МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
ОДЕСЬКИЙ ДЕРЖАВНИЙ ЕКОЛОГІЧНИЙ УНІВЕРСИТЕТ

Факультет комп’ютерних наук,
управління та адміністрування
Кафедра інформаційних технологій

МАГІСТЕРСЬКА КВАЛІФІКАЦІЙНА РОБОТА

на тему: Використання технологій контейнеризації та хмарних провайдерів для розгортання веб-додатку у кластері

Виконав студент 2 курсу групи МІС-22
спеціальності 122 Комп’ютерні науки
Степаненко Андрій Сергійович

Керівник д.т.н., професор
Казакова Надія Феліксівна

Рецензент к.т.н., начальник ІОЦ ОНЕУ
Домаскін Олег Михайлович

Одеса 2023

АНОТАЦІЯ

Тема магістерської роботи «Використання технологій контейнеризації та хмарних провайдерів для розгортання веб-додатку у кластері».

Стрімкий розвиток технологій контейнеризації та систем хмарного обчислення призвели до переходу багатьох цифрових продуктів у кластери розгорнуті у хмарах, що і зумовлює актуальність їх дослідження.

Метою роботи є аналіз концепцій контейнеризації та управління контейнеризованих додатків у Kubernetes кластері, а також концепції хмарного обчислення. Дослідивши основні концепції та інструменти, наступною частиною роботи є розгортання Kubernetes кластеру на базі хмарного провайдеру, та розгортання веб додатку на базі створеного кластеру.

Об'єкт дослідження є технології контейнеризації та хмарних обчислень, а також управління контейнеризованими додатками у кластері.

Предметом дослідження є розгортання веб-додатку у Kubernetes кластері створеному на базі AKS.

В роботі було здійснено аналіз побудови інфраструктури на базі хмарних провайдерів; проведено аналіз технології контейнеризації та управління контейнеризованими додатками у кластері; здійснено розгортання веб застосунку у Kubernetes кластері, проведено його розгортання та налаштування, а також було виявлено подальші кроки для покращення проекту.

Ключові слова: КОНТЕЙНЕРИЗАЦІЯ, ХМАРНІ ОБЧИСЛЕННЯ, TERRAFORM, KUBERNETES КЛАСТЕР, РОЗГОРТАННЯ РЕСУРСІВ.

Магістерська містить 80 сторінок, 61 малюнок, 40 посилань та 1 додаток.

SUMMARY

Theme of master’s thesis is ‘Application of containerization technologies and cloud providers for deploying web-app to a cluster’.

The rapid development of containerization technologies and cloud computing systems has led to the transition of many digital products to clusters deployed in the clouds, which forms the relevance of the research.

The purpose of this qualification work is to analyze the concepts of containerization and management of containerized applications in a Kubernetes cluster, as well as the concept of cloud computing. Having studied the main concepts and tools, the next part of the work is the deployment of a Kubernetes cluster based on a cloud provider, and the deployment of a web application based on the created cluster.

The object of research is containerization and cloud computing technologies, as well as management of containerized applications in a cluster.

The subject of the research is the deployment of a web application in a Kubernetes cluster created in the AKS.

The work analyzed the construction of infrastructure based on cloud providers; analysis of containerization technology and management of containerized applications in the cluster was carried out; the deployment of the web application in the Kubernetes cluster was carried out, its deployment and configuration were carried out, and further steps to improve the project were identified.

Key words: CONTAINERIZATION, CLOUD COMPUTING, RESOURCE PROVISIONING, TERRAFORM, KUBERNETES CLUSTER.

The master's thesis contains 80 pages, 61 figures, 40 references and 1 appendix.

ЗМІСТ

СКОРОЧЕННЯ ТА УМОВНІ ПОЗНАКИ 7
ВСТУП 8
РОЗДІЛ І АНАЛІЗ ПОБУДОВИ ІНФРАСКТУРИ НА БАЗІ ХМАРНИХ ПРОВАЙДЕРІВ 10
1.1 Поняття та характеристика хмарних обчислень 10
1.2 Моделі хмарних обчислень (хмар) та типи хмарних сервісів 13
1.3 Способи та інструменти розгортання ресурсів у хмарі 19
РОЗДІЛ ІІ АНАЛІЗ ТЕХНОГОЛІЇ КОНТЕЙНЕРИЗАЦІЇ ТА УПРАВЛІННЯ КОНТЕЙНЕРИЗОВАНИМИ ДОДАТКАМИ У КЛАСТЕРІ 26
2.1 Технологія контейнеризації 26
2.1.1 Поняття та технічна реалізація Linux контейнерів 26
2.1.2 Огляд інструментів контейнеризація (на базі Docker та Podman) 33
2.2 Технологія управління контейнерами у кластері 43
2.2.1 Поняття та моделі управління контейнерами (оркестрація та хореографія) у кластері 43
2.2.2 Поняття та загальний огляд Kubernetes як інструменту оркестрації 48
2.2.3 Основні ресурси у Kubernetes кластеру 53
РОЗДІЛ ІІІ РОЗГОРТАННЯ ВЕБ ЗАСТОСУНКУ У KUBERNETES КЛАСТЕРІ 61
3.1 Загальний огляд контейнеризованого веб застосунку 61
3.2 Створення та налаштування Kubernetes кластеру 64
3.3 Розгортання веб застосунку 71
ВИСНОВКИ 75
ПЕРЕЛІК ДЖЕРЕЛ ПОСИЛАННЯ 77
ДОДАТОК А Посилання на вихідний код проекту 81

СКОРОЧЕННЯ ТА УМОВНІ ПОЗНАКИ

ВМ	віртуальна машина
ЗУ	закон України
ОС	операційна система
ПЗ	програмне забезпечення
ЦОД	центр обробки даних
ЦП	центральний процесор
AKS	Azure Kubernetes Service
API	application programming interface (прикладний програмний інтерфейс)
AWS	Amazon Web Services
Azure	Microsoft Azure
CLI	command line interface (інтерфейс командного рядка)
GCP	Google Cloud Platform
GUI	graphical user interface (графічний інтерфейс користувача)
IaaS	infrastructure as a serive (інфраструктура як сервіс)
IaC	infrastructure as a code (інфраструктура як код)
NIS	network information service (інформаційна служба мережі)
PaaS	platform as a service (платформа як сервіс)
PID	process identification (ідентифікатор процесу)
SaaS	software as a service (ПЗ як сервіс)

ВСТУП

Постійно зростаючий попит на цифрові товари на послуги вимагає все більшої та більшої кількості обчислювальних ресурсів, що призводить до як кількісних так і якісних змін у сфері цифрових технологій. Особливо це помітно на прикладі зростаючого попиту на обчислювальну потужність з одного боку, та намагання оптимізувати використання комп'ютерних ресурсів для вирішення певних(ої) задач(і).

Ця рушійна сила призвела до появи таких технологій як віртуалізація та контейнеризація. Ці концепції та технології їх використання, що з'явилися в відповідно в 1950х та 1970х роках, стали першими компонентами вирішення вищеописаної проблеми нехватки обчислювальних ресурсів та сучасного підходу до промислової розроби ПЗ. Вони дозволили оптимізувати використання існуючого обладнання та відкрили дорогу до таких концепцій, як мікросервісна архітерктура ПЗ та можливість логічної та апаратної сегрегації обчислювальних ресурсів між користувачами.

На базі віртуалізації було зроблено наступний крок до спільного використання ресурсів – появи провайдерів хмарних ресурсів. Відтепер процес від розробки ПЗ до його розгортання був скорочений у рази, що вимагало перегляду традиційного погляду до розробки ПЗ та використання новітніх підходів. Контейнеризація також знайшла своє місце у новому хмарному середовищі, що додавала гнучкості та незалежності від обладнання. Це призвело до необхідності управління такими ізольованими блоками розкинутими між декількома ВМ – появи технологій та інструментів управління контейнерів, таких як Kubernetes.

Метою даної кваліфікаційної роботи є аналіз концепцій контейнеризації та управління контейнеризованих додатків у Kubernetes кластері, а також концепції хмарного обчислення. Дослідивши основні концепції та інструменти, наступною частиною роботи є розгортання Kubernetes кластеру на базі хмарного провайдеру, та розгортання веб додатку на базі створеного кластеру.

Структура кваліфікаційної роботи магістра складається з вступу, 3 розділів, висновків, переліку посилань на 40 найменувань, 1 додатка. Повний обсяг проекту становить 80 сторінок, містить 61 рисунок.

РОЗДІЛ І АНАЛІЗ ПОБУДОВИ ІНФРАСКТУРИ НА БАЗІ ХМАРНИХ ПРОВАЙДЕРІВ

1.1 Поняття та характеристика хмарних обчислень

З моменту початку ери хмарних обчислень у 2006 році ринок даних послуг показав монументальний зріст та відповідно до звіту підготовленого Flexer «State of the Cloud Report 2020» 90% компаній активно використовують хмарні сервіси, в той час як 10% – планують використовувати їх [14] і думаю, що динаміка найближчим часом буде залишатися позитивна. Серед основних надавачів (провайдерів) хмарних послуг можна виділити таких гігантів як AWS, Azure, та GCP, в той час є достатня і більш менших гравців: Alibaba Cloud, Oracle Cloud, IBM, Tencent Cloud, Digital Ocean та багато інших [24]. Таким чином, можна стверджувати, що хмарні обчислення та хмарні сервіси знаходяться у високому попиті та активно застосовуються при побудові цифрових рішень.

Але перш ніж перейти до визначення хмарних обчислень слід зауважити, що хмарні обчислення є реалізацією так званих парадигм мережево-центричних обчислень (network-centric computing) та мережево-центричного контенту (network-centric content). Вказані концепції визначають, що обробка даних і зберігання даних відбувається на віддалених комп'ютерних системах, доступ до яких здійснюється через всюдисущий Інтернет, а не ніж на місцевому рівні. Під контент зміст мається на увазі будь-який тип або обсяг медіа, будь то статичні або динамічні, монолітні або модульні, живі або збережені, вироблені шляхом агрегації або змішані [23, С. 4].

Якщо спробувати надати визначення хмарному обчислення, то можна навести визначення надане Національним інститутом стандартів і технології (NIST) у 2011 році, який вказав, що хмарне обчислення (cloud computing) – це модель, яка дозволяє повсюдний, зручний доступ мережевий доступ на вимогу користувача до спільного пулу обчислювальних ресурсів (зокрема, мережі, сервери, сховище,

застосунки і сервіси), що можуть бути конфігуруванні, та швидко надані з мінімальними оперативними зусиллями або взаємодії з надавачем послуг [27]. Схожий підхід адаптував й європейський регулятор, який визначив, що хмарні обчислювальні послуги – це сервіси що надають доступ до здатного до масштабування та еластичного пулу спільних (shareable) обчислювальних ресурсів [8]. Якщо перейти на національний рівень, то можна навести приклад визначення наданого у національному законодавстві України, а саме ЗУ «Про хмарні послуги», який визначає *технологію хмарних обчислень* як технологія забезпечення дистанційного доступу на вимогу користувача до хмарної інфраструктури через електронні комунікаційні мережі, а хмару (хмарну інфраструктуру) – як сукупність динамічно розподілюваних та налаштовуваних хмарних ресурсів, що можуть бути оперативно надані користувачу хмарних послуг і вивільнені через глобальну та локальні мережі передачі даних [40].

В академічній та спеціальній літературі стверджують, що хмарні обчислення є результат гібридної еволюції та інтеграції різних концепцій, таких як віртуалізація, службові обчислення, сервісні обчислення, мережеві обчислення та автоматичні обчислення. Він почався з мейнфреймів і пройшов через обчислення на міні-комп’ютерах, обчислення клієнт/сервер, розподілені обчислення, мережеві обчислення та обчислення утиліт. Це не лише технологічний прорив (технічна інтеграція), а й стрибок у бізнес-моделі (платіть стільки, скільки використовуєте, без відходів). Для користувачів хмарні обчислення приховують усі деталі ІТ. Користувачам не потрібно мати жодних знань або будь-якого контролю над технічною інфраструктурою послуг, що надаються хмарою, або навіть конфігурацією системи та географічним розташуванням послуг, що надаються. Їм потрібно лише «увімкнути перемикач» (підключитися до Інтернету), щоб скористатися послугою [19, С. 12].

Проаналізувавши наведені нормативні визначення хмарним обчисленням, можна виокремити ключові, характерні ознаки останніх, а саме:

– Доступні через мережу (глобальну або локальну);

– Ресурси надаються на вимогу без необхідності укладення попередніх домовленостей чи угод;

– Необмежений (умовно) пул ресурсів здатних до швидкого масштабування;

– Можливість надання доступу іншим користувачам до хмарних ресурсів.

Доступність через мережу означає, що користувачі мають доступ до ресурсів провайдера з будь-якої точки світу через публічно доступні інструменти (будь-то веб консоль чи API), або можуть бути обмежені певному колу осіб.

Така властивість як «ресурси надаються на вимогу без необхідності укладення попередніх домовленостей чи угод» передбачає надання доступу (зазвичай обмеженого певною квотою) до усіх ресурсів (мережі, ВМ, дисковий простір, кластери, будь-що) без безпосередньої взаємодії користувача з провайдером (його представниками чи працівниками). Користувач достатньо пройти попередню реєстрацію щоб отримати доступ до хмарних ресурсів.

Необмежений пул ресурсів здатних до *швидкого* масштабування означає, що кількість та об’єм обчислювальних ресурсів наявних у провайдера є майже безмежним та здатний задовольнити потреби як невеликим користувачів так корпоративних клієнтів. При цьому ресурси зазвичай надаються на першу вимогу користувача і без будь-яких зобов’язань щодо тривалості або мінімальної кількості обов’язкових годино-робіт з боку користувача. Також важливою особливістю є можливість збільшення чи зменшення кількості задіяних користувачем ресурсів на його вимогу та у короткий час, що дозволяє досягнути клієнтам хмарного провайдеру гнучкості та не залежати від наявної кількості ресурсів на даний момент часу (Рис. 1.1).

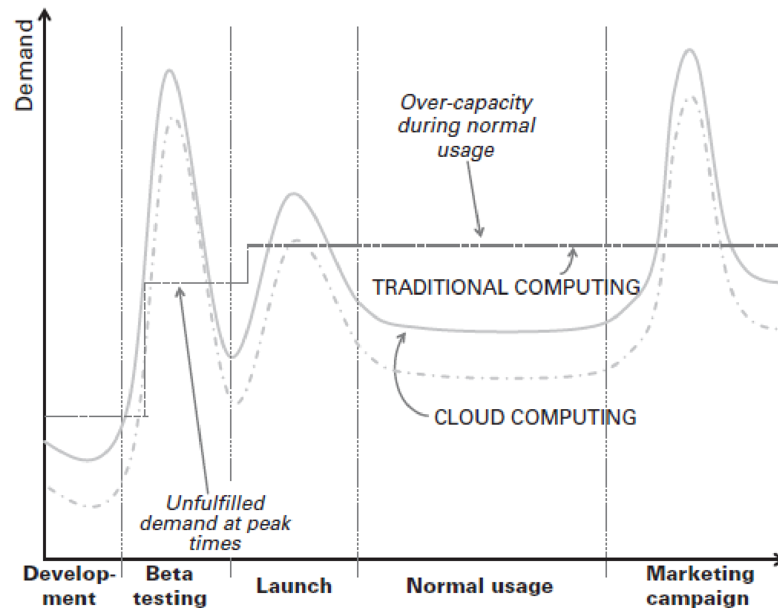


Рисунок 1.1 – Графік зміни попиту на обчислювальні ресурси та можливість адаптації хмарного провайдера

Остання характеристика визначає можливість надання доступу іншим користувачам до хмарних ресурсів та реалізувати «multi-tenant model».

Таким чином можна підсумувати, що хмарні обчисленні (сервіси хмарних обчислень) є підґрунтям реалізації концепцій мережево-центричних обчислень та мережево-центричного контенту, де під першими розуміється модель, за якої забезпечується мережевий доступ до еластичного пулу спільних обчислювальних ресурсів на вимогу користувача, що здатний до швидкого масштабування.

1.2 Моделі хмарних обчислень (хмар) та типи хмарних сервісів

Хмарні обчислення або сервіси їх надання (хмари) можуть бути класифіковані за різними критеріями, разом з тим сталим є підхід за якого їх поділяють за режимом роботи та режимом обслуговування. Перша категорія стосується того, хто володіє хмарною платформою, хто керує хмарною платформою та хто може використовувати хмарну платформу. З цієї точки зору хмари можна розділити на публічні хмари, приватні хмари (або виділені хмари) та гібридні хмари (хоча в деяких джерелах можна зустріти й таки види як хмари спільноти та

промислові хмари [19, С. 28-31]). Остання класифікація базується на моделі обслуговування хмарних обчислень, і хмару можна розділити на три рівні: IaaS, PaaS і SaaS.

Публічна хмара відноситься до категорії хмарного середовища, доступного для широкої громадськості, як правило, належить сторонньому постачальнику хмарних послуг. Його назвали загальнодоступною хмарою, оскільки він забезпечує необмежений доступ для широкої громадськості. Постачальники загальнодоступних хмарних послуг займаються встановленням, керуванням, розгортанням і обслуговуванням різноманітних IT-ресурсів, охоплюючи все: від додатків і операційних середовищ програмного забезпечення до фізичної інфраструктури. Користувачі досягають своїх цілей, використовуючи спільні IT-ресурси, і платять лише за конкретні ресурси, які вони використовують, що робить цей підхід економічно ефективним. У загальнодоступній хмарі користувачі не знають про спільне використання ресурсів, деталі реалізації основних ресурсів і не мають контролю над фізичною інфраструктурою. Отже, постачальник хмарних послуг несе відповідальність за забезпечення безпеки та надійності наданих ресурсів та виконання інших нефункціональних вимог. Хмарні послуги, які вимагають суворого дотримання стандартів безпеки та нормативних вимог, вимагають вищих і зріліших рівнів обслуговування. Приклади публічних хмар включають міжнародні платформи, як-от AWS, Azure, GCP, а також місцеві (локальні) варіанти, як-от Tencent Cloud, Alibaba Cloud, Huawei Cloud і Ucloud.

Підприємства та інші громадські організації, що працюють у закритому режимі (недоступному для широкого загалу) мають звертатися до послуг ЦОДів, які надають хмарні послуги (IT-ресурси) спеціально для таких підприємств або організацій. Такий вид послуг називають *приватними хмарами*. На відміну від публічних хмар, користувачі приватних хмар повністю володіють усім об’єктом хмарного центру. Вони мають повноваження визначати місце виконання програми та можуть керувати доступом користувачів до хмарних сервісів. Оскільки приватні

хмарні служби задовольняють потреби конкретних підприємств або організацій, вони менше зв'язані різноманітними обмеженнями, які застосовуються в публічних хмарах, наприклад обмеженнями, пов'язаними з пропускною спроможністю, безпекою та дотриманням нормативних вимог. Крім того, приватні хмари можуть запропонувати покращені гарантії безпеки та конфіденційності, використовуючи такі заходи, як контроль доступу користувачів і мережеві обмеження. Спектр послуг, що надаються приватними хмарами, також різноманітний. Вони не лише надають послуги ІТ-інфраструктури, але й підтримують різноманітні хмарні служби, включаючи операційні середовища додатків і проміжного програмного забезпечення. Приклади включають хмарні служби для внутрішніх інформаційних систем управління (IMS).

Гібридна хмара об'єднує компоненти «публічної» та «приватної хмари», дозволяючи користувачам мати часткову власність, одночасно розподіляючи ресурси в контрольований спосіб. Цей підхід дозволяє підприємствам використовувати економічні переваги загальнодоступних хмар для менш критичних програм, запускаючи їх у загальнодоступній хмарі, одночасно використовуючи внутрішні приватні хмари для основних програм, які потребують вищої безпеки та критичності.

Як зазначено в літературі, є кілька причин для того, щоб вибрати гібридну хмару. Основні причини включають пошук компромісу між різними міркуваннями та перехід від приватної хмари до публічної хмари. У першому сценарії, навіть якщо деякі організації прагнуть прийняти публічну хмару, нормативні обмеження, вимоги конфіденційності або обмеження безпеки заважають їм перенести всі свої ресурси в публічну хмару. Отже, певні ІТ-ресурси розгортаються в місці розташування бізнесу, що призводить до формування гібридної хмари [19, С. 30].

