МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ

Чорноморський національний університет імені Петра Могили

Факультет комп'ютерних наук

Кафедра комп'ютерної інженерії

ДОПУЩЕНО ДО ЗАХИСТУ

Завідувач кафедри,
д-р техн. наук, проф.

_____ І. М. Журавська

« __ » _____ 2023 р.

КВАЛІФІКАЦІЙНА МАГІСТЕРСЬКА РОБОТА

Аналіз технологій контейнеризації та оптимізації розгортання масштабованого застосунку на платформі Kubernetes

Спеціальність «Комп'ютерна інженерія»
123 – КМР.1 – 605.21710508

Студент:

студент 6 курсу, групи 605,
спеціальності
123 Комп'ютерна інженерія

_____ С. С. Гонтаренко

Керівник:

канд. техн. наук, доцент

Я. М. Крайник

_____

Миколаїв 2023

ЗМІСТ

ПЕРЕЛІК УМОВНИХ ПОЗНАЧЕНЬ

КМР – кваліфікаційна магістерська робота

ОС – операційна система

ПІБ – прізвище, ім’я, по батькові

ФКН – факультет комп’ютерних наук

ЦП – центральний процесор

ЧНУ ім. Петра Могили – Чорноморський національний університет імені Петра Могили

API – Application Programming Interface

AWS – Amazon Web Services

EKS – Elastic Kubernetes Service

FTP – File Transfer Protocol

GCP – Google Cloud Platform

IaaS – Infrastructure as a Service

PaaS – Platform as a Service

RAID – Redundant Array of Independent Disks

SaaS – Software as a Service

ВСТУП

Актуальність теми. Більшість веб-застосунків мають бути доступними для запитів, оновлення налаштувань та збереження даних користувачів постійно. Існує дуже багато варіантів розгорнути застосунок в мережі. Найпопулярніший принцип — наявність фізичного простору для зберігання даних (віртуальний хостинг, віртуальний виділений сервер, фізичний сервер, VPS-хостинг або cloud провайдери).

Отже для того щоб розгорнути деякі застосунки в Інтернеті, необхідно перевірити наявність доменних імен, вибрати відповідний простір імен та зарезервувати його. Вибрати надійний хостинг, зареєструватися та оплатити послуги, завантажити всі необхідні файли, зазвичай, через FTP-клієнт, на сервер. Загалом послідовність кроків нескладна, але основна проблема пов’язана з надійністю. Такий варіант розгортання застосунку є досить нестабільним у використанні. Зі збільшенням функціоналу веб-застосунку та відповідно збільшенням навантаження на нього виникає проблема з масштабуванням. Також, є ризик втрати даних, оскільки звичайний хостинг не передбачає створення бекапів та резервного копіювання.

Тому розглянемо процес розгортання кластерного застосунку, що передбачатиме масштабування в майбутньому, з використанням контейнеризації та хмарних технологій. Контейнеризація — це майбутнє, і воно має багато переваг. Контейнеризація — це поняття упаковки коду з необхідними залежностями. Контейнери є портативними і можуть використовуватися в будь-якій інфраструктурі в будь-якому середовищі, яке підтримує технологію контейнерів, наприклад Kubernetes.

За допомогою служб Kubernetes ви отримуєте доступ до функцій, які допомагають збалансувати навантаження та спростити керування контейнерами на різних хостах. Ці можливості допоможуть вам створити ідеальну платформу, яка пропонує чудову масштабованість і продуктивність.

Ще одна чудова перевага Kubernetes полягає в тому, що платформа є найшвидше зростаючим проєктом у сфері відкритого коду після Linux. Дослідження показують, що кількість інженерів Kubernetes зросла на 67%, досягнувши приголомшливої кількості 3,9 мільйона інженерів Kubernetes до ваших послуг. Це число включає 31% усіх бекенд-розробників у світі, і це число збільшується на 4% щороку.

Мета дослідження: оптимізувати процес розгортання масштабованих застосуноків на базі платформі Kubernetes та проаналізувати технології контейнеризації.

Об’єкт дослідження: технології кластеризації контейнерів Kubernetes.

Предмет дослідження: платформа хмарних технологій — AWS; контейнеризований застосунок з можливістю масштабування.

Для досягнення поставленої мети потрібно вирішити такі **завдання:**

- проаналізувати технології контейнеризації;
- виділити типи хмарних сервісів;
- дослідити механізм роботи платформи Kubernetes в середовищі AWS;
- розгорнути в AWS кластер Kubernetes;
- порівняти методи розгортання застосунку в віртуальному хостингу та в кластері Kubernetes;
- написати поради по розгортанню кластерного застосунку.

Практичне значення отриманих результатів: розроблений метод розгортання масштабованого, кластерного застосунку може бути рекомендовано до застосування в навчальних цілях.

Апробація результатів магістерської роботи відбулася в рамках XXII Всеукраїнської науково-практичної конференції “Інформаційні технології та інженерія”. (м. Миколаїв, ЧНУ ім. Петра Могили).

Публікації. За результатами магістерської роботи опубліковано роботу у збірнику тез “Інформаційні технології та інженерія - 2023” [1].

РОЗДІЛ 1
АНАЛІТИЧНИЙ ОГЛЯД ПРИНЦИПІВ ТА МЕТОДІВ ХМАРНИХ ОБЧИСЛЕНЬ

Ця робота за мету ставить аналіз технологій контейнеризації та оптимізації розгортання масштабованого застосунку на платформі Kubernetes. Розглянемо різні методи та принципи хмарних обчислень. Буде визначена відмінність контейнеризації та віртуалізації.

Поняття хмарних обчислень

Хмарні обчислення — це загальний термін для всього, що передбачає надання розміщених послуг через Інтернет. Ці послуги поділяються на три основні категорії або типи хмарних обчислень: інфраструктура як послуга (IaaS), платформа як послуга (PaaS) і програмне забезпечення як послуга (SaaS).

Хмарні обчислення називаються так тому, що інформація, до якої здійснюється доступ, знаходиться віддалено в хмарі або віртуальному просторі. Компанії, які надають хмарні послуги, дозволяють користувачам зберігати файли та програми на віддалених серверах, а потім отримувати доступ до всіх даних через Інтернет. Приклади хмарних постачальників - це Amazon, Google, Microsoft та ін. Це означає, що користувачеві не потрібно перебувати в певному місці, щоб отримати доступ до нього, що дозволяє користувачеві працювати віддалено.

Хмарні обчислення знімають усі важкі завдання, пов’язані з обробкою даних, із пристрою, який ви носите з собою. Хмара може бути приватною або публічною. Загальнодоступна хмара продає послуги будь-кому в Інтернеті. Приватна хмара — це власна мережа або центр обробки даних, який надає розміщені послуги обмеженій кількості людей із певними налаштуваннями доступу та дозволів. Приватні чи публічні, мета хмарних обчислень полягає в

тому, щоб забезпечити простий, масштабований доступ до обчислювальних ресурсів та IT-послуг.

Хмарна інфраструктура включає апаратні та програмні компоненти, необхідні для належної реалізації моделі хмарних обчислень. Хмарні обчислення також можна розглядати як службові обчислення або обчислення на вимогу.

Мета хмарних обчислень та їх моделі

До появи платформ хмарних обчислень компанії переважно покладалися на сервери, бази даних, апаратне забезпечення, програмне забезпечення та інші периферійні пристрої, щоб вивести свій бізнес в Інтернет. Компанії були змушені купувати ці компоненти, щоб переконатися, що їхні веб-сайти чи програми досягли користувачів.

Крім того, підприємствам також потрібна була команда експертів для керування апаратним і програмним забезпеченням, а також моніторингу інфраструктури. Хоча цей підхід був практичним, він мав свої унікальні проблеми, як-от високу вартість налаштування, складні компоненти та обмежений простір для зберігання. Хмарні обчислення були створені для вирішення цих проблем [1].

