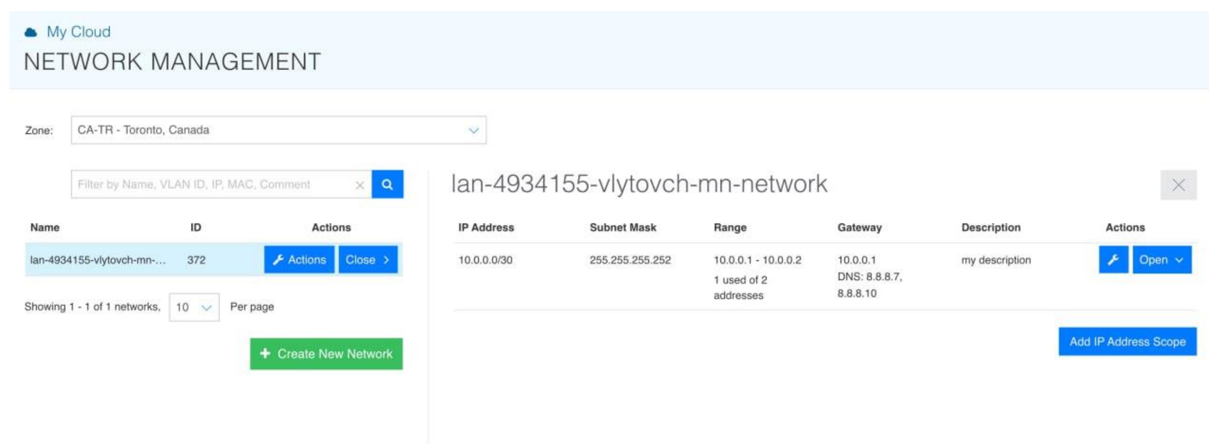



Рисунок 5.8 – Розгорнутий Network ресурс в Kamatera веб консолі (виконано самостійно)

При внесенні змін в конфігураційний файл, можна побачити, як стан ресурсу переходить в положення False, означаючи, що локальний та віддалений об’єкти не синхронізовані. Це є результатом виконання методу Observe, що провокує виклик Update методу для внесення нових модифікацій хмарну інфраструктуру. Після деякого часу синхронізація успішна і статус знову переходить в “здорове” положення (див. рис. 5.11).

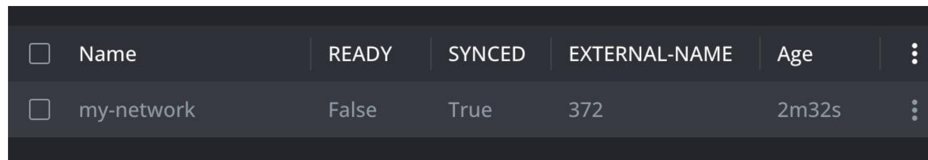



Рисунок 5.11 – Стан ресурсу на кластері після модифікації (виконано самостійно)

5.5 Висновки за проведеною реалізацію

Проведене дослідження допомогло на низькому рівні детально зрозуміти алгоритм роботи Crossplane IaC інструменту. Це дає можливість краще розуміти першопричини значних переваг цієї системи над іншими vendor-neutral системами, які були виявлені в теоретичній частині дослідження роботи.

Crossplane надає досить легкий спосіб додавання свого виду хмарної інфраструктури, тому новоутворенні хмарні постачальники можуть швидко створити власний API для управління ресурсами в своїй екосистемі на готовій платформі.

Крім того, в дослідженні було розглянуто питання, яким саме чином Crossplane отримує користь від функціональності платформи Crossplane.

Також програмна система наглядно продемонструвала, яким саме чином можна покращити процес IaC через легку інтеграцію з вбудованими на прикладі GitHub та ArgoCD.

# ВИСНОВКИ

В ході даної роботи було проаналізовано історію створення та сучасні тенденції ринку хмарних технологій. Крім цього було визначено, які принципи надання хмарних послуг надаються користувачам та з якими проблеми вони стикаються в цій сфері. Для подальшої роботи було визначено, яку роль відіграє процес створення інфраструктури для хмарних додатків та які недоліки є в ручному режимі цього процесу.

Після цього було детально розглянуто методологію Infrasturucture as a Code та її роль в вирішенні ряду проблем в автоматизації проблем розгортання хмарних ресурсів.