Таким чином, основними моделями хмарних обчислень (за режимом роботи) є публічні, приватні та гібридні. Публічна хмара – це хмара, доступ до якої не обмежується та є публічно доступним. Приватна хмара – це хмара, що було

створена та використовуються виключно певною організацією. Гібридна хмара – це хмара, яка представляє собою поєднання ресурсів, частина яких знаходяться у публічній, а інша – у приватній хмарах, та які поєднані між собою за допомогою технологій, що надаються хмарним провайдером [16], чи стороннім розробником (зокрема, Equinix).

Наступною класифікацію хмар, є класифікація за типах хмарних послуг, що надається надавачем або за моделлю обслуговування.

IaaS або *інфраструктура як послуга* – це хмарна послуга, що полягає у наданні користувачу хмарних послуг обчислювальних ресурсів, ресурсів зберігання або систем електронних комунікацій за допомогою технології хмарних обчислень [40]. За даної моделі споживач не керує базовою хмарною інфраструктурою та не контролює її, але має контроль над операційними системами, сховищами, розгорнутими програмами та, можливо, обмежений контроль над деякими мережевими компонентами, наприклад, міжмережевими екранами (firewalls). Послуги, які пропонує ця модель доставки, включають: серверний хостинг, веб-сервери, сховище, обчислювальне обладнання, операційні системи, ВМ, балансування навантаження, доступ до Інтернету та надання пропускної здатності.

Як зазначають дослідники, така модель доставки хмарних обчислень має низку характеристик, таких як: ресурси розподілені та підтримують динамічне масштабування, вона базується на моделі ціноутворення за комунальні послуги та змінних витратах, а апаратне забезпечення спільно використовується кількома користувачами. Ця модель хмарних обчислень особливо корисна, коли попит нестабільний, а новий бізнес потребує обчислювальних ресурсів і не хоче інвестувати в обчислювальну інфраструктуру або коли організація швидко розширюється [23, С. 28].

PaaS або *платформа як послуга* – це хмарна послуга, що полягає у наданні користувачу хмарних послуг доступу до інфраструктури та наборів комп'ютерних програм (операційних систем, системних комп'ютерних програм, програмних

засобів для комп'ютерного програмування, програмних засобів управління базами даних) за допомогою технології хмарних обчислень [40]. Платформа як послуга (PaaS) надає можливість розгортати програми, створені або придбані споживачами, використовуючи мови програмування та інструменти, які підтримуються постачальником послуг. У цій моделі користувачі звільняються від управління або контролю основної хмарної інфраструктури, що включає такі елементи, як мережі, сервери, операційні системи та сховище. Хоча користувачі не контролюють ці основні аспекти, вони зберігають контроль над розгорнутими програмами та, потенційно, конфігураціями в середовищі розміщення програм. Сервіси, які пропонуються в рамках PaaS, включають керування сеансами, інтеграцію пристроїв, пісочниці, інструменти та тестування, керування вмістом, управління знаннями та універсальний опис, виявлення та інтеграцію (UDDI).

Клієнти можуть мати певний вибір щодо механізмів зберігання даних, зокрема того, як їхні програми отримують доступ до збережених даних. Хоча вони звільнені від детального керування ресурсами, клієнти PaaS можуть бути обмежені щодо мов програмування, фреймворків, бібліотек, баз даних і служб, які вони можуть використовувати в певній службі PaaS і клієнтські програми повинні відповідати будь-яким іншим обмеженням постачальника (накладеним для масштабованості та безпеки чи з інших причин). Порівняно з IaaS клієнти також можуть мати меншу видимість того, які ресурси використовуються для запуску їхньої програми та як вони надаються (Рис. 1.2.). Платформи PaaS можуть навіть створюватися на основі чи складатися з інших хмарних служб інших постачальників, наприклад, коли певна пропозиція PaaS може використовувати середовище керування хостом (ВМ) у службі IaaS іншого постачальника [3, С. 35].

SaaS або *ПЗ як послуга* – це хмарна послуга, що полягає у наданні користувачу хмарних послуг доступу до прикладних комп'ютерних програм за допомогою технології хмарних обчислень через онлайн-сервіс або комп'ютерні програми-агенти [40]. Хмарна інфраструктура SaaS запускає лише програми, розроблені

постачальником послуг. Широкий спектр стаціонарних і мобільних пристроїв дозволяє великій кількості клієнтів отримувати доступ до послуг, які надають ці програми, використовуючи тонкий клієнтський інтерфейс, такий як веб-браузер (наприклад, веб-електронна пошта). Користувачі послуг не керують і не контролюють базову хмарну інфраструктуру, включаючи мережу, сервери, операційні системи, сховище або навіть можливості окремих програм, за винятком, можливо, обмежених налаштувань конфігурації програм, призначених для користувача.

Хоча клієнти можуть налаштовувати та встановлювати параметри для деяких додатків SaaS (наприклад, простір для зберігання чи інші обмеження на ресурси – усе це зазвичай визначає вартість), їхні можливості налаштовувати додатки зазвичай обмежені, і вони зазвичай не контролюють, як постачальники керують основними ресурсами (Рис. 1.2.). Деякі служби SaaS використовують одну запущену програму для обслуговування кількох користувачів. Знову ж таки, дані різних орендарів можуть зберігатися в одній базі даних, що створює потенційні ризики для безпеки. Клієнти повинні покладатися на програмне забезпечення SaaS для забезпечення відокремлення [3, С. 36].

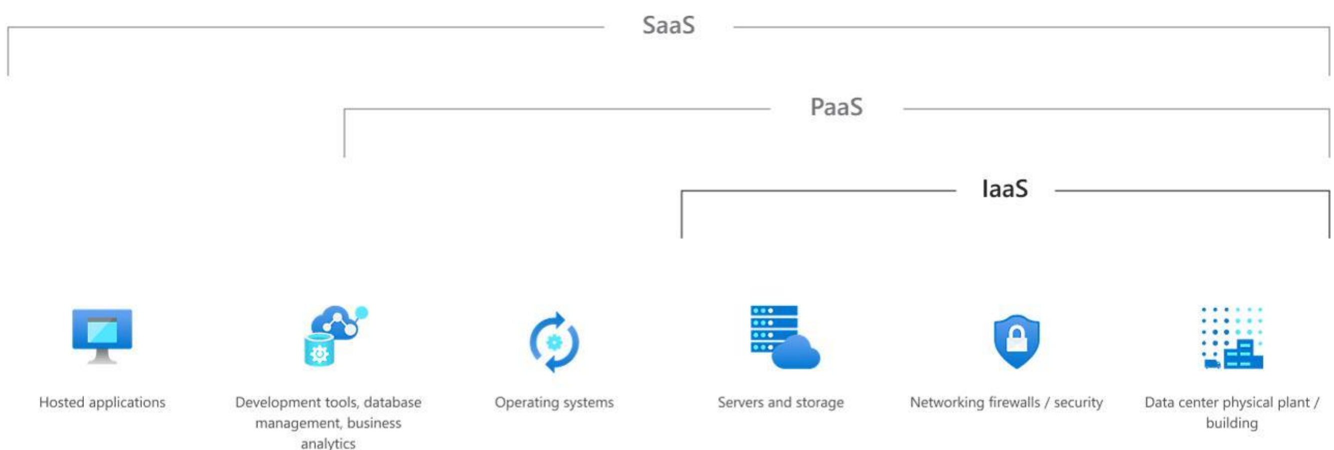


Рисунок 1.2 – Моделі надання послуг хмарними провайдерами

Таким чином, основними моделями надання послуг є інфраструктура як послуга (IaaS), платформа як послуга (PaaS), ПЗ як послуга (SaaS). Де кожна з моделей відрізняється від іншої, окрім іншого, рівнем відповідальності користувача

та контролю за наданими хмарним провайдером ресурсів, де модель IaaS надає найбільшу гнучкість та контроль, а SaaS – найменшу (Рис. 1.3).

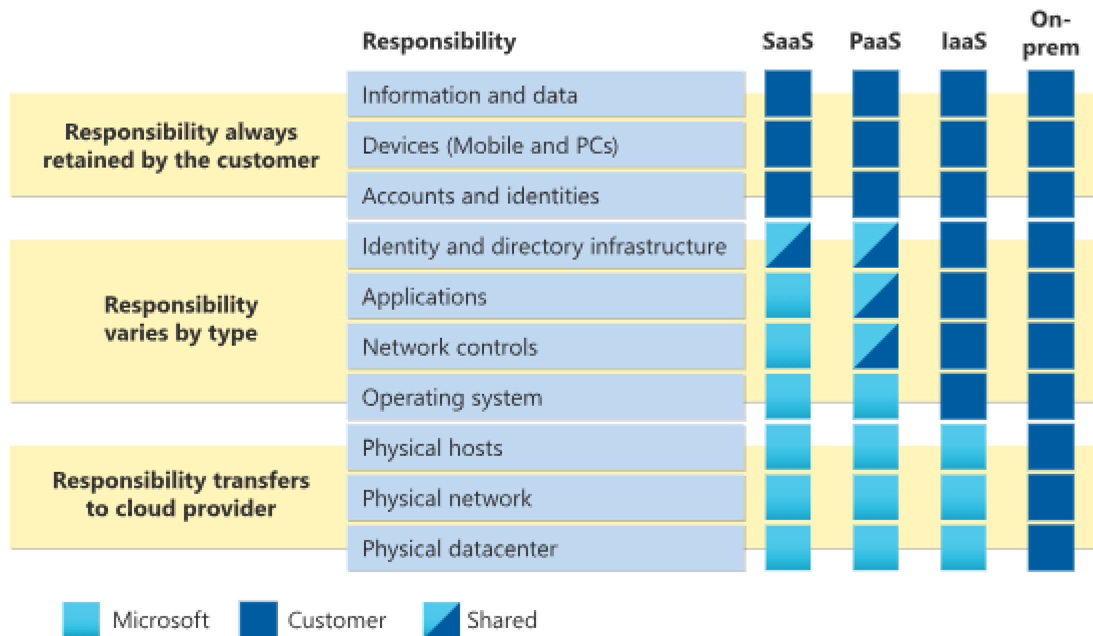


Рисунок 1.3 – Схема розподілу відповідальності користувача та хмарного провайдеру

1.3 Способи та інструменти розгортання ресурсів у хмарі

Розгортання ресурсів (resource provisioning) – це процес переведення віртуальних і фізичних ресурсів в режим онлайн. Він має як практичний компонент (зберігання та підключення пристроїв), так і компонент завантаження (налаштування того, як ресурси завантажуються в стан «готовості»).

Розгортання ресурсів відбувається, коли вперше встановлюється хмарне розгортання, тобто надається початковий набір ресурсів, але також поступово з часом додаються нові ресурси, видаляються застарілі ресурси та коли вони оновлюються. Метою розгортання ресурсів є відсутність необхідності «торкатися» таких ресурсів, що неможливо для апаратних ресурсів, оскільки воно включає ручний крок. В реальності, як зазначають автори спеціалізованої літератури, мета полягає в тому, щоб мінімізувати кількість і складність кроків конфігурації,

необхідних за межами фізичного підключення пристрою [30]. Таким чином, розгортання ресурсів полягає у споживанні ресурсів, що надаються хмарним провайдером.

Незалежно від надавача та моделі хмари користувачу доступні три способи споживання хмарних ресурсів:

1) Графічний інтерфейс веб консолі (GUI): більшість провайдерів хмарних послуг пропонують графічний веб-інтерфейс, який дозволяє користувачам взаємодіяти з ресурсами та надавати ресурси через зручну інформаційну панель. Користувачі можуть переміщатися по меню та використовувати дії «наведіть і клацніть» для розподілу та керування ресурсами (Рис. 1.4).

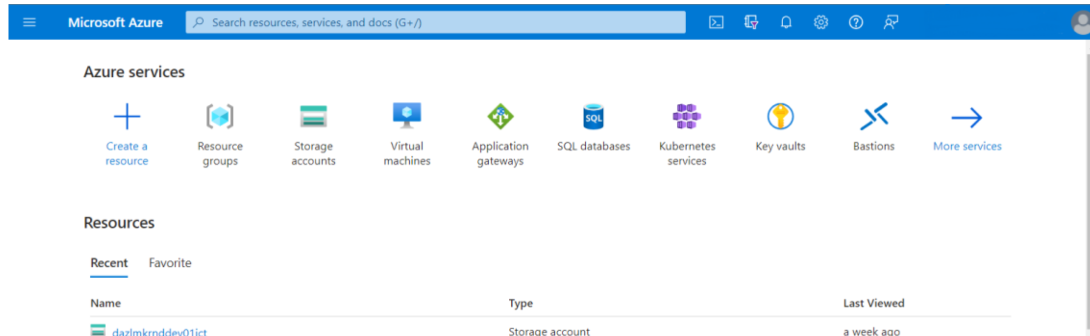


Рисунок 1.4 – Графічний інтерфейс веб консолі Azure

2) Інтерфейс командного рядка (CLI): хмарні постачальники зазвичай надають інструменти командного рядка та інтерфейси, які дозволяють користувачам взаємодіяти з хмарною платформою за допомогою текстових команд. Це корисно для автоматизації, створення сценаріїв і масового надання ресурсів (Рис. 1.5).

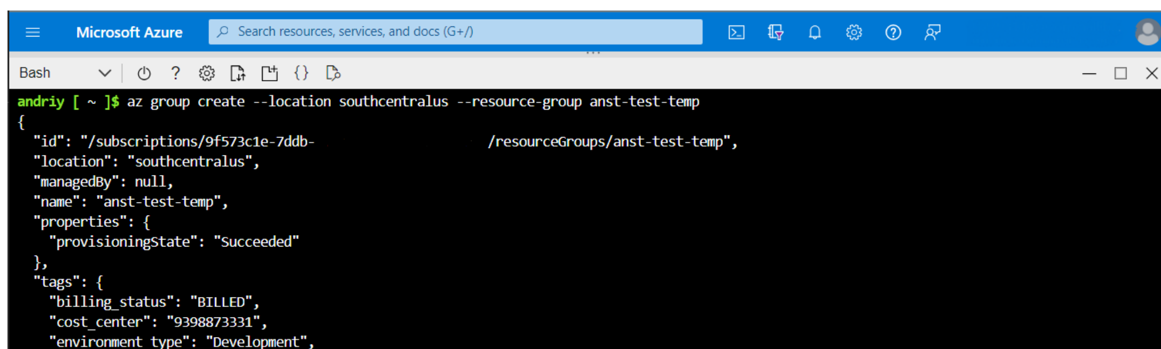


Рисунок 1.5 – Інтерфейс командного рядка az-cli

3) Прикладний інтерфейс програмування (API): хмарні постачальники надають API, які дозволяють розробникам програмно взаємодіяти з хмарною платформою. Це забезпечує високий рівень гнучкості та дає можливість індивідуальної інтеграції, автоматизації та розробки програм, які взаємодіють із хмарними службами [25].

4) Інфраструктура як код (IaC): IaC передбачає використання коду (сценаріїв або конфігураційних файлів) для визначення та надання інфраструктури. Такі інструменти, як Terraform, OpenTofu, AWS CloudFormation і Azure Resource Manager, дозволяють користувачам оголошувати бажаний стан своєї інфраструктури, полегшуючи керування версіями, автоматизацію та тиражування середовищ.

Останній підхід (IaC), де факто, можна назвати стандартним для організацій так і користувачів великої кількості хмарних ресурсів, оскільки щодня випускається все більше додатків, тому інфраструктуру потрібно часто розгортати, масштабувати та знімати. Без практики IaC керувати масштабом сучасної інфраструктури стає все важче [35], якщо не неможливо.

Такий підхід надає багато переваг, серед яких можна виділити наступні:

1) Послідовність. IaC дозволяє визначати інфраструктуру за допомогою коду, забезпечуючи узгодженість серед середовищ. Той самий код можна використовувати для надання та налаштування ресурсів у середовищах розробки, тестування та виробництва, зменшуючи ризик відхилення конфігурації;

2) Контроль версій. IaC дозволяє конфігурації інфраструктури розглядати як код, що робить його придатним для систем контролю версій (наприклад, Git). Це гарантує, що зміни в інфраструктурі можна відстежувати, скасовувати та перевіряти, сприяючи співпраці між командами розробки та операцій;

3) Відтворюваність. За допомогою IaC ви можете легко відтворювати цілі середовища, запускаючи той самий код. Це особливо корисно для сценаріїв

тестування та розробки, де узгоджене середовище має вирішальне значення для точного тестування та налагодження;

4) Автоматизація. IaC автоматизує процес надання та налаштування інфраструктури. Це зменшує мануальні помилки, підвищує ефективність і забезпечує швидке та надійне розгортання ресурсів. Автоматизація особливо важлива в динамічних і масштабованих хмарних середовищах;

5) Масштабованість. IaC полегшує горизонтальне або вертикальне масштабування інфраструктури шляхом коригування коду відповідно до мінливих вимог. Ця масштабованість є важливою в хмарних середовищах, де ресурси можна динамічно масштабувати на основі попиту;

6) Колаборація. IaC полегшує співпрацю між різними командами, такими як розробники, операційні групи та групи безпеки. Кожен може зробити свій внесок у код інфраструктури, а зміни можна переглянути та протестувати перед розгортанням.

Одним з інструментів, що дозволяє реалізувати вказаний принцип є HashiCorp Terraform. Як зазначається на офіційній веб сторінці продукту, Terraform – це інструмент інфраструктури як коду, який дозволяє визначати як хмарні, так і локальні ресурси в зрозумілих для людини конфігураційних файлах, які можна версії, повторне використання та спільний доступ. Потім ви можете використовувати послідовний робочий процес для надання та керування всією інфраструктурою протягом її життєвого циклу. Terraform може керувати компонентами низького рівня, такими як обчислення, сховище та мережеві ресурси, а також компонентами високого рівня, такими як записи DNS і функції SaaS [17].

Terraform використовує декларативний підхід до розгортання ресурсів, за якого користувач передає інструменту бажану конфігурацію, а останній вже здійснює всі необхідні дії для досягнення бажаного результату. Такий підхід дозволяє уникнути користувачу необхідності робити безпосередні зміни у хмарні ресурси, а лише внести зміни до відповідного файлу та запустити Terraform.

Робота з Terraform передбачає три послідовних кроки: 1) написання конфігураційного файлу (бажаного стану інфраструктури), 2) планування – попередній етап на якому можна переглянути, які зміни будуть застосовані інструментом задля досягнення бажаної інфраструктури, та 3) застосування – етап за якого Terraform робить безпосередні API виклики до хмарного серверу (зокрема) у необхідному порядку (у разі наявності залежностей) та вносить відповідні зміни до файлу у якому зберігається фактичний стан ресурсів (state file).

Схематично це виглядає наступним чином (Рис. 1.6).