1.2.1 Типи хмар

Публічна хмара є найпоширенішим типом хмари. У загальнодоступній хмарі ваші дані зберігаються на віддалених серверах, які об’єднані разом, щоб функціонувати як одна всеохоплююча мережа. Ці сервери належать не вам, а сторонньому постачальнику.

Коли ви використовуєте публічну хмару, сприймайте це як оренду місця на чужому сервері. «Хто», у кого ви орендуєте, — це постачальники хмарних послуг. Серед основних постачальників хмарних послуг є Amazon, Google і Microsoft. Ці компанії володіють і керують усім апаратним,

програмним забезпеченням та інфраструктурою, необхідною для роботи хмари.

Ключова відмінність публічної хмари полягає в тому, що нею може користуватися кожен. Все, що вам потрібно зробити, це зареєструватися в постачальника хмарних послуг, і ви отримаєте доступ до хмари. Це означає, що ваші дані використовують ті самі ресурси, що й усі інші клієнти цього хмарного постачальника.

З іншого боку, **приватна хмара** зарезервована для однієї організації чи підприємства. Усе апаратне та програмне забезпечення призначене для власника хмари [2].

Ви можете створити приватну хмару за допомогою власних ресурсів, наприклад центру обробки даних, або скористатися послугами стороннього постачальника. Якщо ви обираєте стороннього постачальника, певне апаратне та програмне забезпечення призначене виключно для вашої організації.

Деякі організації вибирають приватну хмару замість загальнодоступної для кращої гнучкості та безпеки. Завдяки приватній хмарі компанія може налаштувати хмару відповідно до своїх потреб. Це також забезпечує більший контроль, коли йдеться про безпеку, оскільки ресурси не є спільними.

Гібридна хмара — це комбінація загальнодоступних хмар, приватних хмар і локальної інфраструктури. Компанії, які використовують модель гібридної хмари, часто використовують загальнодоступну хмару для базових обчислювальних завдань, а більш конфіденційні дані зберігають у приватній хмарі або на локальному сервері. Для багатьох компаній модель гібридної хмари є найкращою з обох світів.

Компанії можуть контролювати всі свої дані, використовуючи потужність і масштабованість хмари.

1.2.2 Типи хмарних сервісів

Інфраструктура як послуга, відома як IaaS, є хмарним сервісом, який надає клієнтам і миттєву обчислювальну інфраструктуру через використання Інтернету. Використовуючи IaaS, третя сторона забезпечує і керує інфраструктурою, але ви можете придбати, встановити, налаштувати та керувати власним програмним забезпеченням. Це включає операційні системи, проміжне програмне забезпечення та програми.

Замість того, щоб купувати апаратне забезпечення, ви купуєте необхідні вам ресурси на вимогу, часто використовуючи модель оплати за використання. Цей вид послуг є найбільш гнучким, оскільки ви відповідаєте за все, крім інфраструктури. Ви матимете повний контроль над своєю системою, використовуючи ті самі технології та можливості, що й традиційний центр обробки даних, але без необхідності обслуговувати апаратне забезпечення [3].

Платформа як послуга (PaaS) — наступний тип сервісу хмарних обчислень, про який ми говоримо. За допомогою PaaS клієнти використовують хмарні компоненти для розробки та розгортання програмних застосунків. Як і IaaS, третя сторона керує вашою мережею, сховищем, серверами та віртуалізацією, тобто вам не потрібно про це турбуватися. Однак, на відміну від IaaS, ваш постачальник також керуватиме операційною системою, проміжним програмним забезпеченням і середовищем виконання в моделі PaaS.

PaaS — чудова модель обслуговування для користувачів, яким потрібно швидко створювати налаштовані програми на основі хмари. Він підтримує створення, тестування, розгортання, керування та оновлення веб-застосунків і мобільних застосунків, не турбуючись про налаштування або підтримку основної інфраструктури.

SaaS, або **програмне забезпечення як послуга**, є найменш гнучким і налаштовуваним хмарним сервісом, але також може похвалитися

найменшою кількістю необхідного керування та обслуговування (див. на рисунок 1.1). Він надає програмне забезпечення користувачам через Інтернет, як правило, на основі моделі підписки. SaaS є найбільш використовуваним варіантом для компаній на хмарному ринку, із загальними службами, включаючи G Suite і Office 365 [4].

За допомогою SaaS сторонній постачальник керуватиме всім: від вашої мережі, сховища й серверів до програм, даних і проміжного ПЗ. Це робить його надзвичайно зручним для користувачів, але його також важко вказати для організації.

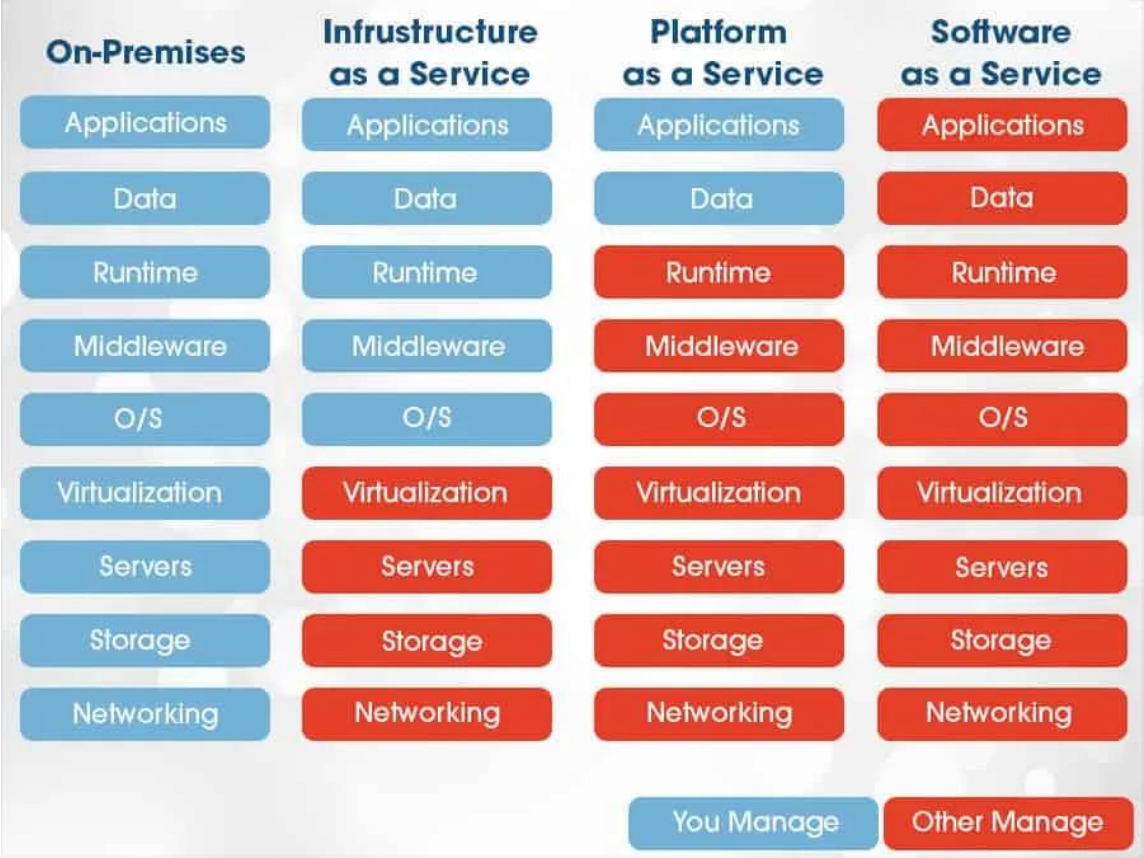


Рисунок 1.1 – Типи хмарних сервісів

Різниця віртуалізації та контейнеризації

Контейнеризація є формою віртуалізації. Віртуалізація спрямована на запуск кількох екземплярів ОС на одному сервері, тоді як контейнеризація запускає один екземпляр ОС із кількома просторами користувачів для

ізоляції процесів один від одного. Це означає, що контейнеризація має сенс для одного користувача хмари AWS, який планує запускати кілька процесів одночасно.