Було визначено основні види підходів програмування, а саме – декларативного та імперативного. Розглянуто переваги, недоліки кожного з них, та визначено їх значення в процесі опису конфігурацій для об’єктів, що складають хмарну інфраструктуру. Крім того, було детально описано множину існуючих способів та методів для автоматизації цього процесу, проведено їх порівняння та визначено найбільш перспективний принцип.

Було визначено два найкращих та найпопулярніших інструмента цього метода на ринку, що мають абсолютно різний механізм виконання процесу синхронізації та розгортання – Terraform та Crossplane. Завдяки теоретичному та практичному ряду зіставлень та дослідів було знайдено переваги та недоліки кожного з них відносно один одного. За підсумком визначено, який інструмент є кращим в даний час та в перспективі – Crossplane.

В кінці було виконано практичну реалізацію існуючого хмарного постачальника на базі Crossplane. Як піддослідного постачальника було обрано Kamatera Cloud, щоб продемонструвати реальний use-case, так як цієї хмари на час написання роботи немає в реєстрі постачальників Crossplane Upbound Marketplace. Для мінімальної робочої реалізації було обрано ресурс інфраструктури типу Network, що відображає локальну мережу в Kamatera. Для неї було імплементовано Kubernetes CRD та відповідний Kubernetes оператор. Після реалізації було детально розглянуто структуру проекту та схему роботи

розгортання, використовуючи написану нами програму. Це дає можливість визначити, яким чином цей інструмент переважає інші IaC інструменти та як саме Kubernetes платформа допомагає йому в цьому. В решті решт було продемонстровано роботу практичної реалізації на реальному прикладі створення Network ресурсу за допомогою локального Kubernetes кластеру.

В подальшому дослідження можна доповнити, ще більше поглибившись в реалізацію Crossplane та спробувати пришвидшити швидкість виконання, доповнити функціональність тощо.

# ПЕРЕЛІК ДЖЕРЕЛ ПОСИЛАННЯ


1. The Simple Guide To The History Of The Cloud. *CloudZero*. URL: https://www.cloudzero.com/blog/history-of-the-cloud/ (Дата звернення: 14.03.2024).

2. Lohr S. Google and I.B.M. Join in ‘Cloud Computing’ Research (Published 2007). *The New York Times*. URL: https://www.nytimes.com/2007/10/08/technology/08cloud.html (Дата звернення: 14.03.2024).

3. Enterprises paying too much for cloud as economic climate bites. *BetaNews*. URL: https://betanews.com/2023/06/21/enterprises-paying-too-much-for-cloud-as-economic-climate-bites/ (Дата звернення: 18.03.2024).

4. Gartner Forecasts Worldwide Public Cloud End-User Spending to Reach Nearly $600 Billion in 2023 - All About Security. *All About Security*. URL: https://www.all-about-security.de/gartner-forecasts-worldwide-public-cloud-end-user-spending-to-reach-nearly-600-billion-in-2023/ (Дата звернення: 18.03.2024).

5. Учасники проектів Вікімедіа. Хмарні обчислення – Вікіпедія. *Вікіпедія*. URL: https://uk.wikipedia.org/wiki/Хмарні_обчислення (Дата звернення: 27.03.2024).

6. Alkilani M., Kobziev V. Enhancing E-government Services by Using Cloud Computing //CEUR Workshop Proceedings. – 2019. – P.66-69.

7. Хмарні технології та обчислення. Теорія та практика. *SIM-Networks – Your Goals, our Tech. IT Infrastructure from German Provider*. URL: https://www.sim-networks.com/ukr/blog/cloud-technologies (Дата звернення: 30.03.2024).

8. Singh K. What is the difference between Public, Private and Hybrid Cloud?. *Medium*. URL: https://karansinghreen.medium.com/what-is-the-difference-between-public-private-and-hybrid-cloud-a41bba631479 (Дата звернення: 02.04.2024).

9. Лановий О., Кульмінський А. Використання даних як сервісу за допомогою хмарних технологій. БИОНИКА ИНТЕЛЛЕКТА. 2017. № 2. С. 177–182.