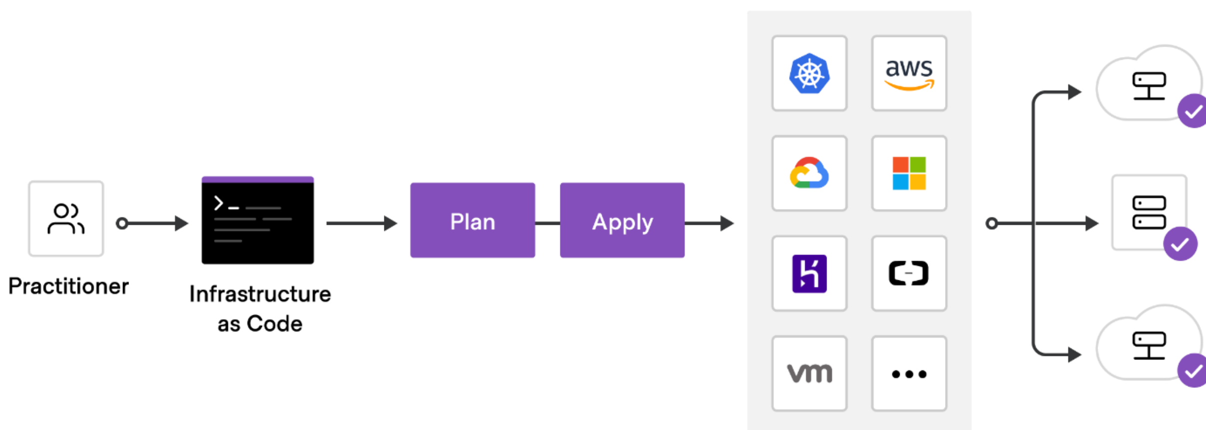


Рисунок 1.6 – Схема розгортання ресурсів Terraform-ом

Фактично робочий процес Terraform може бути представлений у наступному вигляді. За умови, що дано наступну конфігурацію:

```
terraform {
  required_providers {
    aws = {
      source  = "hashicorp/aws"
      version = "~> 4.16"
    }
  }

  required_version = ">= 1.2.0"
}

provider "aws" {
  region  = "us-west-2"
}

resource "aws_instance" "app_server" {
  ami           = "ami-830c94e3"
  instance_type = "t2.micro"
```

```
  tags = {
    Name = "ExampleAppServerInstance"
  }
}
```

Спочатку необхідно ініціалізувати Terraform у директорії: .

```
$ terraform init

Initializing the backend...

Initializing provider plugins...
- Finding hashicorp/aws versions matching "~> 4.16"...
- Installing hashicorp/aws v4.17.0...
- Installed hashicorp/aws v4.17.0 (signed by HashiCorp)

Terraform has created a lock file .terraform.lock.hcl to record the
provider selections it made above. Include this file in your version
control repository so that Terraform can guarantee to make the same
selections by default when you run "terraform init" in the future.

Terraform has been successfully initialized!

You may now begin working with Terraform. Try running "terraform plan"
to see any changes that are required for your infrastructure. All
Terraform commands should now work.

If you ever set or change modules or backend configuration for
Terraform, rerun this command to reinitialize your working directory.
If you forget, other commands will detect it and remind you to do so
if necessary.
```

Далі перевіряємо заплановані інструментом зміни:

```
$ terraform apply

Terraform used the selected providers to generate the following
execution plan.
Resource actions are indicated with the following symbols:
  + create

Terraform will perform the following actions:

  # aws_instance.app_server will be created
  + resource "aws_instance" "app_server" {
      + ami                          = "ami-830c94e3"
      + arn                          = (known after apply)
#...

Plan: 1 to add, 0 to change, 0 to destroy.

Do you want to perform these actions?
  Terraform will perform the actions described above.
  Only 'yes' will be accepted to approve.
```

Після перегляду та верифікації можна надати команду інструменту на виконання плану:

```
  Enter a value: yes

aws_instance.app_server: Creating...
aws_instance.app_server: Still creating... [10s elapsed]
aws_instance.app_server: Still creating... [20s elapsed]
aws_instance.app_server: Still creating... [30s elapsed]
aws_instance.app_server:  Creation  complete  after  36s  [id=i-01e03375ba238b384]

Apply complete! Resources: 1 added, 0 changed, 0 destroyed.
```

Таким чином, розгортання ресурсів – це процес підняття (споживання) ресурсів, що надаються, зокрема, хмарним провайдером з боку користувача. Такий процес може здійснюватися різними методами: 1) через графічний інтерфейс веб консолі, 2) інтерфейс командного рядка, 3) прикладний інтерфейс програмування та 4) застосовуючи інструменти IaC. Де останній спосіб надає багато переваг як послідовність, відтворюваність, контроль версій, автоматизація, масштабованість та колаборація. Одним з таких інструментів є Terraform, який дозволяє декларативно розгортати хмарні ресурси та здійснювати подальший їх контроль.

РОЗДІЛ ІІ АНАЛІЗ ТЕХНОГОЛІЇ КОНТЕЙНЕРИЗАЦІЇ ТА УПРАВЛІННЯ КОНТЕЙНЕРИЗОВАНИМИ ДОДАТКАМИ У КЛАСТЕРІ

2.1 Технологія контейнеризації

2.1.1 Поняття та технічна реалізація Linux контейнерів

Досліджуючи спеціалізовану літературу та ресурси присвячених технології контейнеризації на предмет поняття контейнеру можна вказати, що існує загальне, уніфіковане визначення, де під контейнером розуміється легкий, окремий виконуваний пакет, який містить усе необхідне для запуску частини програмного забезпечення, включаючи код, середовище виконання, бібліотеки та системні інструменти [12; 34]. Трохи інше визначення наведене у офіційній документації AWS, а саме як певна одиниця ПЗ огорнута (запакована) разом з залежностями для надійної та узгодженої роботи в кількох обчислювальних середовищах [1]. Тобто фактично мова йде про деякий ізольований пакет ПЗ, що містить усі необхідні компоненти для його роботи незалежно (ізольовано) від середовища.

Застосування контейнерів є сучасним підходом до розробки та розгортання ПЗ, в тому числі у хмарних середовищах, та надає низку переваг, а саме:

1) Ефективність використання ресурсів: контейнери мають невелику вагу та використовують ядро хост-операційної системи, що забезпечує ефективне використання ресурсів;

2) Портативність: контейнери інкапсулюють програми та залежності, забезпечуючи послідовність і переносимість у різних середовищах;

3) Узгодженість у різних середовищах: контейнери забезпечують узгодженість середовищ розробки, тестування та виробництва, зменшуючи проблеми, пов’язані з розбіжностями середовища;

4) Швидке розгортання: контейнери забезпечують швидкий час запуску та завершення роботи, сприяючи швидкому розгортанню та масштабуванню програм;

5) Ізоляція та безпека: контейнери забезпечують ізоляцію програм, підвищуючи безпеку за рахунок зменшення впливу вразливостей;

6) Архітектура мікросервісів: контейнери підтримують архітектуру мікросервісів, що дозволяє розробляти та розгортати модульні незалежні сервіси;

7) Інтеграція DevOps та CI/CD: контейнери добре інтегруються з практиками DevOps, забезпечуючи постійну інтеграцію та безперервне розгортання (CI/CD);

8) Масштабованість: контейнери можна легко масштабувати по горизонталі, щоб відповідати різним рівням попиту;

9) Оптимізація ресурсів: контейнери оптимізують використання ресурсів, упаковуючи лише необхідні компоненти, що сприяє швидшому запуску;

10) Контроль версій і відкат: можна задати версію образу контейнера, полегшуючи легкий відкат до попередніх версій, коли це необхідно.

Дізнавшись про переваги та місце технології у сучасному IT середовищі можна перейти до дослідження технічної реалізації даної технології.

Технологія контейнеризації вперше була запропонована IBM у 1979 році [39]. Реалізована в операційній системі UNIX V7 і запровадила системний виклик chroot. Цей прогрес був початком процесу ізоляції, запуску ізольованих груп на одному хості [2, С. 1145].

Контейнери базуються на багатій історії технологій, призначених для ізоляції однієї комп’ютерної програми від іншої, дозволяючи багатьом програмам спільно використовувати той самий ЦП, пам’ять, сховище та мережеві ресурси. Контейнери використовують фундаментальні можливості ядра Linux, зокрема простори імен, які створюють окремі перегляди ідентифікаторів процесів, користувачів, файлової системи та мережевих інтерфейсів. Середа виконання контейнерів (container runtimes) використовують кілька типів просторів імен, щоб надати кожному контейнеру ізольоване уявлення про систему [18].

Таким чином контейнери покладаються на такі можливості ядра Linux як 1) простори імен (namespaces), та 2) контрольні групи (control groups або cgroups).

Простір імен (namespace) – це особливість ядра Linux, яка забезпечує ізоляцію та віртуалізацію системних ресурсів та процесів. Кожен простір імен інкапсулює певний набір системних ресурсів, завдяки чому вони виглядають так, ніби вони призначені для окремого процесу або групи процесів. Простір імен загортає глобальний системний ресурс в абстракцію, завдяки якій процеси в просторі імен виглядають як у них є власний ізольований екземпляр глобального ресурсу. Зміни глобального ресурсу видимі для інших процесів, які є членами простору імен, але невидимі для інших процесів [22].

Основними типами просторів імен є:

1) *PID простір імен*, який ізолює простір ідентифікаційних номерів процесу, надаючи кожному процесу власний погляд на процеси системи.

2) *Mount простір імен*, який ізолює точки монтування файлової системи, дозволяючи процесам у різних просторах імен монтування мати різні перегляди ієрархії файлової системи.

3) *Network простір імен*, який забезпечує ізоляцію мережевих ресурсів, таких як мережеві інтерфейси, IP-адреси та таблиці маршрутизації.

4) *UTS простір імен*, який ізолює ім'я хоста та ім'я домену NIS, дозволяючи кожному простору імен мати власне ім'я хоста та ім'я домену.

5) *IPC простір імен*, який ізолює ресурси зв'язку між процесами, включаючи черги повідомлень і семафори.

6) *User простір імен*, який забезпечує ізоляцію для ідентифікаторів користувачів і груп, дозволяючи процесу в просторі імен користувачів мати інші ідентифікатори користувачів і груп, ніж відповідний процес у батьківському просторі імен.

Так, якщо запустити певний контейнер на Linux машині, використовуючи, наприклад, containerd [5] як середу виконання контейнерів (Рис. 2.1), то ми можемо побачити усі перераховані (окрім простору імен користувача) простори імен (Рис. 2.2).

```
root@ubuntu-focal:~# ctr run -t --rm docker.io/library/busybox:latest v1
/ #
/ # ps aux
PID   USER     TIME  COMMAND
    1 root      0:00 sh
    6 root      0:00 ps aux
/ #
```

Рисунок 2.1 – Запуск «busybox» контейнеру за допомогою containerd

```
root@ubuntu-focal:~# ctr task ls
TASK    PID      STATUS
v1      10011    RUNNING
root@ubuntu-focal:~# ps -ef | grep 10011 | grep -v grep
root       10011    9986  0 17:11 pts/0    00:00:00 sh
root@ubuntu-focal:~# ps -ef | grep 9986 | grep -v grep
root        9986       1  0 17:11 ?        00:00:00 /usr/bin/containerd-shim-runc-v2
t -id v1 -address /run/containerd/containerd.sock
root       10011    9986  0 17:11 pts/0    00:00:00 sh
root@ubuntu-focal:~# lsns | grep 10011
4026532194 mnt         1 10011 root            sh
4026532195 uts         1 10011 root            sh
4026532196 ipc         1 10011 root            sh
4026532197 pid         1 10011 root            sh
4026532199 net         1 10011 root            sh
```

Рисунок 2.2 – Ідентифікація процесу контейнера та батьківського процесу

При чому, створення вказаних просторів імен не є якоюсь примхою, чи особливістю роботи containerd, ті самі простори ми можемо спостерігати, у випадки роботи CRI-O [7] (Рис. 2.3 та 2.4).

```
root@ubuntu-focal:~# CONTAINER_ID=$(crictl create $POD_ID cont.yaml pod.yaml)
root@ubuntu-focal:~# crictl start $CONTAINER_ID
4e52a8293182a3808ae33e0f3d6c1de93be3ebf2c2f0bac990c3f39f3358e4cf
root@ubuntu-focal:~# crictl ps
CONTAINER           IMAGE                              CREATED             STATE               NAME
             ATTEMPT             POD ID
4e52a8293182a       docker.io/library/busybox:latest   16 seconds ago      Running             busybox
             0                   b3ac99419a55f
root@ubuntu-focal:~# crictl exec -ti $CONTAINER_ID /bin/sh
/ # ps aux
PID   USER     TIME  COMMAND
    1 root      0:00 /pause
   16 root      0:00 /bin/sleep 36000
   22 root      0:00 /bin/sh
   27 root      0:00 ps aux
/ # exit
root@ubuntu-focal:~# echo $CONTAINER_ID
4e52a8293182a3808ae33e0f3d6c1de93be3ebf2c2f0bac990c3f39f3358e4cf
root@ubuntu-focal:~#
```

Рисунок 2.3 – Запуск «busybox» контейнеру за допомогою CRI-O CLI

```
root@ubuntu-focal:~# PID=$(crictl inspect $CONTAINER_ID | jq '.info.pid')
root@ubuntu-focal:~# ps -ef | grep $PID | grep -v grep
root       13219   13201  0 18:14 ?        00:00:00 /bin/sleep 36000
root@ubuntu-focal:~# ps -ef | grep 13201 | grep -v grep
root       13201       1  0 18:14 ?        00:00:00 /usr/bin/conmon -b /run/containers/storage/overlay
-containers/4e52a8293182a3808ae33e0f3d6c1de93be3ebf2c2f0bac990c3f39f3358e4cf/userdata -c 4e52a8293182a
3808ae33e0f3d6c1de93be3ebf2c2f0bac990c3f39f3358e4cf --exit-dir /var/run/crio/exits -l /var/log/crio/po
ds/b3ac99419a55f488b615c0d3e23111da0f12f1d17b0c37913564d2ead0fb974f/4e52a8293182a3808ae33e0f3d6c1de93b
e3ebf2c2f0bac990c3f39f3358e4cf.log --log-level info -n k8s_busybox_busybox_crio__0 -P /run/containers/
storage/overlay-containers/4e52a8293182a3808ae33e0f3d6c1de93be3ebf2c2f0bac990c3f39f3358e4cf/userdata/c
onmon-pidfile -p /run/containers/storage/overlay-containers/4e52a8293182a3808ae33e0f3d6c1de93be3ebf2c2
f0bac990c3f39f3358e4cf/userdata/pidfile --persist-dir /var/lib/containers/storage/overlay-containers/4
e52a8293182a3808ae33e0f3d6c1de93be3ebf2c2f0bac990c3f39f3358e4cf/userdata -r /usr/lib/cri-o-runc/sbin/r
unc --runtime-arg --root=/run/runc --socket-dir-path /var/run/crio -u 4e52a8293182a3808ae33e0f3d6c1de9
3be3ebf2c2f0bac990c3f39f3358e4cf -s
root       13219   13201  0 18:14 ?        00:00:00 /bin/sleep 36000
root@ubuntu-focal:~# lsns
        NS TYPE   NPROCS   PID USER             COMMAND
4026532194 uts         2 12385 root             /pause
4026532195 ipc         2 12385 root             /pause
4026532196 mnt         1 12385 root             /pause
4026532197 pid         2 12385 root             /pause
4026532198 mnt         1 13219 root             /bin/sleep 36000
```

Рисунок 2.4 – Ідентифікація процесу контейнера та батьківського процесу. Демонстрація створених просторів імен (в т.ч. й CRI-O)

Як було продемонстровано вище, у випадку перегляду запущених процесів зсередини контейнерів за допомогою утиліти ps можна було спостерігати наявність лише процесу з PID 1 та інших процесів, шо було запущені вручну нами (bash та ps) та, ті що були запущені одразу після старту контейнеру середою виконання (pause та sleep). Разом з тим, оскільки при створенні контейнера використовуються вбудовані у ядро Linux можливості, то є змога здійснити такі самі маніпуляції вручну. Так можна використати утиліту unshare для створення нового процесу із власним простором імен PID. Зокрема, в даному прикладі, окрім створення нового процесу із власним простором імен PID здійснюється монтування нової файлову системи /proc у цьому просторі імен, а потім відбувається запуск нової оболонки (/bin/bash) в ізольованому середовищі. В даному випадку, оскільки оболонка bash була запущена в окремому просторі імен, то їй невидимі інші процеси хостової машини (Рис. 2.5).

```
root@ubuntu-focal:~# unshare -f -p --mount-proc -- /bin/sh -c /bin/bash
root@ubuntu-focal:~# ps -ef
UID          PID    PPID  C STIME TTY          TIME CMD
root           1       0  0 19:19 pts/1    00:00:00 /bin/sh -c /bin/bash
root           2       1  0 19:19 pts/1    00:00:00 /bin/bash
root           9       2  0 19:19 pts/1    00:00:00 ps -ef
root@ubuntu-focal:~#
```

Рис. 2.5 – Створення нового процесу із власним простором імен PID

Ми можемо пересвідчитись в наявності окремого простору імен PID дізнавшись PID батьківского процесу та здійснити пошук цього простору за PID (Рис. 2.6).

```
root@ubuntu-focal:~# ls -l /proc/self/ns/pid
lrwxrwxrwx 1 root root 0 Nov 30 19:20 /proc/self/ns/pid -> 'pid:[4026532200]'
root@ubuntu-focal:~# lsns | grep 4026532200
4026532200 pid         4   1 root /bin/sh -c /bin/bash
root@ubuntu-focal:~# lsns
        NS TYPE   NPROCS PID USER COMMAND
4026531835 cgroup      3   1 root /bin/sh -c /bin/bash
4026531837 user        3   1 root /bin/sh -c /bin/bash
4026531838 uts         3   1 root /bin/sh -c /bin/bash
4026531839 ipc         3   1 root /bin/sh -c /bin/bash
4026531992 net         3   1 root /bin/sh -c /bin/bash
4026532199 mnt         3   1 root /bin/sh -c /bin/bash
4026532200 pid         3   1 root /bin/sh -c /bin/bash
```

Рис. 2.6 – Ідентифікація PID процесу та відповідного простору імен

Наступним необхідним елементом для правильної роботи контейнеру, та, що важливіше певної групи контейнерів на ВМ, є розподіл обмежених обчислювальних ресурсів відповідно до певних вимог та обмежень. Ця потреба покривається шляхом застосування cgroups.

Якщо звернутися до офіційної документації Linux, то групи керування (cgroups) визначаються, як певна функція ядра Linux, яка дозволяє керувати системними ресурсами та розподіляти ці ресурси між різними процесами або групами процесів. Cgroups надають спосіб контролювати та обмежувати використання ресурсів (таких як ЦП, пам’ять і введення/виведення) процесів, забезпечуючи краще керування ресурсами, ізоляцію та оптимізацію продуктивності [36; 37].

Створення та конфігурація cgroups виконується за допомогою спеціальної файлової системи, подібно до того, як Linux повідомляє інформацію про систему через файлову систему /proc. За замовчуванням файлова система для контрольних груп знаходиться в /sys/fs/cgroup, де кожна з директорій є типом ресурсу споживання якого можна обмежити, наприклад, ЦП (Рис. 2.7).

```
root@ubuntu-focal:~# cd /sys/fs/cgroup/cpu
root@ubuntu-focal:/sys/fs/cgroup/cpu# ls -F
cgroup.clone_children  cpuacct.usage              init.scope/
cgroup.procs           cpuacct.usage_all          notify_on_release
cgroup.sane_behavior   cpuacct.usage_percpu       release_agent
cpu.cfs_period_us      cpuacct.usage_percpu_sys   system.slice/
cpu.cfs_quota_us       cpuacct.usage_percpu_user  tasks
cpu.shares             cpuacct.usage_sys          user.slice/
cpu.stat               cpuacct.usage_user
cpuacct.stat           default/
root@ubuntu-focal:/sys/fs/cgroup/cpu# _
```

Рис. 2.7. – Зміст директорії /sys/fs/cgroup/cpu

Як видно вище, у директорії /sys/fs/cgroup/cpu наявні три контрольні групи (cgropus), що представлені у виді директорій (init.scope, system.slice та user.slice), де кожна з них має свої конфігураційні файли для контролю за споживанням процесами ЦП, а саме: 1) cpu.shares , 2) cpu.cfs_period_us та 3) cpu.cfs_quota_us (Рис. 2.8.), змінюючі які можна контролювати, в даному випадку, ресурсами ЦП.

```
root@ubuntu-focal:/sys/fs/cgroup/cpu/system.slice/crio.service# ls
cgroup.clone_children  cpu.uclamp.max     cpuacct.usage_percpu_sys
cgroup.procs           cpu.uclamp.min     cpuacct.usage_percpu_user
cpu.cfs_period_us      cpuacct.stat       cpuacct.usage_sys
cpu.cfs_quota_us       cpuacct.usage      cpuacct.usage_user
cpu.shares             cpuacct.usage_all  notify_on_release
root@ubuntu-focal:/sys/fs/cgroup/cpu/system.slice/crio.service# cat cpu.cfs_quota_us
-1
root@ubuntu-focal:/sys/fs/cgroup/cpu/system.slice/crio.service# cat cpu.cfs_period_us
100000
root@ubuntu-focal:/sys/fs/cgroup/cpu/system.slice/crio.service#
```

Рис. 2.8 – Зміст конфігураційних файлів system.slice контрольної групи для CRI-O сервісу

Змінюючи ці показники можна увімкнути квоту для певного сервісу (cpu.cfs_quota_us) та визначити кількість ресурсів ЦП доступних для сервісу (у мікросекундах).

Таким саме чином досягається обмеження у споживанні процесами (а, отже і контейнерами) наявних у системі ресурсів. Тільки у випадку наявності середи виконання контейнерів (зокрема, CRI-O), користувача потрібно лише надати бажані значення у конфігураційному файлі:

```
metadata:
  name: test
image:
  image: docker.io/library/busybox:latest
linux:
  resources:
    cpu_period: 100000
    cpu_quota: 10000
```

Схожим чином можна досягти керування пам’яттю (за допомогою утиліти ulimit) та пропускною здатністю мережевого обладнання (за допомогою tc), з певними особливостями.