Контейнеризація досягається шляхом упаковки програмного коду, бібліотек, фреймворків та інших залежностей разом в ізольованому просторі користувача, який називається контейнером. Цей контейнер є портативним і може використовуватися в будь-якій інфраструктурі в будь-якому середовищі, яке підтримує технологію контейнерів, наприклад Docker і Kubernetes.

Архітектура мікросервісів передбачає розділення основних компонентів програми на окремі ізольовані компоненти. Оскільки компоненти можуть працювати незалежно один від одного, це зменшує ризик помилок або повного відключення служби.

Контейнер містить одну функцію для конкретного завдання або мікросервісу. Поділивши кожну окрему функцію програми на контейнер, мікросервіси підвищують стійкість і масштабованість корпоративних послуг.

Контейнеризація також дозволяє окремо оновлювати компоненти програми, не впливаючи на решту технологічного стеку. Це гарантує швидке застосування оновлень безпеки та функцій із мінімальним порушенням загальної роботи [5].

Віртуалізація — це метод, який може імітувати реальне фізичне обладнання (таке як ЦП, накопичувач і пам’ять) як віртуальну машину. Віртуальні машини — це емульовані системи. Вони можуть виконувати всі функції справжнього фізичного комп’ютера з повною операційною системою. Вони також можуть містити стек програмного забезпечення - поєднання апаратних і програмних пакетів дає нам повністю робочий знімок обчислювальної системи.

Віртуалізація неможлива без гіпервізора. Гіпервізор — це рівень програмного забезпечення або вбудованого програмного забезпечення, який

дозволяє кільком операційним системам працювати паралельно, використовуючи ті самі фізичні ресурси сервера. Гіпервізор також дозволяє фізичній машині відокремити свою операційну систему та програми від апаратного забезпечення для створення віртуальних машин (див. на рисунок 1.2).

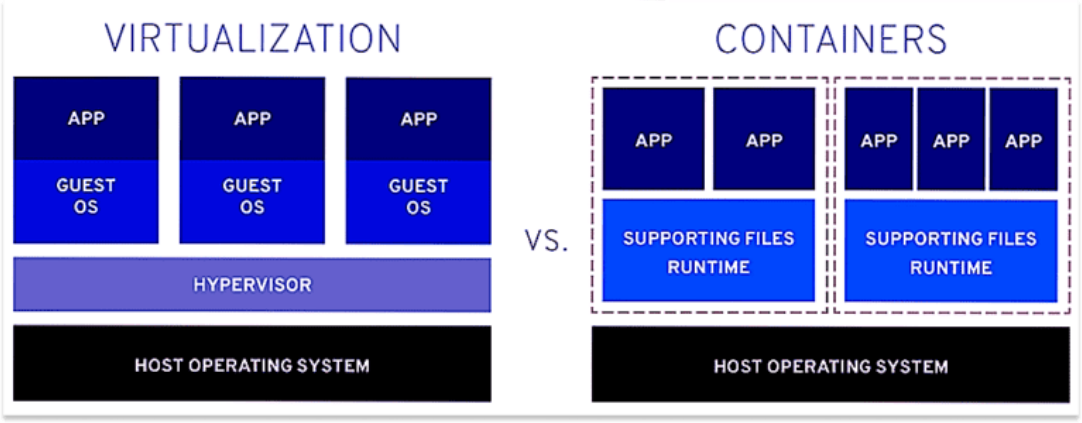


Рисунок 1.2 – Модель віртуалізації та контейнеризації

Віртуалізація дозволяє хмарним провайдерам максимально ефективно використовувати ресурси свого центру обробки даних. Не дивно, що багато корпорацій прийняли хмарну модель доставки для своєї локальної інфраструктури, щоб вони могли досягти максимального використання та економії коштів порівняно з традиційною IT-інфраструктурою, а також запропонувати кінцевим користувачам такий самий рівень самообслуговування та гнучкості.

Контейнеризація – це форма віртуалізації, яка імітує операційну систему вашого комп’ютера. Контейнери виконують функції, подібні до віртуальних машин, але не мають апаратної віртуалізації. Найпоширенішою контейнерною технологією є Docker, яка використовує загальнодоступне сховище. Прикладами залежностей, які може містити контейнер, є системні бібліотеки, сторонні пакети коду та інші програми на рівні операційної системи.

Крім того, контейнер програм дає змогу запакувати всі необхідні програми в портативне середовище. Ви також можете запускати кілька контейнерів одночасно.

Особливості контейнеризації:

– контейнери виглядають як звичайна Linux-система. Сторонні застосунки можуть запускатися в контейнерах без необхідності модифікації;

– користувач може змінювати конфігураційні файли і встановлювати будь-яке додаткове програмне забезпечення в контейнери;

– контейнери повністю ізольовані один від одного (файлова система, процеси, змінні sysctl);

– контейнери використовують динамічні бібліотеки, що значно економить пам'ять;

– контейнери не обмежені одним CPU і можуть використовувати усі CPU хоста.

Першою важливою відмінністю, на яку слід звернути увагу, є ізоляція: у той час як віртуалізація забезпечує повну ізоляцію від операційної системи хоста та інших віртуальних машин, контейнеризація пропонує легку ізоляцію від хоста та інших контейнерів, але не гарантує міцного кордону безпеки, як віртуальна машина.

Для однієї віртуальної машини ви можете використовувати віртуальний жорсткий диск (VHD) і спільний файловий ресурс блоку повідомлень сервера (SMB), щоб спільно використовувати сховище для кількох серверів. За допомогою контейнеризації ви використовуєте локальні диски для локального зберігання для кожного окремого вузла. Ви також можете використовувати SMB для спільного зберігання.

Однією з найвідоміших перешкод для ефективної віртуалізації є вартість. Звичайно, є розумна економія завдяки зменшенню закупівлі фізичних серверів, але з іншого боку, впровадження програмного забезпечення віртуалізації може коштувати дорого. Щоб працювати з

віртуальними машинами, вам потрібно буде заздалегідь покрити витрати на ліцензування та сервер — це може бути особливо складно для стартапів і малих підприємств. Оскільки віртуальні машини залежать від сторонніх постачальників, ризики безпеки дещо вищі, ніж зазвичай.

Переваги і недоліки хмарних обчислень

Хмарні обчислення не є підходом “все або нічого”. Є можливість вибрати хмарне сховище для своїх даних і використовувати стільки ресурсів, скільки потрібно для задоволення бізнес-вимог. Існуючі компанії можуть вибрати поступовий перехід, щоб заощадити витрати на інфраструктуру та адміністрування, тоді як нові компанії можуть почати свою діяльність відразу у хмарі.

Говорячи про переваги можна виділити зниження витрат. Основний фінансовий принцип полягає в тому, що прибуток приходить, коли ви заробляєте більше грошей, ніж витрачаєте. Це економічно ефективно – хмарні обчислення забезпечують модель ціноутворення на основі споживання, яка приносить багато переваг, зокрема: відсутність попередніх витрат на інфраструктуру; немає необхідності купувати та керувати дорогою інфраструктурою, яка може не використовуватися повністю. Можливість масштабувати ресурси лише за потреби або зменшувати їх, якщо вони не потрібні.