10. Hybrid cloud - Glossary | CSRC. *NIST Computer Security Resource Center | CSRC*. URL: https://csrc.nist.gov/glossary/term/hybrid_cloud (Дата звернення: 04.04.2024

11. Boisvert M., Bigelow S. J., Chai W. What is IaaS? Infrastructure as a Service Definition | TechTarget. *Cloud Computing*. URL: https://www.techtarget.com/searchcloudcomputing/definition/Infrastructure-as-a-Service-IaaS (Дата звернення: 05.04.2024).

12. What is Platform as a Service (PaaS) - WalkMe™ - Digital Adoption Platform. *WalkMe™ - Digital Adoption Platform*. URL: https://www.walkme.com/glossary/platform-as-a-service-paas/ (Дата звернення: 05.04.2024).

13. Aravamudhan A. SaaS vs PaaS vs IaaS: Examples, differences, & how to choose. *eG Innovations*. URL: https://www.eginnovations.com/blog/saas-vs-paas-vs-iaas-examples-differences-how-to-choose/ (Дата звернення: 08.04.2024).

14. What is Cloud Infrastructure? - Cloud Computing Infrastructure Explained - AWS. *Amazon Web Services, Inc.* URL: https://aws.amazon.com/what-is/cloud-infrastructure/?nc1=h_ls (Дата звернення: 08.04.2024).

15. NioyaTech LLC. How Does Cloud Infrastructure Work?. *LinkedIn: Log In or Sign Up*. URL: https://www.linkedin.com/pulse/how-does-cloud-infrastructure-work-nioyatech-inc/ (Дата звернення: 14.04.2024).

16. What is Cloud Storage? - Cloud Storage Explained - AWS. *Amazon Web Services, Inc.* URL: https://aws.amazon.com/what-is/cloud-storage/ (Дата звернення: 14.04.2024).

17. Mathur M. Understanding Infrastructure as Code and Configuration Management. *Medium*. URL: https://muditmathur121.medium.com/understanding-infrastructure-as-code-and-configuration-management-558f6d0659d5 (Дата звернення: 18.04.2024).

18. The Role of DevOps in Enhancing the Software Development Life Cycle. *Workplace Management Platforms*.

URL: https://workplacemanagementplatforms.com/the-role-of-devops-in-enhancing-the-software-development-life-cycle/ (Дата звернення: 23.04.2024).

19. What is declarative?. *Data Glossary*. URL: https://glossary.airbyte.com/term/declarative/ (Дата звернення: 27.04.2024).

20. What is AWS CloudFormation? - AWS CloudFormation. URL: https://docs.aws.amazon.com/AWSCloudFormation/latest/UserGuide/Welcome.html (Дата звернення: 28.04.2024).

21. What is the AWS CDK? - AWS Cloud Development Kit (AWS CDK) v2. URL: https://docs.aws.amazon.com/cdk/v2/guide/home.html (Дата звернення: 03.05.2024).

22. Compare AWS Command Line Interface, AWS Cloud Development Kit (AWS CDK), Hashicorp Terraform, and AWS Management Console. URL: https://www.g2.com/compare/aws-command-line-interface-vs-aws-cloud-development-kit-aws-cdk-vs-hashicorp-terraform-vs-aws-management-console (Дата звернення: 07.05.2024).

23. Compare 18 Orchestration Tools: 5,000+ Reviews & Features. *AIMultiple: High Tech Use Cases & Tools to Grow Your Business*. URL: https://research.aimultiple.com/orchestration-tools/ (Дата звернення: 15.05.2024).

24. What Is Kubernetes? | IBM. *IBM - United States*. URL: https://www.ibm.com/topics/kubernetes (Дата звернення: 19.05.2024).

25. Andersen T. What is Operator SDK?. *Medium*. URL: https://medium.com/@ZaradarTR/what-is-operator-sdk-52fd396e90e8 (Дата звернення: 20.05.2024

26. Cope N. Crossplane vs Terraform. *The Crossplane Blog*. URL: https://blog.crossplane.io/crossplane-vs-terraform/ (Дата звернення: 20.05.2024).

27. Homepage. *Kamatera*. URL: https://www.kamatera.com/ (Дата звернення: 26.05.2024).

28. Argo CD - Declarative GitOps CD for Kubernetes. Argo CD - Declarative GitOps CD for Kubernetes. URL: https://argo-cd.readthedocs.io/en/stable/ (date of access: 21.05.2024).