Таким чином, під контейнером розуміється деякий ізольований пакет ПЗ, що містить усі необхідні компоненти для його роботи незалежно (ізольовано) від середовища. Такий підхід дозволяє досягти низки переваг, таких як ефективність використання ресурсів, портативність, ізоляція та безпека, контроль версій та інше. Технологія контейнеризації побудована з використанням таких особливостей ядра Linux як просторі імен (namespaces) та контрольні групи (cgroups), які забезпечують ізоляцію контейнерів та контроль (ізоляцію) ресурсів наданих контейнеру.

2.1.2. Огляд інструментів контейнеризація (на базі Docker та Podman)

Як було продемонстровано в попередньому підрозділі роботи для створення контейнеру достатньо застосувати базові можливості ядра Linux. Разом з тим, очевидним виглядає те, що такий підхід до створення контейнерів, не кажучи, про управління ними, буде вимагати велику кількість праці та значний рівень кваліфікації користувача для досягнення бажаного результату. Саме з цієї причини було створено спеціалізоване ПЗ для автоматизації даного процесу – середа виконання контейнерів (container runtime) або двигун контейнерів (container engine). Такий container runtime забезпечує низькорівневу функціональність для запуску процесів у контейнерах та управління ними (Рис. 2.9.).

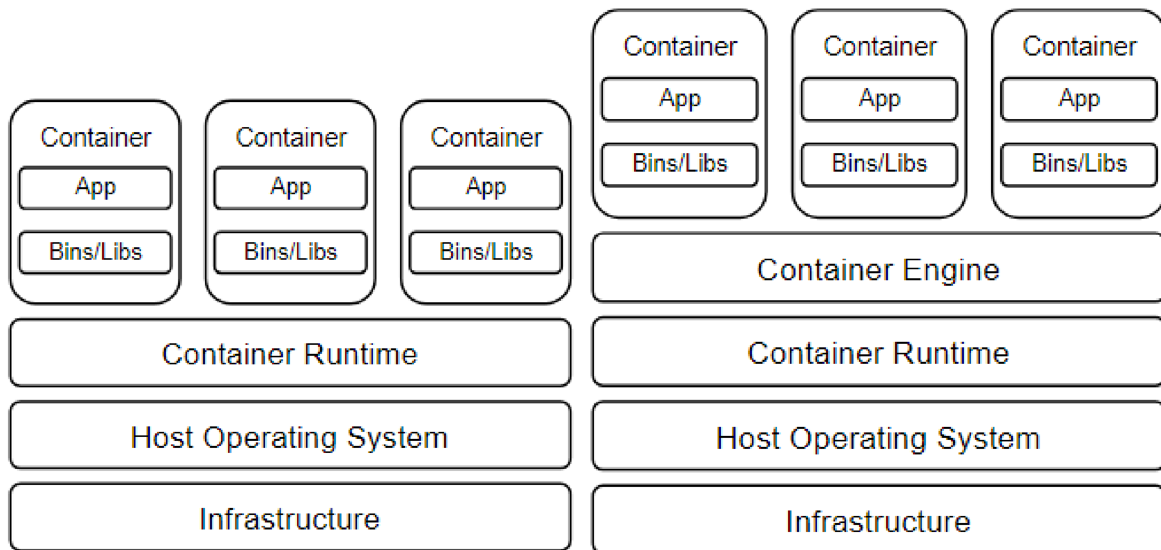


Рисунок 2.9. – Схематичне зображення контейнеру

В дійсності існує невелика різниця між container runtime та container engine, хоч і часто їх використовують як взаємозамінні поняття. Де під container runtime розуміють базове програмне забезпечення, яке відповідає за виконання контейнерів і керування ними. Він надає необхідні функції для створення, запуску та керування життєвим циклом контейнерів. Це включає в себе взаємодію з такими функціями ядра Linux, як контрольні групи та простори імен.

В свою чергу container engine є інструментом або платформою вищого рівня, яка включає не лише container runtime, але й додаткові функції для створення, пакування, розповсюдження та оркестрування контейнерів. Container engine часто включає container runtime як один із своїх компонентів, але розширює свої функціональні можливості, щоб забезпечити більш комплексне контейнерне рішення. Тому можлива і конфігурація з container engine.

Одними з найбільш комерційно успішними та широко вживаними наразі є такі container engine, як Docker та Podman. Docker отримав широке поширення та з моменту запуску (у 2013 році) [33] фактично є синонімом контейнеру, хоч ним і не є. Розробники на офіційній сторінці зазначають, що Docker – це відкрита платформа для розробки, доставки та запуску програм. Docker використовує власний container

runtime – containerd (який в свою чергу працює на базі runc) та відповідальний за взаємодію з ОС (Рис. 2.10.).

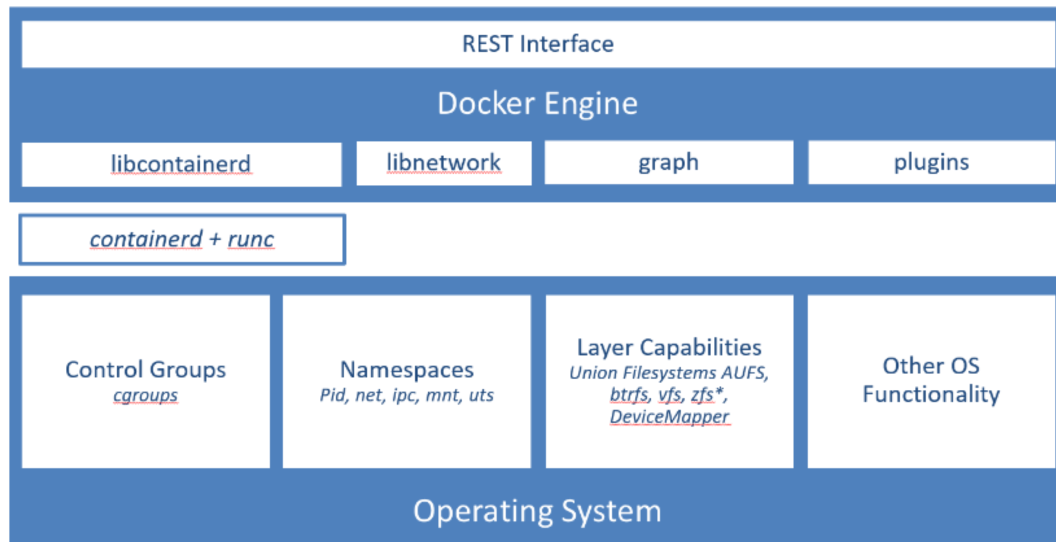


Рисунок 2.10 – Взаємодія Docker engine з хостовою Linux ОС

Docker складається з наступних елементів (Рис. 2.11):

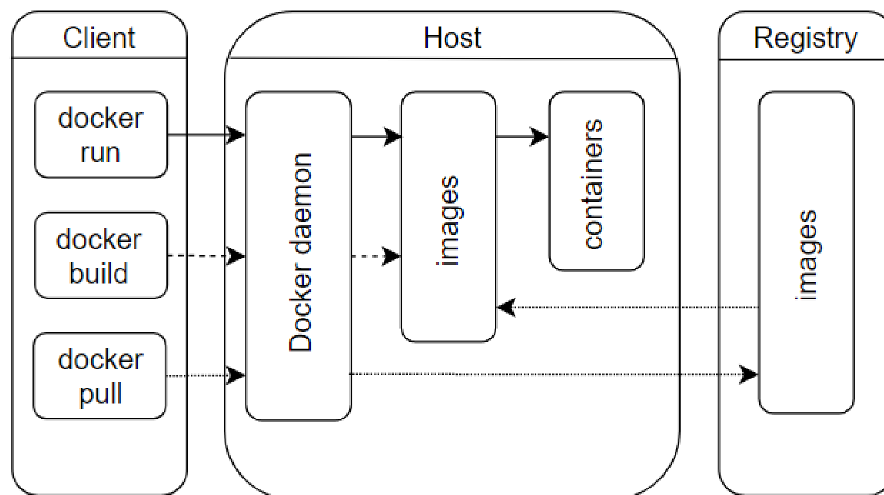


Рисунок 2.11 – Архітектура Docker

1) клієнт (Docker client), точка входу до Docker, наприклад CLI (чи графічний інтерфейс Docker Desktop), за допомогою якої відбувається спілкування з іншими компонентами системи;

2) сервер (Docker host), надає оточення для виконання програм; складається з Docker демона (dockerd), який керує Docker об’єктами: образами, контейнерами,

мережами та томами (volumes). Взаємодія Docker клієнта здійснюється за допомогою API викликів, через UNIX сокети або мережевий інтерфейс;

3) реєстр (Docker registry), місце для зберігання Docker образів.

Робота з Docker контейнерами має декілька основних етапів, які є частиною так званого управління життєвим циклом Docker контейнеру (Docker container lifecycle management) (Рис. 2.12).

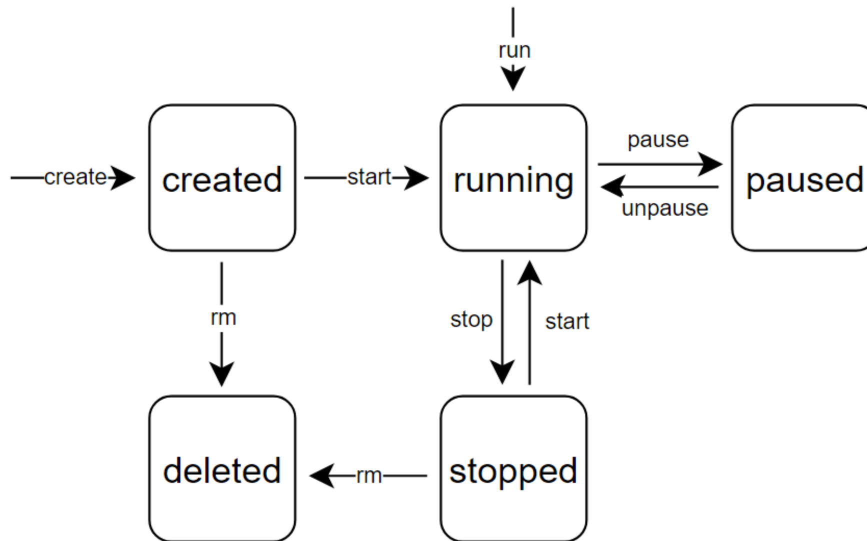


Рисунок 2.12 – Docker container lifecycle management

Відповідно до цієї моделі можна виділити наступні етапи:

1) створення контейнеру, яке відбувається на базі образу контейнеру, який можна скачати з реєстру, або створити самостійно з Dockerfile – текстового файлу, що містить усі команди необхідні для побудови (зборки) образу майбутнього контейнеру [11] (Рис. 2.13). При цьому контейнер не запускається, а знаходиться у вигляді записуваного контейнерного шар поверх вказаного образу (Рис. 2.14).

```
root@ubuntu-focal:~# docker create --name test_me test:1.0
890b7743f889819d85d152b2d7d3d4f2eb857439e9826a6cda5577f06702b9c7
root@ubuntu-focal:~# _
```

Рис. 2.13 – Створення контейнеру

```
root@ubuntu-focal:~# docker build -t test:1.0 .
Sending build context to Docker daemon  23.55kB
Step 1/3 : FROM busybox:1.36
1.36: Pulling from library/busybox
3f4d90098f5b: Pull complete
Digest: sha256:3fbc632167424a6d997e74f52b878d7cc478225cffac6bc977eedfe51c7f4e79
Status: Downloaded newer image for busybox:1.36
 ---> a416a98b71e2
Step 2/3 : ENV NAME=test
 ---> Running in 3c42b032810d
Removing intermediate container 3c42b032810d
 ---> 0a9d187a8f20
Step 3/3 : CMD ["sleep", "360"]
 ---> Running in 69ba2de326f1
Removing intermediate container 69ba2de326f1
 ---> 45d0248addfe
Successfully built 45d0248addfe
Successfully tagged test:1.0
root@ubuntu-focal:~# docker images
REPOSITORY   TAG       IMAGE ID       CREATED          SIZE
test         1.0       45d0248addfe   42 seconds ago   4.26MB
busybox      1.36      a416a98b71e2   4 months ago     4.26MB
root@ubuntu-focal:~# _
```

Рисунок 2.14 – Створення образу контейнеру з Dockerfile

2) запуск контейнеру, за якого здійснюється виконання команд, що зазначені у образі (або отримані з CLI), а Docker здійснює алокацію усіх необхідних ресурсів для цього (Рис. 2.15).

```
root@ubuntu-focal:~# docker container start test_me
test_me
root@ubuntu-focal:~# docker ps
CONTAINER ID   IMAGE      COMMAND       CREATED          STATUS         PORTS     NAMES
62c7b87f477e   test:1.0   "sleep 360"   21 seconds ago   Up 4 seconds             test_me
root@ubuntu-focal:~# docker exec -it test_me /bin/sh
/ # echo $NAME
test
/ # exit
root@ubuntu-focal:~# _
```

Рисунок 2.15 – Запуск контейнеру та перевірка наявності змінної оточення у контейнері

3) зупинення (stop) контейнеру дозволяєте «заморозити» стан контейнеру та звільнити використовувані таким контейнером ресурси. При цьому стан виконання контейнеру зберігається у пам’яті, і у випадку відновлення (Рис. 2.16) роботи продовжить з того стану, що був на момент зупинення (наприклад, у нашому випадку, продовжиться виконання команди sleep з тим часом, що залишився 360с – попередній час виконання). Зупинення (pause) контейнерів досягається за рахунок

відправлення системного сигналу SIGSTOP до контейнеру (процесу). В той же час зупинення не усіх контейнерів можливо [9].

```
root@ubuntu-focal:~# docker pause test_me
test_me
root@ubuntu-focal:~# docker ps -a
CONTAINER ID   IMAGE      COMMAND        CREATED         STATUS                   PORTS     NAMES
62c7b87f477e   test:1.0   "sleep 360"    3 minutes ago   Up 3 minutes (Paused)              test_me
root@ubuntu-focal:~# docker unpause test_me
test_me
root@ubuntu-focal:~# docker ps
CONTAINER ID   IMAGE      COMMAND        CREATED         STATUS         PORTS     NAMES
62c7b87f477e   test:1.0   "sleep 360"    3 minutes ago   Up 3 minutes             test_me
```

Рисунок 2.16 – Зупинення (pause) та відновлення роботи контейнеру

4) зупинення (stop) контейнеру дозволяє плавно завершити роботи контейнеру зі звільненням наданих контейнеру ресурсів. Контейнеру (процесу) надсилається системний виклик SIGTERM, а після спливу періоду плавного завершення (grace period) – SIGKILL, який примусово завершує його роботу. При цьому статус контейнера зберігається, що дозволяє його повторний запуск [10].

```
root@ubuntu-focal:~# docker stop test_me
test_me
root@ubuntu-focal:~# docker ps -a
CONTAINER ID   IMAGE      COMMAND        CREATED         STATUS                        PORTS     NAMES
62c7b87f477e   test:1.0   "sleep 360"    4 minutes ago   Exited (137) 14 seconds ago             test_me
```

Рисунок 2.17 – Зупинення (stop) роботи контейнеру

5) видалення зупиненого контейнеру зупиняє його виконання та звільняє утилізовані ресурси. Таке видалення можливо і щодо працюючих контейнерів, хоч і не рекомендоване.

```
root@ubuntu-focal:~# docker rm 62c7
62c7
root@ubuntu-focal:~# docker ps -a
CONTAINER ID   IMAGE     COMMAND   CREATED   STATUS    PORTS     NAMES
root@ubuntu-focal:~#
```

Рисунок 2.18 – Видалення контейнеру

Іншим інструментом (але не єдиним доступним) контейнеризації є Podman (Pod Manager). Як вказано на офіційному сайті продукту, Podman – це утиліта…, що може бути використана для створення та управління контейнерами [32]. Більше інформації можна знайти у офіційні документації проекту, де зазначено, що podman – це повнофункціональний механізм контейнерів (container engine), який є простим

інструментом та не використовує демон. Podman надає командний рядок, схожий на Docker-CLI, який полегшує перехід від інших механізмів контейнерів і дозволяє керувати модулями, контейнерами та образами [31].

Podman CLI використовує ті самі команди, що й Docker CLI, тому на рівні клієнта відмінності між цими двома інструментами майже відсутні. Користуючись Podman CLI можна досягти усіх етапів життєвого циклу контейнеру, як створення, запуск, зупинення та ін. (Рис. 2.19, 2.20 та 2.21).

```
root@ubuntu-focal:~# podman build -t test_me:1.0 .
STEP 1/3: FROM busybox:1.36
Resolved "busybox" as an alias (/etc/containers/registries.conf.d/000-shortnames.conf)
Trying to pull docker.io/library/busybox:1.36...
Getting image source signatures
Copying blob 3f4d90098f5b skipped: already exists
Copying config a416a98b71 done
Writing manifest to image destination
Storing signatures
STEP 2/3: ENV NAME=test
--> e659ba43cc9
STEP 3/3: CMD ["sleep", "360"]
COMMIT test_me:1.0
--> b8010453c5e
Successfully tagged localhost/test_me:1.0
b8010453c5e3c2ccef216b35fa5e70f58ec59dc7273682cd9a6eae5d8df83ede
root@ubuntu-focal:~# podman images
REPOSITORY                 TAG         IMAGE ID      CREATED         SIZE
localhost/test_me          1.0         b8010453c5e3  16 seconds ago  4.49 MB
docker.io/library/busybox  1.36        a416a98b71e2  4 months ago    4.5 MB
docker.io/library/busybox  latest      a416a98b71e2  4 months ago    4.5 MB
k8s.gcr.io/pause           3.2         80d28bedfe5d  3 years ago     688 kB
```

Рисунок 2.19 – Створення образу

```
root@ubuntu-focal:~# podman run -d --rm --name test test_me:1.0
48ca46e5a4f7c37111ffebcd3f03a83c5e2e0b975d8717e2e0293981475fbf77
root@ubuntu-focal:~# podman ps
CONTAINER ID  IMAGE                  COMMAND     CREATED        STATUS            PORTS       NAMES
48ca46e5a4f7  localhost/test_me:1.0  sleep 360   5 seconds ago  Up 5 seconds ago              test
root@ubuntu-focal:~# podman exec -it test /bin/sh
/ # echo $NAME
test
/ # exit
```

Рисунок 2.20 – Запуск та підключення до контейнеру

```
root@ubuntu-focal:~# docker pause test
Error response from daemon: No such container: test
root@ubuntu-focal:~# podman pause test
48ca46e5a4f7c37111ffebcd3f03a83c5e2e0b975d8717e2e0293981475fbf77
root@ubuntu-focal:~# podman unpause test
48ca46e5a4f7c37111ffebcd3f03a83c5e2e0b975d8717e2e0293981475fbf77
root@ubuntu-focal:~# podman stop test
test
root@ubuntu-focal:~# podman rm test
48ca46e5a4f7c37111ffebcd3f03a83c5e2e0b975d8717e2e0293981475fbf77
root@ubuntu-focal:~# podman ps -a
CONTAINER ID  IMAGE       COMMAND     CREATED     STATUS      PORTS       NAMES
root@ubuntu-focal:~#
```

Рисунок 2.21 – Зупинення та видалення контейнера.

Якщо подивитися схематично на роботу Podman з контейнерами, то можна помітити, що майже така сама: є клієнт, через якого здійснюється взаємодія з Podman утилітою, яка в свою чергу передає відповідні інструкції середі виконання контейнерів (container runtime), а та вже створює відповідні ресурси. Але на відміну від Docker, Podman не вимагає наявності dockerd або якогось іншого «Podman демона», а застосовує «fork/exec» модель [38] (Рис. 2.22).