Хороші сервери коштуватимуть тисячі доларів лише за апаратне забезпечення. Крім того, є постійне обслуговування програмного та апаратного забезпечення. Також потрібна безпечна кімната для їх встановлення. Якщо у вас цього ще немає, вам потрібно буде створити його на місці. Сервери також потребують постійного охолодження для належної роботи, тому будьте готові до деяких жорстоких витрат на кондиціонування повітря.

Компанія може розоритися, купуючи ліцензії на програмне забезпечення високого класу.

Хмарні обчислення вирішують усі ці проблеми для бізнесу. Хмарний постачальник бере на себе всі клопоти, пов’язані з інфраструктурою, обслуговуванням і керуванням утилітами для серверів. Хмарні програми, зазвичай, становлять незначну частину вартості локально встановленого програмного забезпечення. Ви також отримуєте перевагу, сплачуючи лише за час або місце на сервері, які використовуєте.

З іншого боку, масштабування. Ви можете збільшити або зменшити кількість використовуваних ресурсів і послуг залежно від попиту та навантаження. Хмарні обчислення підтримують як вертикальне, так і горизонтальне масштабування залежно від потреб бізнесу.

Вертикальне масштабування, також відоме як «збільшення», це процес додавання ресурсів для збільшення потужності існуючого сервера. Приклади вертикального масштабування включають додавання більшої кількості процесорів або збільшення обсягу пам’яті.

Горизонтальне масштабування, також відоме як «масштабування в кількості», — це процес додавання більшої кількості серверів, які функціонують як одне ціле. Наприклад, є кілька серверів, які обробляють вхідні запити.

Масштабування можна виконувати вручну або автоматично на основі певних тригерів, таких як статистика використання ЦП або кількість запитів і ресурсів. Це може бути досягнуто всього за кілька хвилин.

По-друге безпека. Незважаючи на деякі гучні витоки хмарних даних, є багато аргументів, чому хмарні обчислення є більш безпечними, ніж внутрішні обчислення.

Прямо у верхній частині списку є те, що хмарні провайдери перебувають під більшим контролем та повинні відповідати встановленим

стандартам. Хоча всі компанії юридично зобов’язані захищати інформацію про клієнтів.

Дані, що зберігаються в хмарі, менше піддаються крадіжці. Інформацію легше вкрасти, якщо у вас є фізичний доступ до машини, на якій вона зберігається. Хмарні обчислення відокремлюють ваші дані від будь-яких потенційно незадоволених співробітників [6].

Найбільш очевидним аргументом є те, що постачальники хмарних послуг будуть постійно підтримувати протоколи безпеки та програмне забезпечення в актуальному стані, оскільки від цього залежить їхній бізнес. Цілком ймовірно, що більшість хмарних провайдерів мають штатних працівників, які спеціалізуються на цифровій/мережевій безпеці.

По-третє надійність. Якщо ви не інвестуєте в надлишковий запас незалежних дисків (RAID), усі ваші дані та серверні програми одразу стають недоступними.

Хмарні постачальники використовують резервування. Ваші дані не просто зберігаються на сервері. Вони зберігається на кількох серверах. Залежно від постачальника, дані навіть можуть зберігатися на серверах у кількох місцях. На випадок катастрофічного збою на серверній фермі.

Це означає, що жоден апаратний збій не завадить вашому бізнесу. Це також означає, що ви можете очікувати надзвичайної надійності з точки зору доступу до ваших даних або послуг. Більшість провайдерів навіть гарантують 99,99% безвідмовної роботи [7].

Але є також певні недоліки хмарних сервісів. Час простою є, мабуть, найбільшим недоліком хмарних обчислень. Мається на увазі не простій сервера, а про доступ до самого Інтернету. Поки у вас немає доступу до Інтернету, ви не можете нічого робити з хмарою.

Надійні плани мобільного передавання даних можуть тимчасово вирішити цю проблему. Послуги стільникового зв’язку часто залишаються життєздатними, коли доступ до Інтернету та навіть відключення

електроенергії. Звичайно, плани передачі даних обмежені, а мобільні пристрої мають обмежений час автономної роботи. Знову ж таки, якщо вимкнеться електрика, у вас, ймовірно, виникнуть серйозніші проблеми, ніж доступ до хмарних служб.

У зрілій галузі ви, зазвичай, маєте справу з одним із кількох відомих гравців, які пропонують перевірені часом надійні послуги. Хмарні обчислення – це молода галузь, у якій багато компаній змагаються за бізнес. Існує ймовірність, що у вашого хмарного провайдера закінчаться гроші, і він назавжди закриє свої двері.

Чим важливіше хмара для вашого бізнесу, тим більш руйнівними буде раптове припинення роботи постачальника. Ця проблема посилюється проблемою блокування хмарних постачальників, коли перехід від одного хмарного постачальника до іншого складний і дорогий.

Висновки до розділу 1

В наш час відбувається активний розвиток хмарних технологій. Все більше компаній поступово переходять на них. Основною передумовою стало зростаюча необхідність в інформаційних та обчислювальних ресурсах.

В розділі 1 проаналізовано методи та принципи хмарних обчислень. Хмарні технології вирішують дуже багато проблем. Те що стосується безпеки, доступності, надійності та вартості обслуговування. Також за допомогою хмарних технологій вирішуються проблема з потребою швидкого налаштування нових сервісів. Тому хмарні провайдери використовують та розробляють широкий набір послуг IaaS, SaaS, PaaS.

Значним і найцікавішим нововведенням у технологіях є можливість легкого й автоматичного масштабування застосунків, що необхідно для інформаційного простору, що активно змінюється. Ця тема буде розглянута в наступних розділах роботи.

РОЗДІЛ 2
ВЛАСТИВОСТІ СТРУКТУРИ ВЕБ-ЗАСТОСОНКУ

Характеристика веб-застосунку

Веб-застосунок є програмою зі своїм набором функцій, яка використовується через браузер як клієнт. Іншими словами, якщо для виконання бізнес-логіки програмі потрібне мережне з'єднання та браузер на стороні користувача, то цю програму можна вважати веб-додатком.

Усі дані зберігаються та оброблюються на серверах, до яких можна отримати доступ за допомогою браузера, мобільного застосунку тощо за допомогою протоколів HTTP або HTTPS, які забезпечують шифрування інформації на основі TLS.

Візуально веб-застосунок схожий на звичайну комп'ютерну програму, але він працює через Інтернет. Архітектура веб-застосунків показує взаємодію між додатками, базами даних та іншими проміжними елементами.

Основна мета архітектури веб-програми полягає в тому, щоб переконатися, що всі компоненти працюють правильно, взаємодіючи між собою. Кожна програма складається з двох частин: клієнтської (front-end) та серверної (back-end).

Інтерфейс - це частина програми, яку користувачі можуть бачити та взаємодіяти з нею. Клієнтський код реагує на дії користувача в інтерфейсі програми, тоді як серверна частина не видима користувачам, але відповідає за обробку логічних завдань та відповідей на HTTP-запити.

Наприклад, коли ви вводите свої дані у форму реєстрації, ви працюєте з клієнтською стороною, а коли ви натискаєте кнопку "Відправити" для реєстрації, серверна сторона обробляє ці дані та додає їх до бази даних.

При правильній роботі клієнтська та серверна сторони складають архітектуру програмного забезпечення веб-застосунку.

В типовому веб-застосунку можна виділити дві складові: клієнтську (frontend) та серверну (backend), що представляють відповідно програмний код на боці клієнта та сервера (див. рис. 2.1).