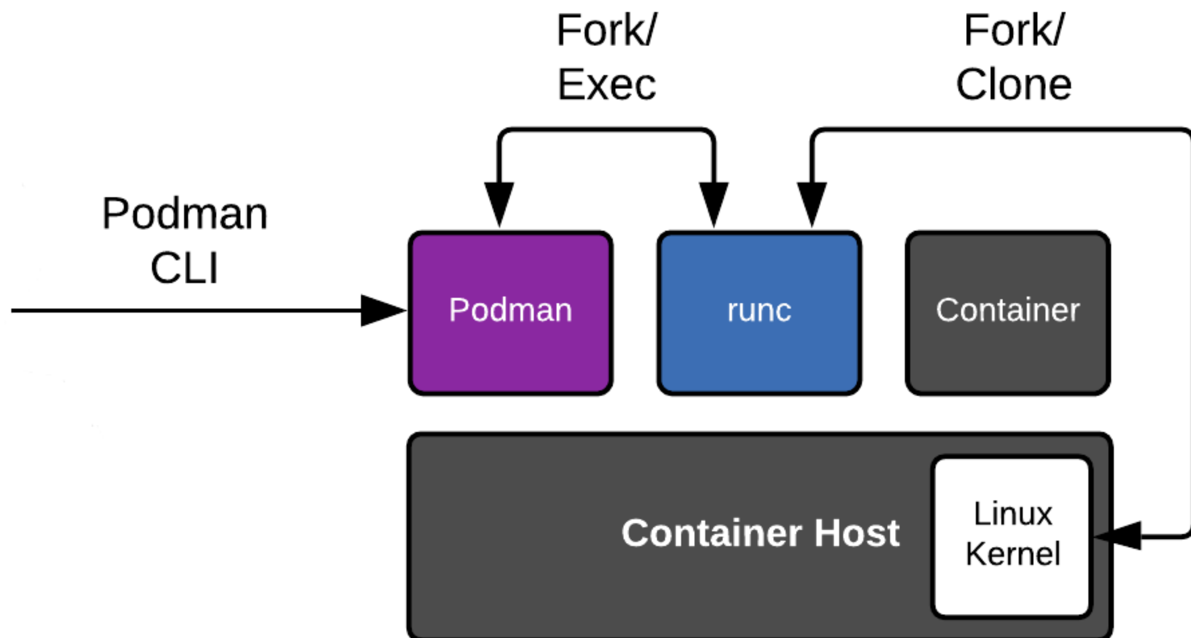


Рисунок 2.22 – Схема роботи Podman

За такої моделі у випадку створення контейнеру Podman робить fork або створює новий процес на базі процесу Podman. Потім процес init замінюється вказаною командою або точкою входу контейнеризованої програми за допомогою системного виклику exec. Цей процес гарантує, що контейнер працює в ізольованому середовищі з власним процесом ініціалізації та пов’язаними процесами. Така модель гарантує, що кожен контейнер працює у власному окремому просторі, як добре укомплектована коробка, яка не заважає решті вашої системи.

Іншою особливістю роботи Podman є наявність можливості створювати такі об’єкти як pod-и. Концепція pod-у прийшлу зі світу Kubernetes та позначає групу з

одного або кількох контейнерів зі спільним сховищем і мережевими ресурсами, а також специфікацією того, як запускати контейнери. Вміст pod-у завжди розміщується разом, а також працюють в спільному контексті. Pod моделює специфічний для програми «логічний хост»: він містить один або більше контейнерів програми, які відносно тісно пов’язані [21]. У випадку з Podman схематично pod можна зобразити наступним чином (Рис. 2.23).

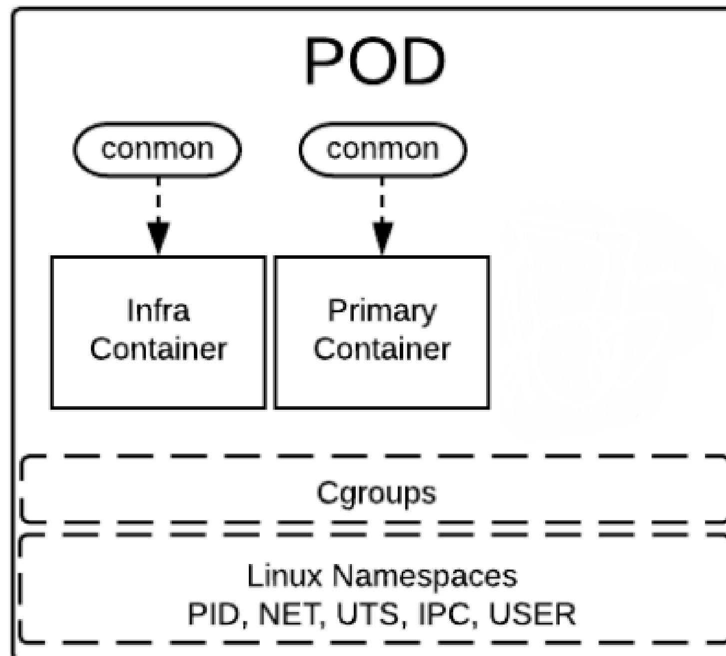


Рисунок 2.23 – Схематичне зображення Pod

Де infra container виступає у якості першого контейнеру, який утримає cgroups та Linux просторі імен, що будуть також доступні основному контейнеру (primary container) для утилізації, а conmon – це невелика утиліта моніторингу, яка використовується Podman для відстеження активності контейнерів. Його метою є моніторинг і збір інформації про процеси, що виконуються всередині контейнера, допомагаючи з відстеженням ресурсів, журналюванням і обробкою подій життєвого циклу контейнера.

Для того щоб керувати pod-ами потрібно використовувати команду pod, що дозволяє створення, запуск, зупинку та перегляд діючих та їх видалення (Рис. 2.24).

```
root@ubuntu-focal:~# podman pod create --name my-pod
fff1e914014577cc870889e6cc464b7e274e896ee0e058b0efff3fa525a2b1fa
root@ubuntu-focal:~# podman pod ls
POD ID        NAME        STATUS      CREATED        INFRA ID      # OF CONTAINERS
fff1e9140145  my-pod      Created     7 seconds ago  4a4755964f27  1
root@ubuntu-focal:~# podman create --pod my-pod --name test-container test_me:1.0
cb59f54958e2b877dfae2d7bb93ab3f9559b571970a00f558c85d9318814f568
root@ubuntu-focal:~# podman pod ls
POD ID        NAME        STATUS      CREATED         INFRA ID      # OF CONTAINERS
fff1e9140145  my-pod      Created     23 seconds ago  4a4755964f27  2
root@ubuntu-focal:~# podman pod start my-pod
fff1e914014577cc870889e6cc464b7e274e896ee0e058b0efff3fa525a2b1fa
root@ubuntu-focal:~# podman ps
CONTAINER ID  IMAGE                  COMMAND     CREATED             STATUS            PORTS       NAMES
4a4755964f27  k8s.gcr.io/pause:3.5               About a minute ago  Up 8 seconds ago              fff1e9140145-infra
cb59f54958e2  localhost/test_me:1.0  sleep 360   46 seconds ago      Up 7 seconds ago              test-container
root@ubuntu-focal:~# podman pod stop my-pod
fff1e914014577cc870889e6cc464b7e274e896ee0e058b0efff3fa525a2b1fa
root@ubuntu-focal:~# podman ps -a
CONTAINER ID  IMAGE                  COMMAND     CREATED             STATUS                         PORTS       NAMES
4a4755964f27  k8s.gcr.io/pause:3.5               About a minute ago  Exited (0) 29 seconds ago                  fff1e9140145-infra
cb59f54958e2  localhost/test_me:1.0  sleep 360   About a minute ago  Exited (137) 19 seconds ago                test-container
root@ubuntu-focal:~# podman pod rm my-pod
fff1e914014577cc870889e6cc464b7e274e896ee0e058b0efff3fa525a2b1fa
root@ubuntu-focal:~# podman pod ls
POD ID      NAME        STATUS      CREATED     INFRA ID    # OF CONTAINERS
root@ubuntu-focal:~# _
```

Рисунок 2.24 – Управління pod-ами за допомогою Podman

Таким чином, можна підсумувати, що наразі у використанні находяться різні інструменти для створення та управління контейнерами, такими як Docker та Podman. Обидва інструменти забезпечують можливість створення, запуску, зупинення/відновлення та видалення контейнерів, а також роботу з образами, реєстром та іншими компонентами. Так сам обидва є високорівневими рішеннями, що «під капотом» застосують низкьорівневе ПЗ – середи виконання контейнерів, такі як runc та/або containerd. Однак на відміну від Docker, Podman не застосвую «демон» підхід, а покладається на «fork/exec» модель, яка надає перевагу більшої ізольованості контейнерів. Ще одною відмінністю можна назвати наявність такої сутності у Podman як pod, під яким розуміється певне логічне угрупування контейнерів (одного або декількох) на одному хості, які ділять між собою певні простори імен (наприклад, мережевий) та нероздільні один від одного.

2.2 Технологія управління контейнерами у кластері

2.2.1 Поняття та моделі управління контейнерами (оркестрація та хореографія) у кластері

Попит на ефективне розгортання додатків, масштабованість і узгодженість у різноманітних обчислювальних середовищах породив керування контейнерами. Історично розгортання програмного забезпечення стикалося з проблемами, пов'язаними із залежностями, сумісністю та керуванням версіями. Docker, представлений у 2013 році, зробив революцію в ландшафті, інкапсулюючи програми та їхні залежності в легкі портативні контейнери. Цей підхід спростив процес розгортання, створивши більш надійне та відтворюване середовище.

Важливість управління контейнерами поширюється на полегшення практики DevOps, дозволяючи розробникам зосередитися на коді, а не на складнощах інфраструктури. Коли організації переходили на архітектуру мікросервісів, керування контейнерами стало невід'ємною частиною досягнення гнучкості, масштабованості та ефективності роботи.

Історично рішення для керування контейнерами розвинулися у вигляді певних оркестраторів, щоб вирішити складність організації великої кількості контейнерів у розподілених системах. Вперше з'явившись у вигляді внутрішньої системи Borg у Google у 2003-2004 роках, та згодом трансформувавшись у Omega, та у Kubernetes у 2014 році, ця система стала де-факто еталонною платформою оркестровки контейнерів [13]. Kubernetes забезпечив автоматизоване масштабування, балансування навантаження та можливості самовідновлення, змінивши розгортання та керування контейнерними програмами. Разом з тим існує й інший підхід до керування контейнерів – хореографія. Таким чином, не дивлячись на домінування Kubernetes доцільно розглянути та порівняти обидва підходи.

Використання мікросервісної архітектури, надає багато переваг але основна ідея полягає в тому, що програма поділяється на серію модульних і повторно використовуваних служб. Невеликі, легкі та прості у впровадженні служби

призводять до зниження витрат на розробку та модифікацію та дозволяють швидко та легко масштабувати.

Контейнерна оркестрація – це автоматизоване керування та координація контейнерних додатків, що полегшує розгортання, масштабування та обслуговування в кластері машин. Він включає інструменти та платформи, які оптимізують розгортання та роботу контейнерних робочих навантажень, забезпечуючи ефективне використання ресурсів і безперебійне керування додатками. У даному підході існує певний контролер – центральний елемент, який керує усіма контейнерами та іншими ресурсами, та відповідає за комунікацію між ними (Рис. 2.25) [28].

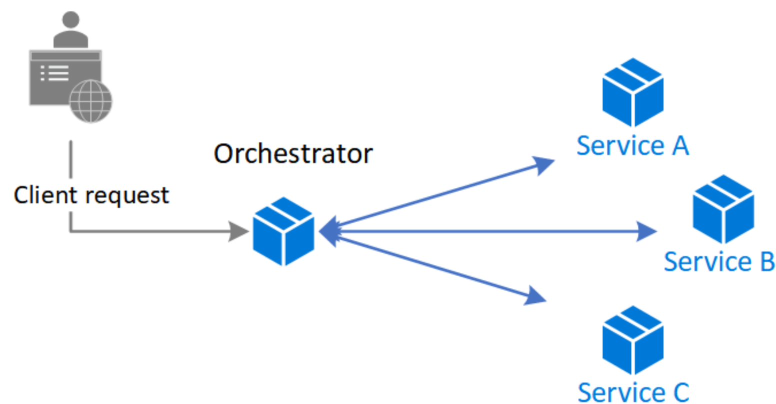


Рисунок 2.25 – Схематичне зображення роботи сервісів використовуючи метод оркестрації

Ключовими елементами такого підходу є:

1) Автоматизація розгортання: інструменти оркестровки автоматизують розгортання контейнерів, спрощуючи процес і усуваючи ручне втручання.

2) Масштабування: автоматичне масштабування дозволяє програмам адаптуватися до різних навантажень шляхом динамічного регулювання кількості запущених контейнерів.

3) Балансування навантаження: платформи оркестровки розподіляють вхідний трафік між контейнерами для оптимізації продуктивності та запобігання перевантаженню.

4) Виявлення служб (service discovery): інструменти дозволяють контейнерам виявляти та спілкуватися один з одним у динамічних і розподілених середовищах.

5) Моніторинг працездатності: інструменти оркестровки відстежують працездатність контейнерів, автоматично перезапускаючи або переплановуючи невдалі екземпляри.

Як вже зазначалося прикладами таких систем є Kubernetes, OpenShift Container Platform, Docker Swarm, Nomad та Apache Mesos.

Інший підхід полягає у хореографії контейнерами або мікросервісами за якого коли кожна служба спілкується з іншими службами безпосередньо, без центрального посередника. За такого підходу хореографії служби публікують і підписуються на події або повідомлення, і кожна служба реагує на ці події або оголошення незалежно (Рис. 2.26). Такий підхід забезпечує більшу автономію та масштабованість служб, оскільки кожна служба може реагувати на події, не покладаючись на центрального оркестратора [29].

Ключовими елементами такого підходу є:

1) Децентралізована координація: служби спілкуються одна з одною напряму, приймають власні рішення та реагують на події без центральної координації.

2) Архітектура, керована подіями: хореографія покладається на події для запуску дій і зв’язку між службами.

3) Автономність: кожна служба працює незалежно, приймаючи рішення на основі місцевих знань і реагуючи на події автономно.

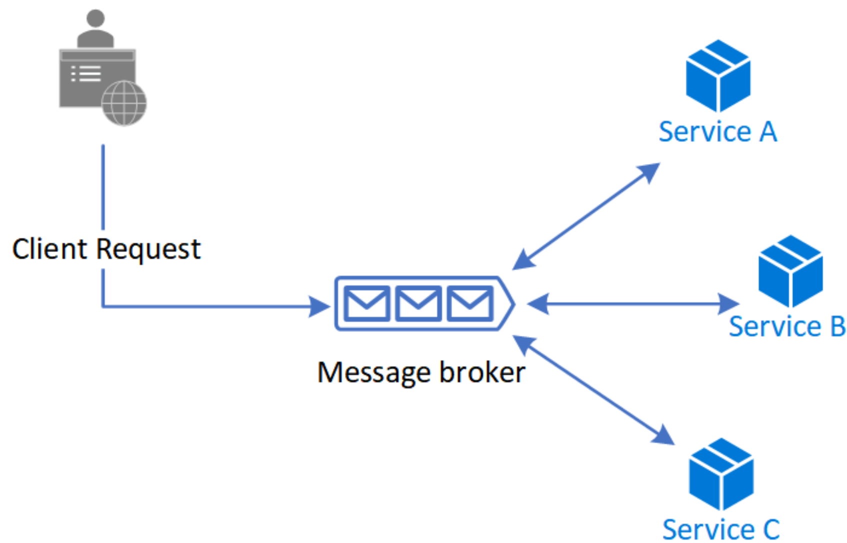


Рисунок 2.26 – Схематичне зображення роботи сервісів використовуючи метод хореографії

Прикладами роботи таких систем є Apache Kafka, Rabbit MQ та NATS. Разом з тим потрібно зауважити, що все це приклади систем обмін повідомлення або брокерів повідомлень, що не здатні розгортати контейнери.

Разом з тим, як вже зазначалося, історично перший підхід знайшов більше поширення щодо управління та розгортанням контейнеризованих сервісів, і саме Kubernetes є де факто стандартом у даній галузі. Kubernetes має низку переваг серед яких можна виокремити:

1) Потужні можливості оркестровки контейнерів. Можна сказати, що Kubernetes було розроблено разом з Docker. Він глибоко інтегрований з Docker контейнерами і природним чином адаптується до характеристик контейнерів. Він має потужні можливості оркестровки контейнерів, такі як створення контейнерів, вибір міток і виявлення служб, щоб задовольнити потреби на рівні підприємства.

2) Легкість. Kubernetes дотримується теорії архітектури мікросервісів. Вся система розділена на компоненти з незалежними функціями. Межі між компонентами чіткі, розгортання просте, і його можна легко запускати в різних

системах і середовищах. Водночас багато функцій у Kubernetes є плагінами, які можна легко розширити та замінити.

3) Відкритий код. Kubernetes відповідає тенденціям відкритого та відкритого коду, залучаючи багатьох розробників і компаній до участі в ньому та спільної роботи для побудови екосистеми. Водночас Kubernetes активно співпрацює та розвивається разом із спільнотами з відкритим кодом, такими як OpenStack та Docker. Як підприємства, так і окремі особи можуть брати участь і отримувати від цього користь [19, С. 318].

Таким чином, з розвитком контейнеризації було змінено підхід до розробки, розгортання та оперування сервісами, що породило необхідність у ефективному способі керування та налагодження взаємодії між контейнеризованими сервісами. Існують, що найменше, два підходи до вирішення вказаної проблеми: 1) оркестрація та 2) хореографія.

Під оркестрацією розуміється автоматизоване керування та координація контейнерних додатків, що забезпечується шляхом існування певного контролеру – центральний елемент усіє системи, що відповідальний за ці задачі. Він включає інструменти та платформи, які оптимізують розгортання та роботу контейнерних робочих навантажень, забезпечуючи ефективне використання ресурсів і безперебійне керування додатками.

Під хореографією контейнерами або мікросервісами досягається ситуація коли кожна служба спілкується з іншими службами безпосередньо, без центрального посередника. За такого підходу хореографії служби публікують і підписуються на події або повідомлення, і кожна служба реагує на ці події або оголошення незалежно. Такий підхід забезпечує більшу автономію та масштабованість служб, оскільки кожна служба може реагувати на події, не покладаючись на центрального оркестратора.

2.2.2 Поняття та загальний огляд Kubernetes як інструменту оркестрації

Як вже було зазначено Kubernetes (або k8s) – це портативна, розширювана платформа з відкритим вихідним кодом для керування контейнерними робочими навантаженнями та службами, яка полегшує як декларативне налаштування, так і автоматизацію [20].

Kubernetes надає вам структуру для стійкої роботи розподілених систем. Він піклується про масштабування та відновлення після відмови вашої програми, надає шаблони розгортання тощо. Наприклад: Kubernetes може легко керувати розгортанням типу canary для вашої системи.

Kubernetes надає вам:

- Виявлення служби та балансування навантаження. Kubernetes може розкривати контейнер за допомогою імені DNS або власної IP-адреси. Якщо трафік до контейнера великий, Kubernetes може балансувати навантаження та розподіляти мережевий трафік, щоб розгортання було стабільним;
- Організація зберігання. Kubernetes дозволяє автоматично монтувати систему зберігання на ваш вибір, наприклад локальні сховища, публічні хмарні постачальники тощо;
- Автоматичне розгортання та відкат. Ви можете описати бажаний стан своїх розгорнутих контейнерів за допомогою Kubernetes, і він може змінити фактичний стан на бажаний із контрольованою швидкістю. Наприклад, ви можете автоматизувати Kubernetes для створення нових контейнерів для вашого розгортання, видалення існуючих контейнерів і адаптації всіх їхніх ресурсів до нового контейнера;
- Автоматичне пакування в бункер. Ви надаєте Kubernetes кластер вузлів, які він може використовувати для виконання контейнерних завдань. Ви повідомляєте Kubernetes, скільки процесора та пам’яті (RAM) потрібно кожному контейнеру. Kubernetes може розмістити контейнери на ваших вузлах, щоб якнайкраще використовувати ваші ресурси;

- Самовідновлення. Kubernetes перезапускає контейнери, які вийшли з ладу, замінює контейнери, ліквідує контейнери, які не реагують на визначену користувачем перевірку працездатності, і не рекламує їх клієнтам, доки вони не будуть готові до обслуговування;
- Управління секретами та налаштуваннями. Kubernetes дозволяє зберігати та керувати конфіденційною інформацією, як-от паролі, маркери OAuth і ключі SSH. Ви можете розгортати та оновлювати секрети та конфігурацію програми без перебудови образів контейнерів і без розкриття секретів у конфігурації стека;
- Пакетне виконання. Крім служб, Kubernetes може керувати вашими пакетними навантаженнями та робочими навантаженнями CI, замінюючи за бажання контейнери, які вийшли з ладу;
- Горизонтальне масштабування. Масштабуйте свою програму вгору та вниз за допомогою простої команди, за допомогою інтерфейсу користувача або автоматично залежно від використання ЦП;
- Подвійний стек IPv4/IPv6. Призначення адрес IPv4 і IPv6 для модулів і служб
- Розроблено для розширення. Додайте функції до свого кластера Kubernetes, не змінюючи початковий вихідний код [20].

Kubernetes дотримується теорії архітектури мікросервісів. Вся система розділена на компоненти з незалежними функціями. Межі між компонентами чіткі, розгортання просте, і його можна легко запускати в різних системах і середовищах [19, С. 320].

Коли ви розгортаєте Kubernetes, ви отримуєте кластер. Кластер Kubernetes складається з набору робочих машин, які називаються вузлами (Nodes), які запускають контейнерні програми. Кожен кластер має принаймні один робочий вузол (Рис. 2.27).

Робочі вузли розміщують pod-и, які є компонентами робочого навантаження програми. Control plane керує робочими вузлами та pod-ами в кластері. У

промислових середовищах control plane зазвичай працює на кількох комп’ютерах, а кластер зазвичай запускає кілька вузлів, забезпечуючи відмовостійкість і високу доступність.

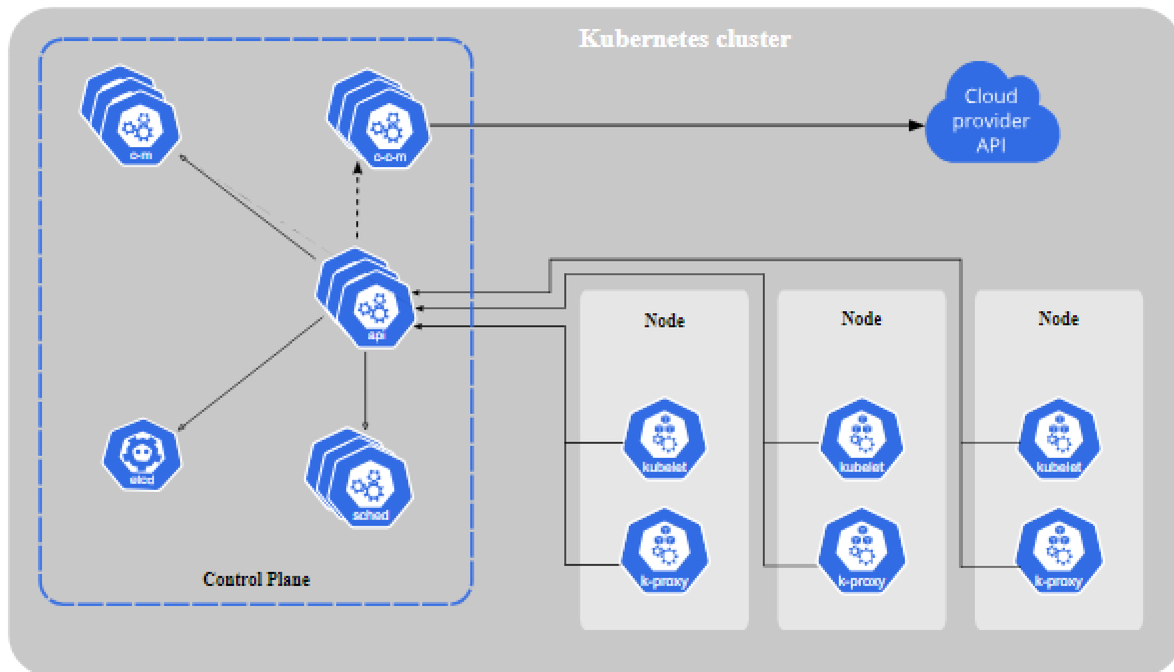


Рисунок 2.27 – Компоненти Kubernetes кластера

Kubernetes належить до розподіленої архітектури «головний-підлеглий», і вузли, з точки зору ролей, поділяються на головні (master або control plane) та вузли.

Kubernetes використовує etcd як проміжне програмне забезпечення для зберігання. *Etcd* – це високодоступне, узгоджене (strongly consistent) розподілене сховище ключів і значень, яке забезпечує надійний спосіб зберігання даних, доступ до яких потребує розподілена система або кластер машин. Сховище було натхненне ZooKeeper і Doozer, яке використовує консенсусний алгоритм Raft для обробки реплікації журналу для забезпечення надійної узгодженості. Kubernetes використовує etcd як центр зберігання конфігурації системи. Важливі дані в Kubernetes зберігаються в etcd, що робить різні компоненти архітектури Kubernetes без збереження стану (stateless), що полегшує реалізацію розгортання розподіленого кластера.

Control plane у Kubernetes відноситься до вузла керування кластером. Кожному кластеру Kubernetes потрібен головний вузол, який відповідатиме за керування всім кластером. На нього надсилаються всі керуючі команди Kubernetes, відповідальні за конкретний процес виконання. Усі команди, які клієнти надсилають до кластеру, виконуються на головному вузлі. Головний вузол зазвичай займає незалежний сервер (для розгортання високої доступності рекомендується три сервери).

Сервер Kubernetes API: як вхід до системи Kubernetes, він інкапсулює операції додавання, видалення, модифікації та запитів основних об’єктів і надає зовнішнім клієнтам і виклики внутрішніх компонентів у формі REST API. Об’єкти REST, які він підтримує, зберігатимуться в etcd.

Наступний елемент control plane це *kube-scheduler*. Kube-scheduler спостерігає за новоствореними pod-ами без призначеного вузла та вибирає вузол для їх запуску. Фактори, які враховуються для прийняття рішень щодо планування, включають: індивідуальні та колективні вимоги до ресурсів, обмеження апаратного/програмного забезпечення/політики, специфікації спорідненості та антиспорідненості, локальність даних, перешкоди між робочими навантаженнями та кінцеві терміни.

Kube-controller-manager – компонент control plane який запускає та керує різноманітними контролерами (controller). Логічно кожен контролер є окремим процесом, але для зменшення складності всі вони скомпільовані в один бінарний файл і виконуються в одному процесі. Є велика кількість різних контролерів, але можна виділити наступні:

- Контролер вузлів: відповідає за помічання та реагування, коли вузли виходять з ладу;
- Контролер завдань (jobs): стежить за об’єктами типу job, які представляють одноразові завдання, а потім створює модулі для виконання цих завдань до кінця;

- Контролер EndpointSlice: заповнює об'єкти EndpointSlice (для забезпечення зв'язку між службами та модулями);
- Контролер ServiceAccount: створює облікові записи ServiceAccount за умовчанням для нових просторів імен.

Ще одним компонентом є cloud-controller-manager, який не є обов'язковою частиною Kubernetes кластеру, але з появою готових Kubernetes кластерів на базі хмарних провайдерів, такий компонент є їх інтегральним компонентом. Цей компонент control plane, який включає спеціальну хмарну логіку керування. Менеджер хмарного контролера дозволяє підключити ваш кластер до API вашого хмарного провайдера та відокремлює компоненти, які взаємодіють із цією хмарною платформою, від компонентів, які взаємодіють лише з вашим кластером. Тобто cloud-controller-manager запускає лише контролери, які є специфічними для вашого хмарного провайдера (наприклад, AWS Cloud Controller Manager).

Наступним елементом Kubernetes кластеру є робочі вузли (worker nodes), які складаються з kubelet, kube-proxy та середи виконання контейнерів (container runtime).

Kubelet – Агент, який працює на кожному вузлі в кластері. Він гарантує, що контейнери працюють у Pod. Kubelet використовує набір PodSpec, які надаються через різні механізми, і гарантує, що контейнери, описані в цих PodSpec, працюють і є справними. Водночас kubelet тісно співпрацює з control plane для реалізації основних функцій управління кластером.

Kube-proxy – життєво важливий компонент, який реалізує зв'язок і механізм балансування навантаження Kubernetes сервісів. Він підтримує мережеві правила на вузлах, які роблять можливим мережевий зв'язок із pod-ами під час мережевих сеансів у кластері чи поза ним. Цікавим є те, що один лиш цей компонент є недостатнім для побудови мережевого з'єднання між Kubernetes ресурсами, та вимагає встановлення мережевого плагіну (CNI plugin), таких як Flannel, Calico, Weave Net, та Cilium.

Середа виконання контейнерів (*container runtime*) – основний компонент, який дозволяє Kubernetes ефективно керувати контейнерами, він наглядає за виконанням і керуванням життєвим циклом контейнерів у середовищі Kubernetes. Підтримує різні середи виконання контейнерів, які підтримують Kubernetes CRI (складається зі специфікацій/вимог, API protobuf і бібліотек для середовища виконання контейнерів для інтеграції з kubelet на вузлі) [6].

Таким чином, чином Kubernetes – це платформа оркестровки контейнерів з відкритим кодом, яка автоматизує розгортання, масштабування та керування контейнерними програмами. Він забезпечує надійну структуру для координації та керування контейнерами в кластерах машин, спрощуючи складності, пов’язані з розгортанням і масштабуванням програм у динамічних і розподілених середовищах. Структурно складається з control plane (або master node), тобто центрального вузла, через який здійснюється комунікація з іншими вузлами кластеру (worker nodes) та їх керування.

2.2.3 Основні ресурси у Kubernetes кластеру

Одним із критичних компонентів Kubernetes cluster plane є kube API server, який надає інтерфейс для керування та моніторингу кластера, та який використовують інші користувачі кластера та cluster plane. Kubernetes API є декларативним і надає такі кінцеві точки, як створення, виправлення, отримання та видалення (*create*, *patch*, *get* та *delete*). Ці команди створюють, читають, оновлюють і видаляють ресурси з конфігурації кластера – конкретна конфігурація кожного ресурсу повідомляє Kubernetes, що ми хочемо робити від кластера.

Цей декларативний API необхідний для вирішення наскрізних проблем динамічного планування та розподіленого стану. Оскільки декларативний API просто повідомляє або оновлює конфігурацію кластера, реагувати на збої сервера або мережі, які можуть спричинити пропуск команди, дуже легко. Розглянемо приклад, коли з’єднання з API-сервером втрачається відразу після того, як надана

команда застосування для зміни конфігурації кластера. Коли з'єднання відновлено, клієнт може просто запитати конфігурацію кластера та визначити, чи команда була отримана успішно. Або, що ще простіше, клієнт може просто виконати ту саму команду *apply* ще раз, знаючи, що поки конфігурація кластера буде бажаною, Kubernetes намагатиметься зробити «правильні речі» з фактичним кластером. Цей основний принцип відомий як ідемпотентність (idempotence), тобто безпечно виконувати ту саму команду кілька разів, оскільки вона буде застосована щонайбільше один раз [18].

В силу зазначених особливостей робота з ресурсами Kubernetes відбувається декларативно шляхом їх опису у конфігураційних файлах (*.YAML) та передання до kube API серверу (Рисунок 2.28).

```
azureuser$ cat test.yaml
apiVersion: v1
kind: Pod
metadata:
  creationTimestamp: null
  labels:
    run: po
  name: po
spec:
  containers:
  - args:
    - test
    image: nginx
    name: po
    resources: {}
  dnsPolicy: ClusterFirst
  restartPolicy: Always
status: {}
azureuser$ kubectl apply -f test.yaml
pod/po created
azureuser$ kubectl get pods
NAME   READY   STATUS    RESTARTS      AGE
po     1/1     Running   3 (24s ago)   43s
```

Рисунок 2.28 – управління Kubernetes ресурсами

Існує безліч Kubernetes ресурсів, але можна виділити певні фундаментальні серед них, саме:

Pods. Це найменші та найпростіші одиниці в об’єктній моделі Kubernetes. Вони представляють окремий екземпляр запущеного процесу та можуть містити один або кілька контейнерів, які спільно використовують той самий мережевий простір імен.

ReplicaSets. Забезпечують постійну роботу певної кількості реплік (копій) pod-ів. Вони відповідають за підтримку бажаної кількості екземплярів pod-ів, що робить їх ключовим ресурсом для досягнення високої доступності.

Deployments. Забезпечують декларативний спосіб керування розгортанням і масштабуванням програм. Вони являють собою абстракцію вищого рівня, яка використовує ReplicaSets для забезпечення роботи бажаної кількості реплік pod-ів, полегшуючи оновлення та відкат.

Services. Визначають логічний набір pod-ів і політику доступу до них. Вони забезпечують зв’язок між різними частинами програми та забезпечують стабільну IP-адресу та ім’я DNS, що дозволяє іншим службам виявляти їх і підключатися до них.

ConfigMaps. Зберігають конфігураційні дані як пари ключ-значення, що дозволяє відокремити конфігурацію від коду програми. Вони зазвичай використовуються для налаштування програм і монтуються як томи або змінні середовища в pod-ах.

Secrets. Зберігають конфіденційну інформацію, таку як паролі або ключі API, у безпечний спосіб. Вони закодовані в Base64 і можуть використовуватися pod-ами для доступу до конфіденційних даних, не розкриваючи їх у конфігурації.

ServicesAccounts: ServiceAccounts надають ідентичність для процесів, запущених у модулі. Вони використовуються для надання дозволів групам і дозволяють їм взаємодіяти з іншими ресурсами в кластері.

Namespace: простори імен забезпечують спосіб створення віртуальних кластерів у фізичному кластері. Вони допомагають в організації та ізоляції ресурсів,

дозволяючи кільком командам або проектам використовувати один кластер, не заважаючи один одному.

Persistent Volumes (PVs) і Persistent Volume Claims (PVCs). Керують ресурсами зберігання в кластері. PV представляють фізичне сховище, тоді як PVC – це запити на зберігання контейнерами. Вони забезпечують постійне зберігання для програм із збереженням стану.

У випадку існування контейнеризованого додатку, наприклад nginx веб серверу типовий процес розгортання буде виглядати наступним чином. Спочатку можна створити окремий namespace для логічного обмеження ресурсів, що ми плануємо розгорнути у кластері (Рис. 2.29).

```
azureuser$ k create ns test
namespace/test created
azureuser$ k describe ns test
Name:         test
Labels:       kubernetes.io/metadata.name=test
Annotations:  <none>
Status:       Active

No resource quota.

No LimitRange resource.
```

Рис. 2.29 – Створення та опис namespace ‘test’

Після цього ми можемо створити файл з конфігурацією нашого веб серверу у вигляді ConfigMap-и:

```
apiVersion: v1
kind: ConfigMap
metadata:
  name: nginx-config
data:
  nginx.conf: |
    server {
        listen 80;
        server_name localhost;

        location / {
            root /usr/share/nginx/html;
            index index.html;
        }
    }
```

та бажаним змістом домашньої веб сторінки, також у вигляді ConfigMap-и:

```
apiVersion: v1
kind: ConfigMap
metadata:
  name: custom-webpage
data:
  index.html: |
    <!DOCTYPE html>
    <html>
    <head>
      <title>Custom Web Page</title>
    </head>
    <body>
      <h1>Welcome to my custom web page!</h1>
    </body>
    </html>
```

Надалі необхідно створити Deployment з опису бажаного веб додатку, що має бути розгорнутий у кластері, а також передати йому бажану конфігурації, що були описана вище, та задати бажану кількість pod-ів:

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: nginx-deployment
spec:
  replicas: 3
  selector:
    matchLabels:
      app: nginx
  template:
    metadata:
      labels:
        app: nginx
    spec:
      containers:
      - name: nginx
        image: nginx:latest
        volumeMounts:
        - name: config-volume
          mountPath: /etc/nginx/conf.d
        - name: custom-webpage-volume
          mountPath: /usr/share/nginx/html
      volumes:
      - name: config-volume
        configMap:
          name: nginx-config
      - name: custom-webpage-volume
        configMap:
          name: custom-webpage
```

І на останок необхідно описати service (типу NodePort в нашому випадку), для того, щоб був створений DNS запис у кластері до наших pod-ів – веб-серверу та було відкрито порти на хостових машинах для доступу до нього:

```
apiVersion: v1
kind: Service
metadata:
  name: nginx-service
spec:
  selector:
    app: nginx
  ports:
    - protocol: TCP
      port: 80
      targetPort: 80
  type: NodePort
```

Після того як бажаний стан кластеру було описано залишилось лише передати бажану конфігурацію Kubernetes, а він створить ресурси для нас (Рис. 2.30 і Рис. 2.31).

```
azureuser$ ls
deploy.yaml  page.yaml  srv-conf.yaml  svc.yaml
azureuser$ k apply -f . -n test
deployment.apps/nginx-deployment created
configmap/custom-webpage created
configmap/nginx-config created
service/nginx-service created
azureuser$ _
```

Рисунок 2.30 – Створення ресурсів з yaml маніфестів

```
azureuser$ k get all -n test
NAME                                    READY   STATUS    RESTARTS   AGE
pod/nginx-deployment-6cbb9948b7-4zllt   1/1     Running   0          2m44s
pod/nginx-deployment-6cbb9948b7-6n9xc   1/1     Running   0          2m44s
pod/nginx-deployment-6cbb9948b7-hpm67   1/1     Running   0          2m44s

NAME                    TYPE       CLUSTER-IP     EXTERNAL-IP   PORT(S)        AGE
service/nginx-service   NodePort   10.0.189.190   <none>        80:32103/TCP   2m44s

NAME                               READY   UP-TO-DATE   AVAILABLE   AGE
deployment.apps/nginx-deployment   3/3     3            3           2m44s

NAME                                          DESIRED   CURRENT   READY   AGE
replicaset.apps/nginx-deployment-6cbb9948b7   3         3         3       2m44s
azureuser$ k get cm -n test
NAME               DATA   AGE
custom-webpage     1      2m52s
kube-root-ca.crt   1      43m
nginx-config       1      2m52s
azureuser$
```

Рисунок 2.31 – Перевірка наявності описаних ресурсів

І тепер можна перевірити доступність нашого веб-серверу, дізнавшись IP адреси хостових машин (Рис. 2.32) та здійснивши перевірку як з зовні кластеру, так і в середині його (Рис. 2.33).

```
azureuser$ k describe svc nginx-service -n test
Name:                     nginx-service
Namespace:                test
Labels:                   <none>
Annotations:              <none>
Selector:                 app=nginx
Type:                     NodePort
Endpoints:                10.110.0.36:80,10.110.0.63:80,10.110.1.97:80
Session Affinity:         None
```

Рисунок 2.33 – Опис ресурсу Service ‘nginx-server’

```
azureuser$ curl 10.110.0.36:80
<!DOCTYPE html>
<html>
<head>
  <title>Custom Web Page</title>
</head>
<body>
  <h1>Welcome to my custom web page!</h1>
</body>
</html>
azureuser$ kubectl exec -it nginx-deployment-6cbb9948b7-4zllt -n test -- /bin/sh
# curl nginx-service:80
<!DOCTYPE html>
<html>
<head>
  <title>Custom Web Page</title>
</head>
<body>
  <h1>Welcome to my custom web page!</h1>
</body>
</html>
# exit
azureuser$ _
```

Рисунок 2.34 – Перевірка доступності веб-серверу

Таким чином, створення ресурсів у Kubernetes кластері передбачає використання маніфестів YAML або JSON для визначення бажаного стану програм, служб та інших компонентів. Такий підхід Потім ви застосовуєте ці маніфести до кластера за допомогою інструмента командного рядка kubectl. Цей декларативний підхід необхідний для вирішення проблем динамічного планування та розподіленого стану.

Основними ресурсами Kubernetes кластеру, які були розглянуті у підрозділі є Pod, ReplicaSet, Deployment, Service, ConfigMap, Secret, ServiceAccount, Namespace, PV та PVC. В той же час є безліч інших ресурсів, що необхідні для

досягнення більш гнучкого налаштування та отримання бажаного результату (наприклад, DaemonSet, Jobs, ClusterRole, Roles, PodDistruptionBudget та багато інших). Типовий приклад з розгортанням веб-серверу nginx було продемонстровано.

РОЗДІЛ ІІІ РОЗГОРТАННЯ ВЕБ ЗАСТОСУНКУ У KUBERNETES КЛАСТЕРІ

3.1 Загальний огляд контейнеризованого веб застосунку

Перед тим як перейти до імплементації базового веб-застосунку, варто оглянути базові концепти такого поняття.

Якщо вести мову про базовий рівень, то веб технологія складається з великої кількості частин, але базовими є поняття веб серверу та веб клієнту. Інші компоненти, такі як проксі, сервіси кешування, розподільники навантаження та інші є лише інструментами та засобами покращення взаємодії цих двох базових компонентів.

Архітектура веб технології складається з трьох базових елементів [19, С. 39–40]:

1. Уніфікований покажчик ресурсу (URL). Стандартизований синтаксис, який використовується для позначення ідентифікатора веб-ресурсів. URL-адреси зазвичай представляють логічні розташування мережі.

2. Протокол передачі гіпертексту (HTTP). Основний комунікаційний протокол, що полегшує обмін вмістом і даними у Всесвітній павутині. Як правило, URL-адреси передаються через HTTP. Коли користувач відвідує веб-сторінку, браузер використовує HTTP, щоб надіслати запит на відповідний веб-сайт на основі URL-адреси.

3. Мова розмітки. Надання легкого методу представлення веб-орієнтованих даних і метаданих. Зараз на веб-сторінках переважає мова гіпертекстової розмітки (HTML) із фіксованими значеннями для її тегів.

Переходячи до рівня веб застосунків, то розподілені програми на основі веб-технологій (зазвичай вважаються веб-застосунками (веб-програмами). Завдяки високій доступності ці програми з’являються в усіх типах хмарних середовищ.

Типова веб-програма може мати трирівневу модель (Рис. 3.1). Перший рівень – це рівень презентації, який використовується для представлення інтерфейсу користувача. Другий рівень – прикладний рівень, який використовується для реалізації логіки програми. Третій рівень – це рівень даних, який складається з постійного зберігання даних.

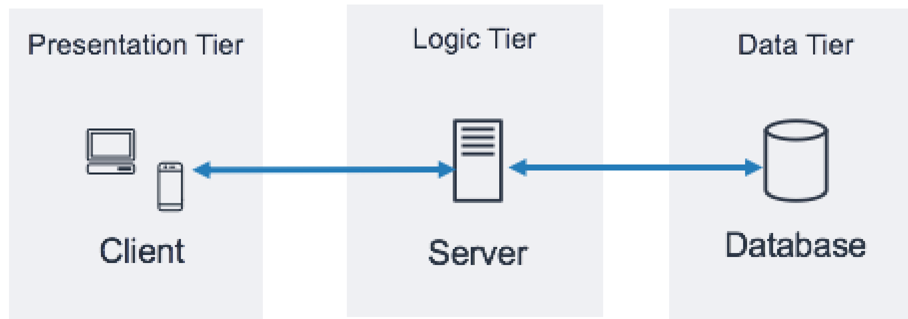


Рисунок 3.1 – Схематичне зображення трьох-шарового веб-застосунку

Для демонстрації веб застосунку було вирішено його спростити до рівня веб серверу, який буле віддавати базову веб сторінку, наповнення якої може біти змінено при початкову розгортанні. Для імплементації було обрано Flask як фреймворк для написання коду застосунку на базі мови програмування Python (третя версія).

Структурно веб додаток складається з наступних елементів (Рис. 3.2).