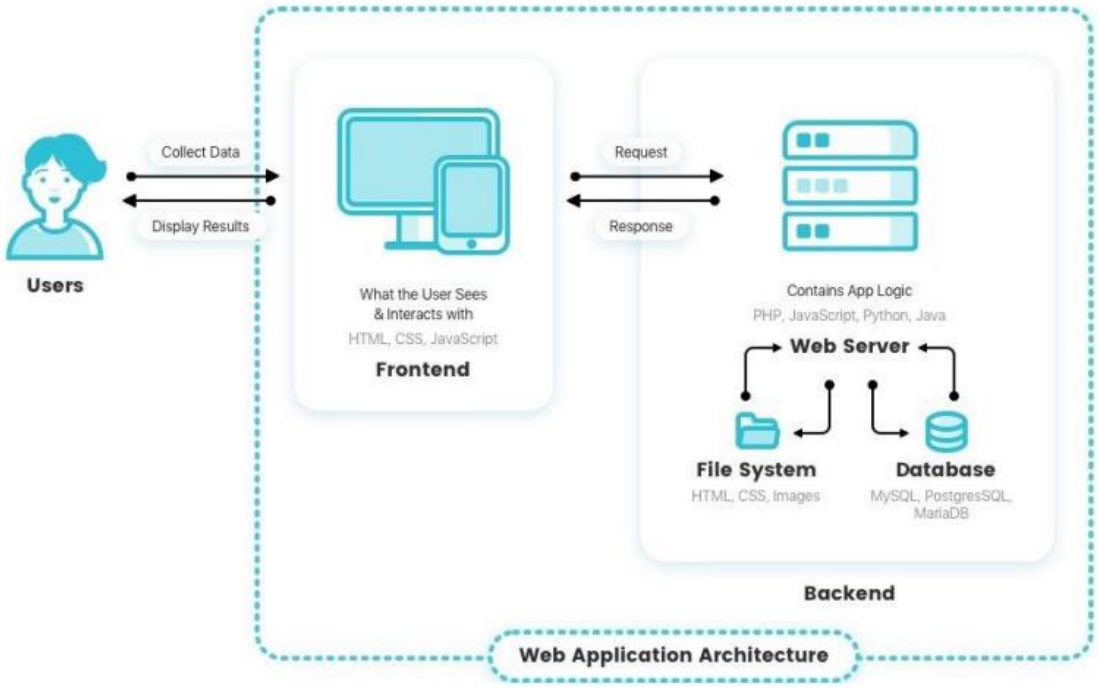


Рисунок 2.1 – Взаємодія користувача з веб застосунком

Для отримання більш глибокого розуміння архітектури веб-застосунку, необхідно детальніше ознайомитися з його компонентами та рівнями. Функції веб-програм поділяються на рівні, що дає можливість замінити або оновити кожен рівень незалежно від інших [8].

Базові компоненти веб-архітектури включають компоненти інтерфейсу користувача та структурні компоненти, які діляться на клієнтські та серверні. Компоненти інтерфейсу користувача включають всі елементи інтерфейсу, такі як журнали активності, інформаційні панелі, повідомлення, налаштування тощо. Вони є частиною макету інтерфейсу веб-програми.

Структурні компоненти складаються з клієнтської та серверної сторін. Клієнтський компонент розробляється з використанням HTML, CSS та JavaScript. Код запускається веб-браузером та перетворюється на інтерфейс, тому не потрібно налаштовувати операційну систему.

Серверний компонент будується на мовах програмування, таких як Java, .Net, Node.JS, Python тощо. Сервер складається з двох частин - логіки програми та бази даних. Логіка програми є центром керування веб-програмою, а база даних відповідає за зберігання інформації, наприклад, облікових даних.

Компоненти веб механізму

Існує чотири загальні рівні веб-застосунків, а саме:

- Рівень представленн (PL)
- Рівень обслуговування даних (DSL)
- Рівень бізнес-логіки (BLL)
- Рівень доступу до даних (DAL)

Рівень представлення (PL) відображає інтерфейс користувача та спрощує взаємодію з ним. Цей рівень включає компоненти інтерфейсу користувача, які візуалізують та показують дані користувачам, а також компоненти процесу користувача, які визначають взаємодію з користувачем. Основна мета рівня представлення - отримати вхідні дані, обробити запити користувачів, надіслати їх на рівень обслуговування даних та показати результати.

Шар бізнес-логіки (BLL) відповідає за належний обмін даними та визначає логіку бізнес-операцій та правил. Наприклад, вхід на сайт є прикладом рівня бізнес-логіки.

Рівень обслуговування даних (DSL) передає дані, оброблені на рівні бізнес-логіки, на рівень представлення. Цей рівень підтримує безпеку даних, ізолюючи бізнес-логіку клієнта [9].

Рівень доступу до даних (DAL) забезпечує спрощений доступ до даних, які зберігаються у постійних сховищах, таких як бінарні файли та файли XML. Крім того, рівень доступу до даних відповідає за операції CRUD - створення, читання, оновлення та видалення даних.

Зазвичай веб-програми складаються як мінімум з трьох основних компонентів:

- Клієнтська частина веб-програми, яка представляє собою графічний інтерфейс та відобража.
- Серверна частина веб-додатка - це програмне забезпечення або скрипт, що виконується на сервері та оброблює запити, які користувачі надсилають з браузерів. Зазвичай, серверна частина веб-додатка розробляється з використанням мови програмування PHP.

Кожен раз, коли користувач переходить за посиланням, його браузер надсилає запит на сервер. Сервер оброблює запит, запускаючи відповідний PHP-скрипт, який генерує веб-сторінку, описану мовою HTML, і відправляє її клієнту через мережу. Після цього браузер відображає збудовану сторінку. Поміж цими запитами та відповідями може бути багато посередників, які називаються проксі (proxy). Проксі виконують різноманітні операції і функціонують як шлюзи або кеш (див. рис. 2.2).

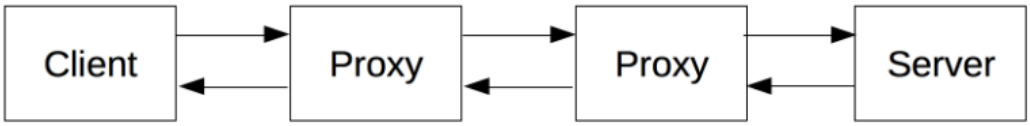


Рисунок 2.2 – Посередники між клієнтом і сервером

Шаблони створення веб застосунків

Шаблони проектування веб-застосунків, подібні до шаблонів проектування веб-сайтів та програмного забезпечення, можуть запропонувати безліч ефективних рішень.

Є багато літератури, як ефективніше та якісніше проектувати інтерфейси, як використовувати готові рішення на практиці та як їх подальше покращення. Це може допомогти досягнути небувалих висот у веб-програмуванні.

Монолітний застосунок є повністю замкнутим в контексті поведінки. Зазвичай, застосунок може взаємодіяти з іншими службами або сховищами даних, але весь функціонал реалізується в рамках одного процесу, і такий застосунок зазвичай розгортається як один елемент. Для горизонтального масштабування такий застосунок може дублюватися на декількох серверах або віртуальних машинах .

Термін "моноліт" використовується для опису архітектури, в якій всі компоненти зібрані в одну програму, що працює на одній платформі.

На рис. 2.3 зображено приклад монолітної архітектури, яка складається з декількох сервісів, що взаємодіють з базою даних. Клієнти, використовуючи браузер або мобільні пристрої, звертаються до веб-сервера для взаємодії з застосунком.