```
andrey@ASTEPANENKO-NB:/mnt/d/HYDROMET/master/project/app$ tree .
.
├── Dockerfile
├── app.py
├── requirements.txt
└── templates
    └── index.html

1 directory, 4 files
```

Рисунок 3.2 – Структура директорії з веб застосунком

1) app.py – файл з кодом веб-затосунку;

2) templates/index.html – темплейт файл для рендерингу веб сторінки програмою;

3) requirements.txt – файл з усіма залежностями необхідними для запуску програми;

4) Dockerfile – файл з конфігурацією для побудови образу застосунку.

Маючи наступну структуру ми повинні контейнеризувати наш додаток за допомогою Docker на базі Dockerfile наступним чином (Рис. 3.3).

```
andrey@ASTEPANENKO-NB:/mnt/d/HYDROMET/master/project/app$ docker build .
[+] Building 6.7s (3/4)
                                                            docker:default
 => [internal] load build definition from Dockerfile
                                                                      0.1s
 => => transferring dockerfile: 281B
                                                                      0.0s
 => [internal] load .dockerignore
[+] Building 7.7s (12/12) FINISHED                                                        docker:default
 => [internal] load build definition from Dockerfile                                                0.1s
 => => transferring dockerfile: 281B                                                                0.0s
 => [internal] load .dockerignore                                                                   0.0s
 => => transferring context: 2B                                                                     0.0s
 => [internal] load metadata for docker.io/library/python:3.10.13-alpine                           7.5s
 => [auth] library/python:pull token for registry-1.docker.io                                       0.0s
 => [stage-0 1/6] FROM docker.io/library/python:3.10.13-alpine@sha256:d662f68cd63678394e2fb5cde3da3a9df8f0e  0.0s
 => [internal] load build context                                                                   0.0s
 => => transferring context: 132B                                                                   0.0s
 => CACHED [stage-0 2/6] WORKDIR /app                                                               0.0s
 => CACHED [stage-0 3/6] COPY requirements.txt /app                                                 0.0s
 => CACHED [stage-0 4/6] COPY app.py /app                                                           0.0s
 => CACHED [stage-0 5/6] COPY templates /app/templates                                              0.0s
 => CACHED [stage-0 6/6] RUN --mount=type=cache,target=/root/.cache/pip     pip3 install -r requirements.tx  0.0s
 => exporting to image                                                                              0.0s
 => => exporting layers                                                                             0.0s
 => => writing image sha256:f53b67720f4d1c7cddcfab67b252f7f4f5ed3a760ff4b9a30a1ff4a6839b47cb         0.0s

What's Next?
andrey@ASTEPANENKO-NB:/mnt/d/HYDROMET/master/project/app$
```

Після цього ми можемо призначити тег образу та завантажити його до нашого приватного репозиторію (в нашому випадку *andrstp/master*) (Рис. 3.4). та перевірити чи доступний образ для завантаження (Рис. 3.5).

```
andrey@ASTEPANENKO-NB:/mnt/d/HYDROMET/master/project/app$ docker tag web-master:1.0 andrstp/master:1.0
andrey@ASTEPANENKO-NB:/mnt/d/HYDROMET/master/project/app$ docker push andrstp/master:1.0
The push refers to repository [docker.io/andrstp/master]
5200135b8473: Pushed
e58d39046eb8: Pushed
1f1e92a87dd4: Pushed
71bdc1d900ba: Pushed
ea26d3909e4d: Pushed
7dca32005037: Pushed
43e9d151612d: Pushed
3d7c75826ea8: Pushed
186ce2d777be: Pushed
9fe9a137fd00: Pushed
1.0: digest: sha256:889d594e594771bf792d136fa46d1ba5d409f4aa4acf5718fdb072ecae18a838 size: 2406
andrey@ASTEPANENKO-NB:/mnt/d/HYDROMET/master/project/app$
```

Рисунок 3.4 – Завантаження образу у репозиторій

```
andrey@ASTEPANENKO-NB:/mnt/d/HYDROMET/master/project/app$ docker images
REPOSITORY                    TAG       IMAGE ID       CREATED         SIZE
gcr.io/k8s-minikube/kicbase   v0.0.40   c6cc01e60919   4 months ago    1.19GB
alpine                        latest    9c6f07244728   16 months ago   5.54MB
busybox                       1.28      8c811b4aec35   5 years ago     1.15MB
andrey@ASTEPANENKO-NB:/mnt/d/HYDROMET/master/project/app$ docker pull andrstp/master:1.0
1.0: Pulling from andrstp/master
Digest: sha256:889d594e594771bf792d136fa46d1ba5d409f4aa4acf5718fdb072ecae18a838
Status: Downloaded newer image for andrstp/master:1.0
andrey@ASTEPANENKO-NB:/mnt/d/HYDROMET/master/project/app$ docker images
REPOSITORY                    TAG       IMAGE ID       CREATED         SIZE
andrstp/master                1.0       f53b67720f4d   16 hours ago    60.7MB
gcr.io/k8s-minikube/kicbase   v0.0.40   c6cc01e60919   4 months ago    1.19GB
alpine                        latest    9c6f07244728   16 months ago   5.54MB
busybox                       1.28      8c811b4aec35   5 years ago     1.15MB
andrey@ASTEPANENKO-NB:/mnt/d/HYDROMET/master/project/app$
```

Рисунок 3.5 – Завантаження образу веб додатку з приватного репозиторію

Надалі можна переходити до наступних етапів – розгортання Kubernetes кластеру та розгортання нашого додатку у веб-контейнері та перевірка його працездатності.

3.2 Створення та налаштування Kubernetes кластеру

Розгортання кластера Kubernetes передбачає різні варіанти, кожен з яких адаптований до конкретних випадків використання та вподобань інфраструктури. Одним із популярних підходів є використання керованих служб Kubernetes, які надаються хмарними провайдерами, такими як Amazon EKS, Google Kubernetes Engine (GKE) і Azure Kubernetes Service (AKS). Ці служби абстрагують базову складність інфраструктури, пропонуючи автоматизоване керування кластером, масштабування та оновлення.

Для локальних або самокерованих розгортань такі інструменти, як kubeadm, спрощують процес, автоматизуючи завантаження кластера. Цей метод дозволяє користувачам налаштовувати конфігурації кластера та контролювати весь життєвий цикл розгортання.

Крім того, існують спеціалізовані дистрибутиви, такі як Rancher, OpenShift і kops, кожен з яких надає унікальні функції. Rancher зосереджується на мультикластерному управлінні та простоті використання, тоді як OpenShift наголошує на корпоративних функціях та інтегрованих інструментах розробника.

kops особливо корисний для розгортання готових до виробництва кластерів на AWS.

Контейнерні платформи, такі як Docker Desktop і Minikube, пропонують легкі рішення для локальної розробки та тестування. Вони дозволяють користувачам запускати одновузлові кластери на своїх робочих станціях, полегшуючи розробку додатків і налагодження.

В нашому випадку, було обрано платформу AKS на базі Azure провайдеру.

Для розгортання кластеру та пов'язаних з ним ресурсів було обрано такий інструмент як Terraform, про який було докладно описано у Розділі 1 роботи.

Було вирішено сформувати Terraform модулі для побудови інфраструктури, де директорія з Terraform проектом виглядає наступним чином (зображена на Рис. 3.6).

```
andrey@ASTEPANENKO-NB:/mnt/d/HYDROMET/master/project$ tree .
.
├── main.tf
├── modules
│   ├── aks
│   │   ├── main.tf
│   │   ├── outputs.tf
│   │   ├── providers.tf
│   │   ├── ssh.tf
│   │   └── variables.tf
│   ├── base
│   │   ├── keyvault.tf
│   │   ├── outputs.tf
│   │   ├── rg.tf
│   │   └── variables.tf
│   └── kubernetes
│       ├── data.tf
│       ├── main.tf
│       ├── outputs.tf
│       └── variables.tf
├── outputs.tf
├── providers.tf
├── sp.txt
├── test.yaml
└── variables.tf
```

Рисунок 3.6 – Зміст директорії з файлами для розгортання інфраструктури

Передбачено наступні модулі:

1) base модуль, для розгортання ресурсної групи та Azure KeyVault (сховища секретів та ключів) та секретів – даних для підключення до приватного репозиторію;

2) aks модуль, для розгортання ресурсної групи та безпосередньо AKS кластеру (на 2 робочих вузла), та запису kubeconfig файлу на локальну машину для надання доступу до кластеру;

3) kubernetes_config модуль, для створення kubernetes ресурсів таких як просторі імен, configMap-ресурс з даними для передання нашому веб-застосунку, секрету з даними для підключення до приватного репозиторію (дані беруться з раніше створеного Azure KeyVault), а також serviceAccount clusterRole та clusterRoleBinding ресурсів для можливості застосування pod-ами для завантаження образів з нашого приватного репозиторію;

4) root модуль, з якого здійснюється запуск вищеописаних модулів.

Кожен модуль відповідає загальній практиці побудови Terraform модулів, де передбачено файли з конфігурацією ресурсів (зокрема, AKS кластер, сховище зберігання секретів Azure KeyVault, ресурсна група та інше), які передбачені у файлах з назвою *main.tf*, або з ім'я ресурсу, що має бути створений *keyvault.tf* та *rg.tf*. Наступним типом файлів є *variables.tf* в якому передбачено вхідні змінні, що використовуються для зміни параметрів ресурсів, починаючи з назви ресурсною групи і закінчуючи ім'я секрету у Kubernetes кластері. Також передбачено *outputs.tf* файли, які слугують для передачі результатів або параметрів створених ресурсів, та подальшої передачі їх як ввідних для вищезазначених змінних інших модулів, або, зокрема, для запису даних для підключення до кластеру. Останнім типом файлу, що передбачені в структурі є *providers.tf*, який містить дані щодо необхідних Terraform провайдерів (назву та версію) для створення модулів та розгортання інфраструктури проекту в цілому.

Загалом архітектура виглядає наступним чином (Рис. 3.7):

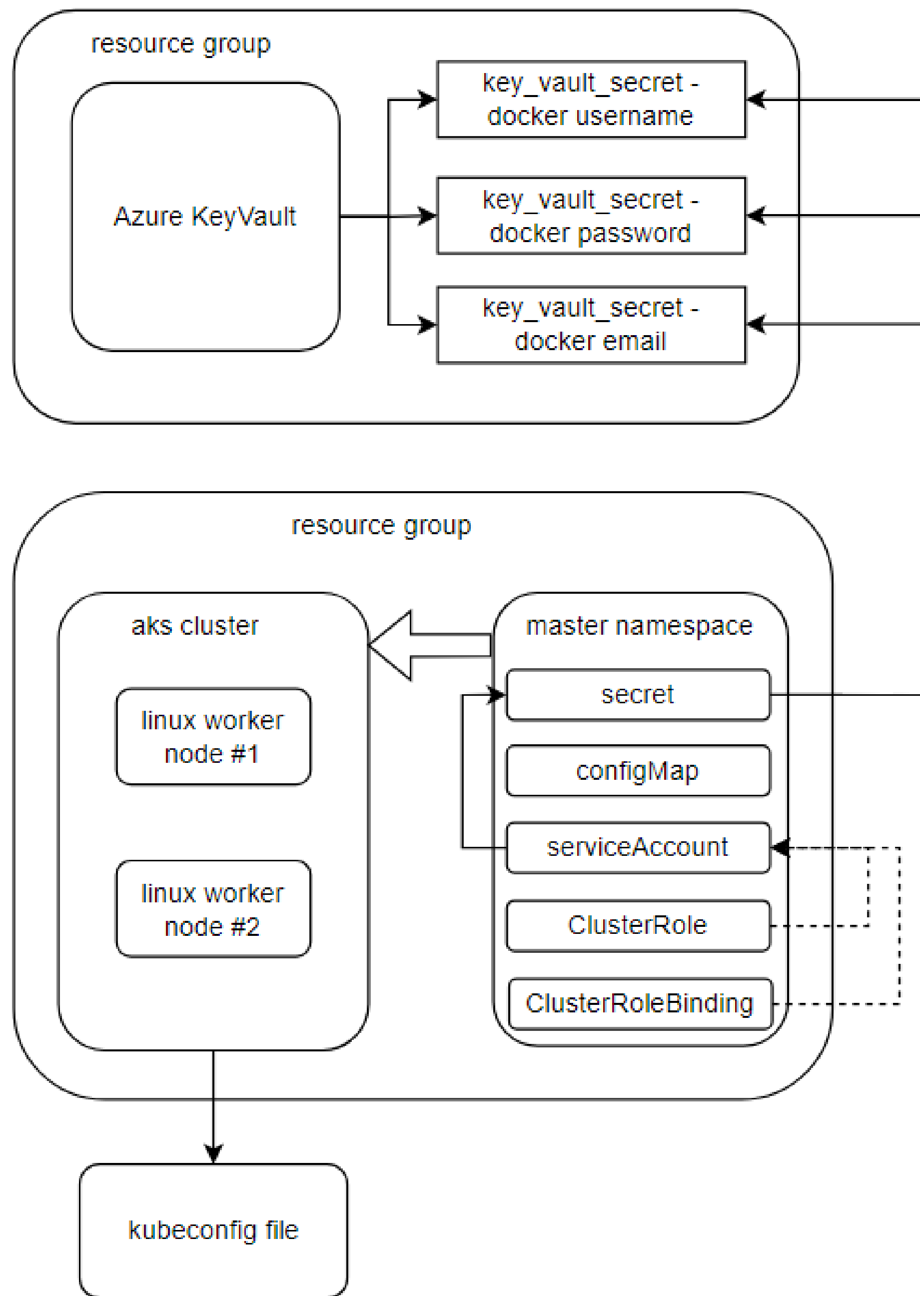


Рисунок 3.7 – Загальна схема створення хмарних ресурсів за допомогою Terraform

Після виконання підготовчих процедур з Terraform (ініціалізація та завантаження необхідних провайдерів) можна переходити до застосування описаної конфігурації Terraform-ом. Але перші ніж застосувати цю конфігурацію необхідно надати доступ Terraform на створення таких хмарних ресурсів. Одним з

способів є створення Service Principal сутності за допомогою Azure CLI утиліти (Рис. 3.8, Рис. 3.9 та Рис. 3.10).

```
andrey@ASTEPANENKO-NB:~$ az login
A web browser has been opened at https://login.microsoftonline.com/organizations/oauth2/v2.0/authorize.
 available or if the web browser fails to open, use device code flow with `az login --use-device-code`.
[
  {
    "cloudName": "AzureCloud",
    "homeTenantId": "",
    "id": "",
    "isDefault": true,
    "managedByTenants": [],
    "name": "master",
    "state": "Enabled",
    "tenantId": "",
    "user": {
      "name": "",
      "type": "user"
    }
  }
]
andrey@ASTEPANENKO-NB:~$ _
```

Рисунок 3.8 – Авторизація az CLI

localhost:61684/?code=0.AWAA_78z0DWgKkiDFGcP5odA2JV3sATbjRpGu-4C-eG_e0aRAOc.AgABAAIAAAAmoFfGtYxvRrNriQdPKIZ

You have logged into Microsoft Azure!

You can close this window, or we will redirect you to the Azure CLI documentation in 1 minute.

Announcements

[Windows only] Azure CLI is collecting feedback on using the Web Account Manager (WAM) broker for the login experience.

You may opt-in to use WAM by running the following commands:

```
az config set core.allow_broker=true
az account clear
az login
```

Рисунок 3.9 – Вікно після успішної авторизації у Azure

```
andrey@ASTEPANENKO-NB:~$ export MSYS_NO_PATHCONV=1
andrey@ASTEPANENKO-NB:~$ az ad sp create-for-rbac --name <service_principal_name> --role Contributor --scopes /subscriptions/<subscription_id>
Creating 'Contributor' role assignment under scope '/subscriptions/'
The output includes credentials that you must protect. Be sure that you do not include these credentials in your code or check the credentials
or more information, see https://aka.ms/azadsp-cli
{
  "appId": "",
  "displayName": "tf-master-sp",
  "password": "",
  "tenant": ""
}
andrey@ASTEPANENKO-NB:~$
```

Рисунок 3.10 – Створення Service Principal з роллю Contibutor.

Та передання необхідних для авторизації даних через змінні оточення (Рис. 3.11).

```
andrey@ASTEPANENKO-NB:/mnt/d/HYDROMET/master/project$ cat env_vars_cp.sh
#!/bin/bash

export ARM_SUBSCRIPTION_ID="<azure_subscription_id>"
export ARM_TENANT_ID="<azure_subscription_tenant_id>"
export ARM_CLIENT_ID="<service_principal_appid>"
export ARM_CLIENT_SECRET="<service_principal_password>"
andrey@ASTEPANENKO-NB:/mnt/d/HYDROMET/master/project$ source env_vars.sh
```

Рисунок 3.11 – Запуск створення Terraform-ом описаних у проекті ресурсів

Після цього можна приступати до запуску Terraform (Рис. 3.12)

```
Plan: 24 to add, 0 to change, 0 to destroy.

Changes to Outputs:
  + cluster_password        = (sensitive value)
  + cluster_username        = (sensitive value)
  + host                    = (sensitive value)
  + keyvault_id             = (known after apply)
  + keyvault_name           = (known after apply)
  + kube_config             = (sensitive value)
  + kubernetes_cluster_name = (known after apply)
  + namespace_name          = "master"
  + resource_group_location = "eastus"
  + resource_group_name     = (known after apply)

Do you want to perform these actions?
  Terraform will perform the actions described above.
  Only 'yes' will be accepted to approve.

  Enter a value: yes

module.cluster.random_pet.azurerm_kubernetes_cluster_dns_prefix: Creating...
module.base.random_pet.rg_name: Creating...
module.cluster.random_pet.ssh_key_name: Creating...
module.cluster.random_pet.azurerm_kubernetes_cluster_name: Creating...
module.cluster.random_pet.rg_name: Creating...
```

Рисунок 3.12 – Запуск створення Terraform-ом описаних у проекті ресурсів

По закінченню Terraform, окрім іншого, створить kubeconfig файл (Рис. 3.13) з необхідною конфігурацією для підключення до новоствореного кластеру за допомогою kubectl утиліти.