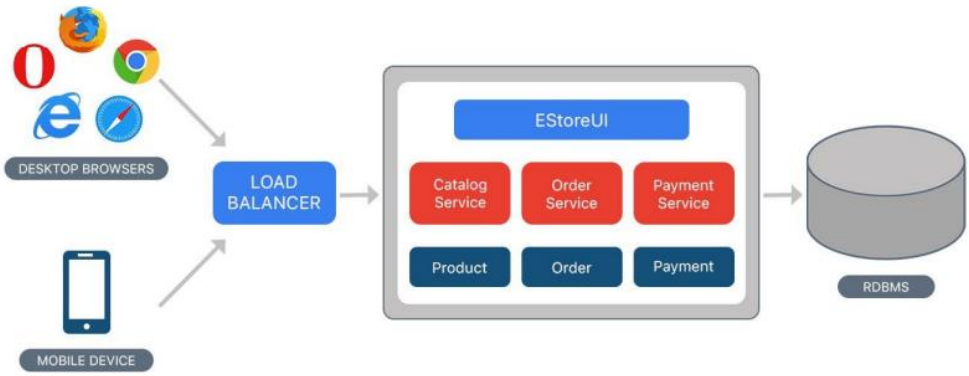


Рисунок 2.3 – Приклад монолітної архітектури

Мікросервісна архітектура - це підхід до розробки застосунків, де великий застосунок розбивається на набір незалежних, модульних служб або сервісів, які мають малу залежність від інших модулів або компонентів (див. рис. 2.4).

Рисунок 2.3 – Приклад мікросервісної архітектури

Замість того, щоб обмінюватись інформацією з однією централізованою базою даних, що відбувається в монолітній архітектурі, у мікросервісній архітектурі кожен сервіс має свою власну базу даних [10]. Такий підхід забезпечує краще розділення між компонентами програми, оскільки дозволяє кожному сервісу використовувати базу даних, що найкраще відповідає його потребам (наприклад, SQL або NoSql).

Безсерверна архітектура - це спосіб розробки та запуску програм та сервісів, що не вимагає від розробників управління інфраструктурою. Програма все ще працює на серверах, але керування цими серверами покладено на сторонню службу. Це звільняє від необхідності володіти, масштабувати та обслуговувати сервери для запуску програм, баз даних та засобів зберігання даних.

Модель безсерверних обчислень використовується для виконання комп’ютерного коду, і розробники можуть зосередитись на написанні функцій без необхідності управління системною інфраструктурою. Це також називається функцією як сервіс (FaaS), коли постачальник хмарних послуг забезпечує активацію та деактивацію функцій контейнерної платформи, перевірку безпеки інфраструктури, зниження витрат на обслуговування та покращення масштабованості (див. рис. 2.4).

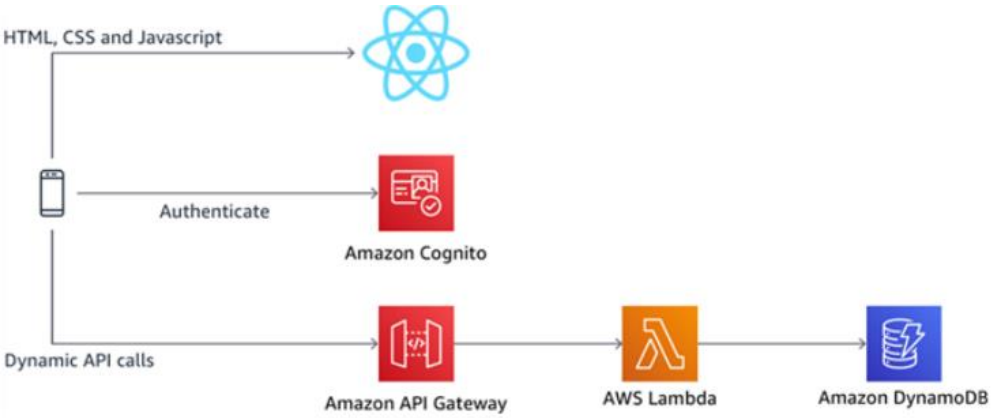


Рисунок 2.4 – Приклад безсерверної архітектури

Теорема CAP (теорема Брюєра) - це евристичне твердження, згідно з яким в будь-якій реалізації розподілених обчислень можливо забезпечити лише дві з трьох наступних властивостей (див. Таблицю 2.1):

– узгодженість даних (consistency) - коли всі дані в різних вузлах розподіленої системи в один момент часу однакові та не суперечать один одному,

– доступність (availability) - коли на будь-який запит до кожного вузла розподіленої системи надходить очікувана відповідь, проте немає гарантії, що всі відповіді в один момент часу будуть співпадати,

– стійкість до розподілу (partition tolerance) - коли поділ вузлів розподіленої системи на кілька ізольованих сегментів не призводить до неочікуваної роботи системи.

Таблиця 2.1

Характеристика CAP

Архітектура	Узгодженість	Доступність	Стійкість до розподілу
Моноліт	+	-	-
Мікросервіс	-	+	+
Безсерверна	-	+	+

Термін "вертикальне масштабування" означає збільшення потужності серверів, щоб збільшити кількість ресурсів, доступних для програмного

забезпечення. У цьому випадку перевага надається більш потужному обладнанню.

"Горизонтальне масштабування" - це збільшення кількості серверів, що об'єднані в кластер або групу процесів за рахунок операційної системи. Таблиця 2.2 показує, який тип масштабування допустимий для різних видів архітектури.

Таблиця 2.2

Характеристика по типу масштабування

Архітектура	Вертикальне	Горизонтальне
Моноліт	+	-
Мікросервіс	+	+
Безсерверна	+	+

Висновки до розділу 2

Швидкі зміни та складність можуть бути причинами, які впливають на вибір архітектури мікросервісів. Можливість безперервно розробляти та розгортати великі, складні програми допоможе бізнесу активно рости. Навпаки, якщо немає чіткої предметної області, можна розпочати з моноліту, але варто модулювати послуги. Якщо вирішено розділити моноліт на кілька мікросервісів, це полегшить роботу. Безсерверна архітектура дозволяє уникнути необхідності виділяти ресурси, масштабувати та підтримувати сервери для виконання програм, тож можна зосередитися на бізнес-вимогах та розробці застосунків.

РОЗДІЛ 3
ПЛАТФОРМИ ТА СЕРВІСИ ХМАРНИХ ОБЧИСЛЕНЬ

Docker – основний інструмент для контейнеризації

Docker — це платформа контейнеризації та середовище виконання, а Коли Docker був представлений у 2013 році, він приніс нам сучасну еру контейнерів і започаткував обчислювальну модель на основі мікросервісів. Оскільки контейнери не залежать від власної операційної системи, вони сприяють розробці слабозв’язаних і масштабованих мікросервісів, дозволяючи командам декларативно пакувати застосунок, його залежності та конфігурацію разом як образ контейнера.

Однак у міру ускладнення застосунків для зберігання контейнерів, розподілених між численними серверами, виникли проблеми, зокрема: як координувати та планувати кілька контейнерів, як увімкнути зв’язок між контейнерами, як масштабувати екземпляри контейнерів тощо. Kubernetes було представлено як спосіб вирішення цих проблем [11].

Контейнери — це тип технології віртуалізації, який дозволяє запускати кілька програм в одній операційній системі (ОС), ізолюючи кожну програму та її процеси один від одного. Ця ізоляція досягається за рахунок використання ОС, яка контролює розподіл ресурсів, таких як ЦП, пам’ять і диск для кожного процесу. Це означає, що кожна програма працює у своєму власному контейнері та не може отримати доступ до ресурсів інших контейнерів або основної ОС. Це забезпечує рівень безпеки та стабільності для програм, а також можливість запускати кілька програм на одному комп’ютері без втручання.

Контейнери пропонують вирішення цієї проблеми, дозволяючи програмам працювати в ізольованих середовищах, які можуть спільно використовувати базову операційну систему та ресурси сервера. Це означає,

що кожна програма має власні виділені ресурси, які можна легко збільшити або зменшити за потреби, не впливаючи на інші програми.