```
andrey@DESKTOP-8TM2AMP:/mnt/d/Coding/projects/master$ ll
total 216
drwxrwxrwx 1 andrey andrey  4096 Dec  7 14:49 ./
drwxrwxrwx 1 andrey andrey  4096 Dec  5 17:35 ../
drwxrwxrwx 1 andrey andrey  4096 Dec  7 09:53 .git/
-rwxrwxrwx 1 andrey andrey   127 Dec  7 08:48 .gitignore*
drwxrwxrwx 1 andrey andrey  4096 Dec  5 17:38 .terraform/
-rwxrwxrwx 1 andrey andrey  5471 Dec  6 09:36 .terraform.lock.hcl*
drwxrwxrwx 1 andrey andrey  4096 Dec  7 08:48 app/
-rwxrwxrwx 1 andrey andrey   492 Dec  6 13:47 env_vars.sh*
-rwxrwxrwx 1 andrey andrey  9813 Dec  7 14:46 kubeconfig*
```

Рисунок 3.13 – Зміст директорії з проектом після створення ресурсів

Найшвидший спосіб передати дані для підключення – це вказати шлях до конфігураційного файлу через змінну середовища «*KUBECONFIG*». Після виконання зазначеної вище операції ми можемо під'єтнатися до кластеру та перевірити, чи були створені kubernetes ресурси Terraform-ом (Рис. 3.14 та Рис. 3.15).

```
andrey@DESKTOP-8TM2AMP:/mnt/d/Coding/projects/master$ export KUBECONFIG=kubeconfig
andrey@DESKTOP-8TM2AMP:/mnt/d/Coding/projects/master$ kubectl get nodes
NAME                                STATUS   ROLES   AGE     VERSION
aks-agentpool-35937859-vmss000000   Ready    agent   7m30s   v1.27.7
aks-agentpool-35937859-vmss000001   Ready    agent   7m31s   v1.27.7
andrey@DESKTOP-8TM2AMP:/mnt/d/Coding/projects/master$
```

Рисунок 3.14 – Встановлення змінної середовища KUBECONFIG та перевірка доступності кластеру

```
andrey@DESKTOP-8TM2AMP:/mnt/d/Coding/projects/master$ k get ns
NAME              STATUS   AGE
default           Active   12m
kube-node-lease   Active   12m
kube-public       Active   12m
kube-system       Active   12m
master            Active   10m
andrey@DESKTOP-8TM2AMP:/mnt/d/Coding/projects/master$ k describe ns master
Name:         master
Labels:       env=master
              kubernetes.io/metadata.name=master
Annotations:  name: master
Status:       Active

No resource quota.

No LimitRange resource.
andrey@DESKTOP-8TM2AMP:/mnt/d/Coding/projects/master$ k get secrets -n master
NAME         TYPE                             DATA   AGE
docker-cfg   kubernetes.io/dockerconfigjson   1      7m38s
andrey@DESKTOP-8TM2AMP:/mnt/d/Coding/projects/master$ k get cm -n master
NAME               DATA   AGE
config             2      11m
kube-root-ca.crt   1      11m
andrey@DESKTOP-8TM2AMP:/mnt/d/Coding/projects/master$ k get sa -n master
NAME      SECRETS   AGE
default   0         11m
master    0         11m
andrey@DESKTOP-8TM2AMP:/mnt/d/Coding/projects/master$ k describe sa master -n master
Name:                master
Namespace:           master
Labels:              env=master
Annotations:         <none>
Image pull secrets:  docker-cfg
Mountable secrets:   <none>
Tokens:              <none>
Events:              <none>
andrey@DESKTOP-8TM2AMP:/mnt/d/Coding/projects/master$
```

Рисунок 3.15 – Перевірка наявності kubernetes ресурсів

3.3 Розгортання веб застосунку

Останні кроком даного проекту є розгортання веб застосунку, що був створений, зібраний у контейнер та завантажений до приватного репозиторію. Для досягнення поставленої мети нам окрім створених Terraform-ом kubernetes ресурсів залишилось створити лише безпосередньо Deployment з бажаною кількістю pod-ів та Service типу LoadBalancer, щоб kubernetes контролер cloud-controller створив балансир навантаження (load balancer) та надав IP адресу цьому сервісу [26].

Таким чином ми можемо створити бажаний kubernetes маніфест наступного змісту (розміщений у директорії проекту *./kubernetes_manifests/*):

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: web-app
spec:
  replicas: 2
  selector:
    matchLabels:
      app: web-app
  template:
    metadata:
      labels:
        app: web-app
    spec:
      serviceAccountName: master
      containers:
      - name: web-app
        image: andrstp/master:1.0
        ports:
        - containerPort: 5000
        envFrom:
        - configMapRef:
            name: config
---
apiVersion: v1
kind: Service
metadata:
  name: web-app-service
spec:
  selector:
    app: web-app
  type: LoadBalancer
  ports:
  - protocol: TCP
    port: 80
    targetPort: 5000
```

В даному випадку потрібно звернути увагу на такі поля як containerPort та envFrom у spec.template.spec.containers.web-app, де визначено порт на якому працює наш веб-сервер та посилання на ConfigMap ресурс у якому містяться змінні оточення, що будуть передані веб-серверу при запуску. Дані поля потенційно також можливо конфігурувати через ConfigMap у наступному релізі образу. Pod-и будуть створені з образом, який ми завантажили у приватний репозиторій, а доступ буде наданий через застосування ServiceAccount у якого є доступ до секрету з обліковими даними необхідними для отримання доступу до приватного репозиторію (Рис. 3.16).

```
andrey@DESKTOP-8TM2AMP:/mnt/d/Coding/projects/master$ k get secret -n master docker-cfg   -o jsonpath=
'{.data.\.dockerconfigjson}' | base64 -d | jq
{
  "auths": {
    "https://index.docker.io/v1/": {
      "auth": "",
      "email": "",
      "password": "",
      "username": ""
    }
  }
}
andrey@DESKTOP-8TM2AMP:/mnt/d/Coding/projects/master$
```

Рисунок 3.16 – Облікові дані приватного репозиторію у Secret-і.

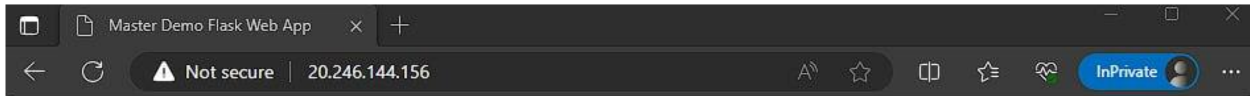
У частині маніфесту, що описує Service потрібно звернути увагу на поля spec.selector та spec.ports.targetPort, які мають відповідати відповідним полям у Deployment маніфесті.

Таким чином ми можемо застосувати цей файл з маніфестом щодо кластеру (Рис. 3.17).

```
andrey@DESKTOP-8TM2AMP:/mnt/d/Coding/projects/master$ k apply -f kubernetes_manifests/ -n master
deployment.apps/web-app created
service/web-app-service created
andrey@DESKTOP-8TM2AMP:/mnt/d/Coding/projects/master$ k get all -n master
NAME                           READY   STATUS    RESTARTS   AGE
pod/web-app-5978978c96-kqmjv   1/1     Running   0          33s
pod/web-app-5978978c96-rnzpf   1/1     Running   0          33s

NAME                      TYPE           CLUSTER-IP    EXTERNAL-IP     PORT(S)        AGE
service/web-app-service   LoadBalancer   10.0.18.171   20.246.144.156   80:31713/TCP   33s

NAME                      READY   UP-TO-DATE   AVAILABLE   AGE
deployment.apps/web-app   2/2     2            2           33s

NAME                                 DESIRED   CURRENT   READY   AGE
replicaset.apps/web-app-5978978c96   2         2         2       34s
andrey@DESKTOP-8TM2AMP:/mnt/d/Coding/projects/master$
```

Рисунок 3.17 – Створення ресурсів з маніфесту.

Тепер можна перевірити працездатність веб-серверу, перейшовши на адресу балансеру навантаження і пересвідчитись, що дата з ConfigMap була використана нашим веб-сервером (Рис. 3.18).



Hello there! I'm TEST from TESTLAND

Рисунок 3.18 – Веб сторінка веб застосунку

```
andrey@DESKTOP-8TM2AMP:/mnt/d/Coding/projects/master$ k describe cm config -n master
Name:         config
Namespace:    master
Labels:       env=master
Annotations:  <none>

Data
====
LOCATION:
----
testland
NAME:
----
test

BinaryData
====

Events:  <none>
andrey@DESKTOP-8TM2AMP:/mnt/d/Coding/projects/master$
```

Рисунок 3.19 – Зміст ConfgiMap-и, що використовується веб-застосунком

Використовуючи тип Deployment для розгортання нашого веб-застосунки у кластері, ми отримуємо усі переваги, що надає такий ресурс, а саме, швидка зміна розміру кількості працюючих веб-серверів, що може досягатися як в мануальному режимі (Рис. 3.20), так і в автоматичному – за допомогою створення та налаштування ресурсу типу HorizontalPodAutoscaler (Рис. 3.21).

```
andrey@DESKTOP-8TM2AMP:/mnt/d/Coding/projects/master$ k get pods -n master
NAME                        READY   STATUS    RESTARTS   AGE
web-app-5978978c96-7qj6x    1/1     Running   0          4m22s
web-app-5978978c96-kqmjv    1/1     Running   0          21m
andrey@DESKTOP-8TM2AMP:/mnt/d/Coding/projects/master$ k scale deploy web-app -n master --replicas=4
deployment.apps/web-app scaled
andrey@DESKTOP-8TM2AMP:/mnt/d/Coding/projects/master$ k get pods -n master
NAME                        READY   STATUS    RESTARTS   AGE
web-app-5978978c96-7qj6x    1/1     Running   0          4m40s
web-app-5978978c96-kqmjv    1/1     Running   0          21m
web-app-5978978c96-xv6nf    1/1     Running   0          5s
web-app-5978978c96-zvtw5    1/1     Running   0          5s
```

Рисунок 3.20 – Зміна кількості працюючих Pod-ів в ручному режимі

```
andrey@DESKTOP-8TM2AMP:/mnt/d/Coding/projects/master$ k autoscale deploy web-app --min=2 --max=4 --c
pu-percent=80 -n master
horizontalpodautoscaler.autoscaling/web-app autoscaled
andrey@DESKTOP-8TM2AMP:/mnt/d/Coding/projects/master$ k get pods -n master -w
NAME                        READY   STATUS    RESTARTS   AGE
web-app-5978978c96-7qj6x    1/1     Running   0          6m6s
web-app-5978978c96-kqmjv    1/1     Running   0          22m
web-app-5978978c96-xv6nf    1/1     Running   0          91s
web-app-5978978c96-zvtw5    1/1     Running   0          91s
^Candrey@DESKTOP-8TM2AMP:/mnt/d/Coding/projects/master$ k get hpa -n master
NAME      REFERENCE            TARGETS          MINPODS   MAXPODS   REPLICAS   AGE
web-app   Deployment/web-app   <unknown>/80%    2         4         4          26s
andrey@DESKTOP-8TM2AMP:/mnt/d/Coding/projects/master$ k top pods -n master
NAME                        CPU(cores)   MEMORY(bytes)
web-app-5978978c96-7qj6x    1m           18Mi
web-app-5978978c96-kqmjv    1m           18Mi
web-app-5978978c96-xv6nf    1m           18Mi
web-app-5978978c96-zvtw5    1m           18Mi
andrey@DESKTOP-8TM2AMP:/mnt/d/Coding/projects/master$
```

Рисунок 3.21 – Створення та налаштування ресурсу
типу HorizontalPodAutoscaler

Таким чином, було успішно розгорнуто простий веб-додаток на базі kuberentes кластеру та надано доступ до нього з зовні через створення ресурсу типу Service, де останній був представлений розподільником навантадення, що був створений хмарним контролем, що постачається AKS та Azure провайдером. В подальшому можливо здійснити розвиток проекту, за рахунок створення бази даних (наприклад, Cosmos DB) у провайдері та підключення її до кластеру, налаштування ingress контролеру та налаштування SSL сертифікату та реєстрація доменного імені веб-серверу. З точки зору кластеру, то можливо інсталювання додаткових інструментів моніторингу (зокрема, Prometeus та Grafana), та налаштування service mesh, зокрема застосовуючи такий відкритий продукт як Istio.

ВИСНОВКИ

В результаті виконання магістерської кваліфікаційної роботи було отримано наступні результати:

Проаналізовано концепцію хмарних обчислень під якою має розмітися забезпечення мережевого доступу до еластичного пулу спільних обчислювальних ресурсів на вимогу користувача, що здатний до швидкого масштабування. Було встановлено основні моделі хмарних обчислень (за режимом роботи) – приватні, публічні та гібридні; також було визначено основні моделі надання послуг та їх переваги та недоліки.

Розглянуто процес розгортання (споживання) ресурсів, що надаються хмарними провайдерами та основні способи здійснення такого процесу: 1) через графічний інтерфейс веб консолі, 2) інтерфейс командного рядка, 3) прикладний інтерфейс програмування та 4) застосовуючи інструменти IaC. Було розглянуто процес розгортання хмарних ресурсів на прикладі Terraform.

Обґрунтовано, що під контейнером розуміється деякий ізольований пакет ПЗ, що містить усі необхідні компоненти для його роботи незалежно (ізольовано) від середовища. Було продемонстровано, що технологія контейнеризації побудована з використанням таких особливостей ядра Linux як просторі імен (namespaces) та контрольні групи (cgroups), які забезпечують ізоляцію контейнерів та контроль (ізоляцію) ресурсів наданих контейнеру. Було проаналізовано інструменти контейнеризації на прикладі Docker та Podman (схожі риси та відмінності) та продемонстровано роботу з ними та життєвий цикл контейнеру.

Досліджено основні концепції управління контейнерами а мікросервісами, такі як оркестрація та хореографія. Було досліджено приципи роботи, архітектуру роботи оркестратора – Kubernetes та продемонстровано основні ресурси Kubernetes кластеру та принципи взаємодії з останнім.

Було успішно розгорнуто простий веб-додаток на базі AKS кластеру (створеного за допомогою Terraform та декларативного підходу до керування хмариними ресурсами) та надано доступ до нього з зовні через створення ресурсу типу Service, де останній був представлений розподільником навантадення, що був створений хмарним контролем, що постачається AKS та Azure провайдером. В подальшому можливо здійснити розвиток проекту, за рахунок створення бази даних (наприклад, Cosmos DB) у провайдері та підключення її до кластеру, налаштування ingress контролеру та налаштування SSL сертифікату та реєстрація доменного імені веб-серверу. З точки зору кластеру, то можливо інсталювання додаткових інструментів моніторингу (зокрема, Prometeus та Grafana), та налаштування service mesh, зокрема застосовуючи такий відкритий продукт як Istio.

ПЕРЕЛІК ДЖЕРЕЛ ПОСИЛАННЯ

1. Amazon ECR. What is Amazon Elastic Container Service? URL: https://docs.aws.amazon.com/AmazonECS/latest/developerguide/what-is-ecs.html

2. Bentaleb, O. *et al.* (2021) 'Containerization technologies: taxonomies, applications and challenges,' *The Journal of Supercomputing*, 78(1), pp. 1144–1181. https://doi.org/10.1007/s11227-021-03914-1

3. Cloud Computing Law (2021) in Oxford University Press eBooks. https://doi.org/10.1093/oso/9780198716662.001.0001. P. 36.

4. Computing Services Based on NIST SP 800-145, Special Publication 500-322' (2018) URL: https://nvlpubs.nist.gov/nistpubs/SpecialPublications/NIST.SP.500-322.pdf

5. Containerd. URL: https://containerd.io/

6. CRI: the Container Runtime Interface. GitHub. URL: https://github.com/kubernetes/community/blob/master/contributors/devel/sig-node/container-runtime-interface.md

7. CRI-O. URL: https://cri-o.io/

8. Directive (EU) 2016/1148 concerning measures for a high common level of security of network and information systems across the Union [2016] OJ L194. URL: http://data.europa.eu/eli/dir/2016/1148/oj

9. Docker. Docker pause. URL: https://docs.docker.com/engine/reference/commandline/pause/

10. Docker. Docker stop. URL: https://docs.docker.com/engine/reference/commandline/stop/

11. Docker. Dockerfile reference. URL: https://docs.docker.com/engine/reference/builder/

12. Docker. What is a Container? URL: https://www.docker.com/resources/what-container

13. Ferenc Hámori. The History of Kubernetes on a Timeline. URL: https://blog.risingstack.com/the-history-of-kubernetes/

14. Flexera, 'State of the Cloud Report 2020' (2020) URL: https://info.flexera.com/SLO-CM-REPORT-State-of-the-Cloud-2020

15. GCP. Kubernetes Engine API. URL: https://cloud.google.com/kubernetes-engine/docs/reference/rest/

16. Google Cloud Platform. Cloud interconnect overview. URL: https://cloud.google.com/network-connectivity/docs/interconnect/concepts/overview

17. HashiCorp Terraform. What is Terraform? URL: https://developer.hashicorp.com/terraform/intro

18. Hohn, A. (2022) *The Book of Kubernetes: A Complete Guide to Container Orchestration*. No Starch Press.

19. Huawei Technologies Ltd. Co, *Cloud computing technology*. 2023. doi: 10.1007/978-981-19-3026-3. P. 12.

20. Kubernetes Documentation. Overview. URL: https://kubernetes.io/docs/concepts/overview/

21. Kubernetes Documentation. Pods. URL: https://kubernetes.io/docs/concepts/workloads/pods/

22. Linux manual page. Namespaces (7). URL: https://man7.org/linux/man-pages/man7/namespaces.7.html

23. Marinescu Dan C. *Cloud Computing: Theory and Practice*. Second edition. Morgan Kaufmann; Elsevier Inc. 2018.

24. Michael Shirer, `Worldwide Public Cloud Services Revenues Surpass $500 Billion in 2022, Growing 22.9% Year Over Year, According to IDC Tracker` (2023) URL: https://www.idc.com/getdoc.jsp?containerId=prUS51009523

25. Microsoft Learn. Azure REST API reference. URL: https://learn.microsoft.com/en-us/rest/api/azure/

26. Microsoft Learn. Use a public standard load balancer in Azure Kubernetes Service (AKS). URL: https://learn.microsoft.com/en-us/azure/aks/load-balancer-standard?source=recommendations

27. National Institute of Standards and Technology, US Department of Commerce, Eric Simmon 'Evaluation of Cloud Computing Services Based on NIST SP 800-145, Special Publication 500-322' (2018) URL: https://nvlpubs.nist.gov/nistpubs/SpecialPublications/NIST.SP.500-322.pdf

28. Orchestration vs Choreography, which one should you use? The pros and cons. URL: https://blog.sparkfabrik.com/en/orchestration-vs-choreography

29. Orchestration vs Choreography: How to Pick the Right Integration Pattern in Azure. URL: https://techcommunity.microsoft.com/t5/azure-integration-services/orchestration-vs-choreography-how-to-pick-the-right-integration/m-p/3792149

30. Peterson, L.L., Baker, S. and Davie, B. (2022) Edge Cloud Operations: A Systems Approach. URL: https://ops.systemsapproach.org/index.html

31. Podman – Podman Documentation. URL: https://docs.podman.io/en/stable/markdown/podman.1.html

32. Podman. Get Started with Podman. URL: https://podman.io/get-started

33. Rani Osnat. A Brief History of Containers: From the 1970s Till Now. URL: https://blog.aquasec.com/a-brief-history-of-containers-from-1970s-chroot-to-docker-2016

34. RedHat. Understanding containers. URL: https://www.redhat.com/en/topics/containers

35. RedHat. What is Infrastructure as Code (IaC)? URL: https://www.redhat.com/en/topics/automation/what-is-infrastructure-as-code-iac

36. The Linux Kernel Archives. Cgroup v1. URL: https://docs.kernel.org/admin-guide/cgroup-v1/index.html

37. The Linux Kernel Archives. Cgroup v2. URL: https://docs.kernel.org/admin-guide/cgroup-v2.html

38. Walsh, D. (2023) *Podman in action: Secure, Rootless Containers for Kubernetes, Microservices, and More*. Simon and Schuster. P. 19.

39. Zhang Q, Liu L, Pu C, Dou Q, Wu L, Zhou W (2018) A comparative study of containers and virtual machines in big data environment. In: 2018 IEEE 11th International Conference on Cloud Computing (CLOUD). San Francisco, CA, pp 178–185. URL: https://doi.org/10.1109/CLOUD.2018.00030

40. Про хмарні послуги : ЗУ від 17.02.2022 № 2075-IX // База даних «Законодавство України» / Верховна Рада України. URL: https://zakon.rada.gov.ua/go/2075-20

ДОДАТОК А

Посилання на вихідний код програми

https://github.com/AndrStp/master