Контейнери також забезпечують певний рівень абстракції від базової інфраструктури, полегшуючи переміщення програм з одного середовища в інше, будь то із середовища розробки до середовища виробництва або з одного сервера на інший. Це полегшує керування, розгортання та масштабування застосунків, зменшуючи ризик простою та забезпечуючи стабільну продуктивність.

Docker використовує архітектуру клієнт-сервер із простими командами та автоматизацією через єдиний API. Docker також надає набір інструментів, який зазвичай використовується для упаковки застосунків у незмінні образи контейнерів шляхом написання Dockerfile і виконання відповідних команд для створення образу за допомогою сервера Docker (див. рис. 3.1). Розробники можуть створювати контейнери без Docker, але платформа Docker полегшує це.

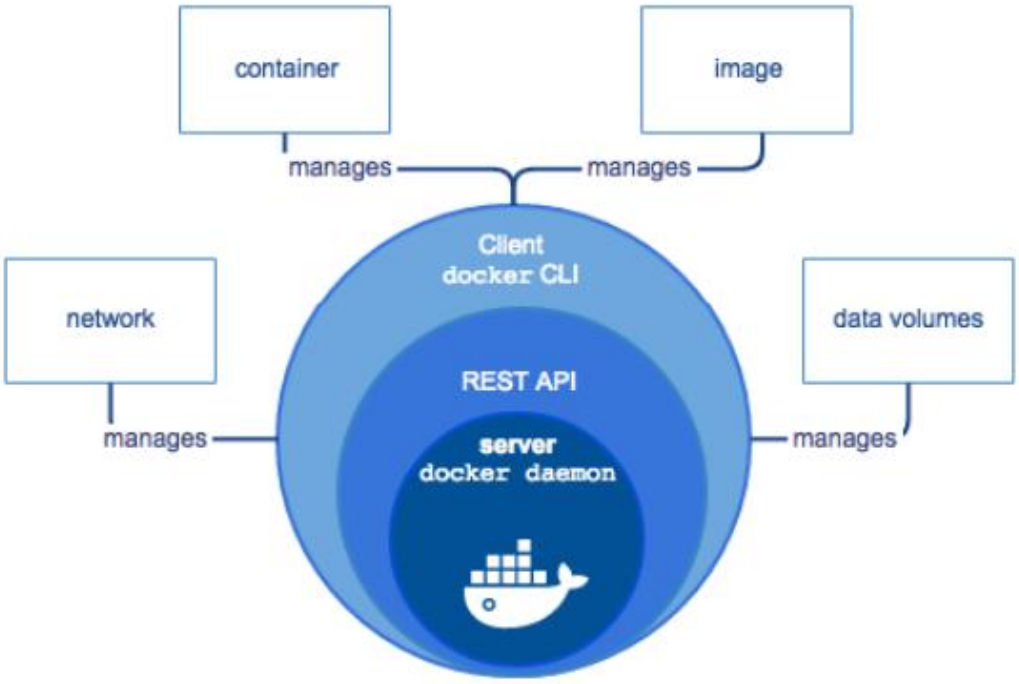


Рисунок 3.1 – Принцип роботи Docker

Docker-демон відповідає за створення, роботу та моніторинг контейнерів, а також для побудови та зберігання образів.

Демон Докер запускається командою docker daemon за що зазвичай відповідає ОС хоста. Користувач не взаємодіє з сервером на пряму, а використовує для цього клієнт.

Служба (демон) Docker використовує UNIX-сокет /var/run/docker.sock для вхідних з’єднань API. Власником даного ресурсу повинен бути тільки користувач root. Зміна прав доступу до даного ресурсу може привезти до небажаного доступу на хостову систему.

Якщо контейнери намагаються використовувати більше пам'яті, ніж доступно системі, може статись Out Of Memory Exception (OOME) і контейнер, або процес docker-демона, може бути знищений OOM ядром системи, що може привести до падіння додатку та втрати важливих даних.

Щоб цього не сталося, потрібно переконатись, що програма працює на хостах із достатньою оперативною пам’яттю. За допомогою команди dockerd можна запустити сервер Docker. Після введення команди, сервер недійшле інформаційні повідомлення про успішний запуск сервера.

Демон Docker використовує «драйвер виконання» для створення контейнера. За замовчуванням це власний драйвер Docker runc, однак є підтримка LXC.

Runc відповідає за функціональність:

– cgroups, які відповідають за управління використовуваними ресурсами контейнерів (тобто використання CPU та RAM).

namespaces, які відповідають за ізоляцію контейнерів та гарантують, що файлова система, ім’я хосту, мережеве оточення, список процесів та контейнери є відокремленими від інших частин вашої основної операційної системи.

Docker-клієнт – головний інтерфейс до Docker. Клієнт отримує команди від користувача і взаємодіє з докер-демоном. Клієнт Docker

використовується для спілкування з демоном Docker по протоколу HTTP. За замовчуванням відбувається через сокет домену Unix, а саме – IPC (Inter-Process Communication Socket, сокет міжпроцесорної взаємодії), але також можна використовувати сокет TCP, що дає можливість створювати віддалених клієнтів.

Так, при роботі з інтерфейсом командного рядка Docker (інтерфейс командного рядка Docker, CLI), в терміналі вводять команди, починаючи з ключів слова docker, що означає звернення до Docker-клієнта. Після чого Docker-клієнт використовує API Docker для відправки команд демону Docker.

Важливою частиною також є docker image. Образ – це пакет з усіма залежностями і інформацією, яка необхідна для створення контейнера. Образ включає в себе всі залежності (наприклад, платформа, бібліотеки), а також конфігурацію розгортання і виконання для середовища виконання контейнера.

Як правило, образ створюється на основі декількох базових образів, нашарованих один на одного в файлової системі контейнера. Після створення образ залишається незмінним.

Dockerfile – це текстовий файл, що містить інструкції по збірці образу Docker. Це схоже на пакетний сценарій, де перший рядок вказує базовий образ, з якого починається робота, а наступні інструкції встановлюють необхідні програми, копіюють файли і тд., для створення необхідного робочого середовища. Образ Docker складається з ряду шарів (layers) (див. рис. 3.2). Кожен шар являє собою інструкцію з Dockerfile

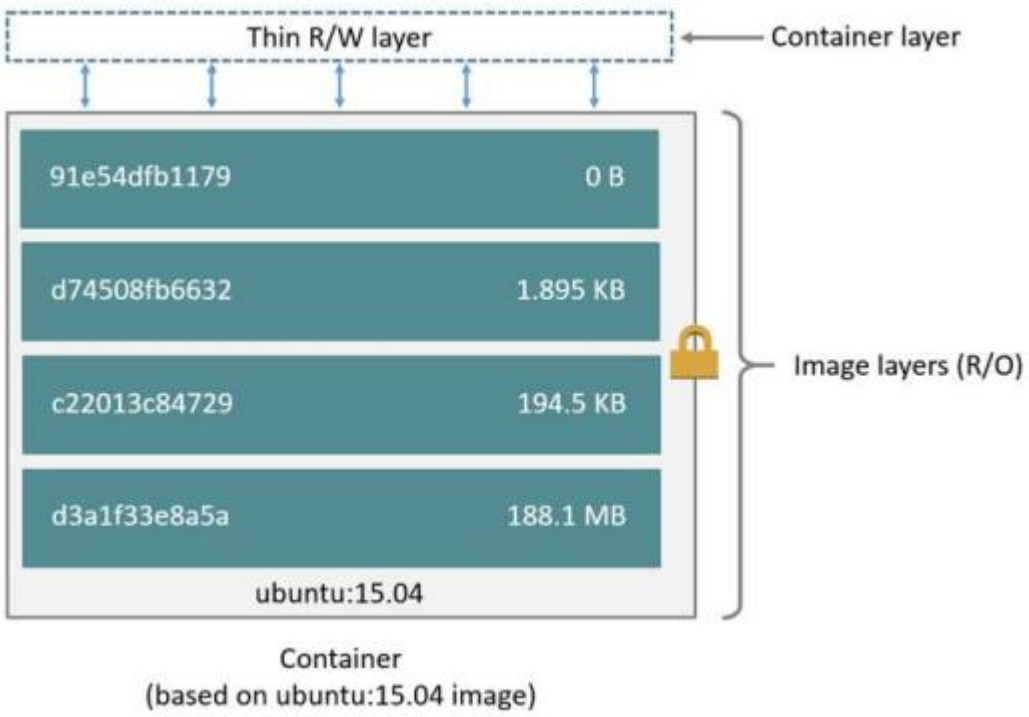


Рисунок 3.2 – Схема шарів image

Незважаючи на те, що Docker забезпечує ефективний спосіб упаковки та розповсюдження контейнерних програм, запуск і керування контейнерами в масштабі є проблемою лише з Docker. Координація та планування контейнерів на кількох серверах/кластерах, оновлення або розгортання застосунків із нульовим простоєм, а також моніторинг працездатності контейнерів – це лише деякі з міркувань, які необхідно враховувати (див. рис. 3.3).

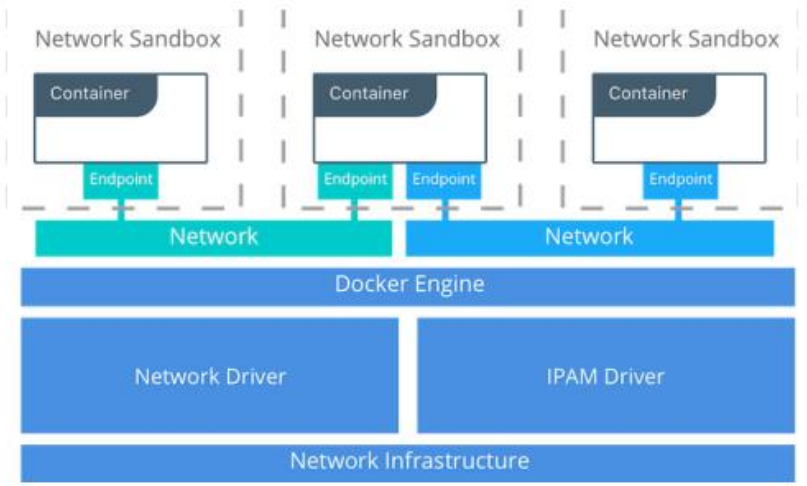


Рисунок 3.2 – Архітектура мережі Docker

Віртуальні машини (ВМ) — це сервери, які абстрагуються від базового апаратного забезпечення комп’ютера, що дозволяє кільком ВМ працювати на одному фізичному сервері або одній ВМ, що охоплює кілька фізичних серверів. Віртуальна машина запускає свій власний екземпляр операційної системи, і можна ізолювати будь-яку програму в межах своєї власної віртуальної машини, зменшуючи ймовірність того, що програми, які працюють на тому самому апаратному забезпеченні, будуть заважати одна одній. Віртуальні машини є більш ресурсоефективними, набагато простішими та економічнішими для масштабування, ніж традиційна інфраструктура [12].

Контейнери революціонізують концепцію віртуалізації, піднімаючи її на вищий рівень. На відміну від віртуальних машин, які мають власну окрему операційну систему, контейнери спільно використовують віртуалізоване ядро ОС і апаратне забезпечення, що сприяє більш ефективному використанню ресурсів. За допомогою контейнерів ви можете насолоджуватися тим самим рівнем ізоляції, масштабованості та паралельності, що й віртуальні машини, але з додатковими перевагами зменшення ваги та можливістю запускати більше програм на меншій кількості машин, як віртуальних, так і фізичних, з меншою кількістю екземплярів операційної системи.

Kubernetes

Kubernetes — це проєкт із відкритим вихідним кодом, який став одним із найпопулярніших інструментів оркестрації контейнерів; це дозволяє розгортати та керувати багатоконтейнерними програмами в масштабі. Хоча на практиці Kubernetes найчастіше використовується з Docker, найпопулярнішою платформою контейнеризації, він також може працювати з будь-якою контейнерною системою, яка відповідає стандартам Open

Container Initiative (OCI) для форматів зображень контейнерів і середовища виконання. І оскільки Kubernetes є відкритим вихідним кодом, з відносно невеликими обмеженнями на те, як його можна використовувати, його може вільно використовувати будь-хто, хто хоче запускати контейнери, переважно скрізь, де вони хочуть їх запускати — локально, у загальнодоступній хмарі або в обох випадках.

Kubernetes зародився як проект у Google. Він є спадкоємцем Google Borg, попереднього інструменту керування контейнерами, який Google використовував для себе, але не є прямим нащадком. Google відкрила Kubernetes у 2014 році, частково тому, що архітектури розподілених мікросервісів, які підтримує Kubernetes, спрощують запуск програм у хмарі. Google розглядає впровадження контейнерів, мікросервісів і Kubernetes як потенційне залучення клієнтів до своїх хмарних сервісів (хоча Kubernetes, звичайно, також працює з Azure та AWS). Наразі Kubernetes підтримує Cloud Native Computing Foundation, яка сама є під егідою Linux Foundation.

Кластер Kubernetes має одну або кілька control plane та один або кілька обчислювальних вузлів. Загалом control plane відповідає за керування кластером у цілому, розкриває інтерфейс прикладної програми (API) і планує запуск і завершення роботи обчислювальних вузлів на основі бажаної конфігурації.

Вузли кластера, керовані control plane, є машинами, які запускають контейнери. Кожен вузол запускає агент для зв’язку з площиною керування, kubelet — основним контролером Kubernetes. Кожен вузол також запускає механізм виконання контейнерів, наприклад Docker або rkt. Вузол також запускає додаткові компоненти для моніторингу, журналювання, виявлення служб і додаткові функції.

Площина керування — це нервовий центр, у якому розміщено компоненти архітектури кластера Kubernetes, які керують кластером. Він

також підтримує запис даних про конфігурацію та стан усіх об’єктів Kubernetes кластера.

Control palne Kubernetes постійно контактує з обчислювальними машинами, щоб забезпечити роботу кластера відповідно до налаштованих параметрів. Контролери реагують на зміни кластера, щоб керувати станами об’єктів і управляти фактичним, спостережуваним станом або поточним статусом системних об’єктів відповідно до бажаного стану чи специфікації.

Кілька основних компонентів складають площину керування: сервер API, планувальник, контролер-менеджер тощо. Ці основні компоненти Kubernetes забезпечують роботу контейнерів із достатньою кількістю необхідних ресурсів. Усі ці компоненти можуть працювати на одному основному вузлі.

Сервер Kubernetes API

Передній кінець площини керування Kubernetes, сервер API, підтримує оновлення, масштабування та інші види оркестровки життєвого циклу, надаючи API для різних типів програм. Клієнти повинні мати можливість доступу до сервера API з-за меж кластера, оскільки він служить шлюзом, підтримуючи оркестровку життєвого циклу на кожному етапі. У цій ролі клієнти використовують сервер API як тунель до модулів, служб і вузлів і автентифікуються через сервер API.

Планувальник Kubernetes

Планувальник Kubernetes зберігає дані про використання ресурсів для кожного обчислювального вузла; визначає, чи здоровий кластер; і визначає, чи слід розгортати нові контейнери, і якщо так, то де їх слід розмістити. Планувальник розглядає працездатність кластера, як правило, разом із вимогами до ресурсів модуля, наприклад ЦП або пам’яті. Потім він вибирає відповідний обчислювальний вузол і планує завдання, модуль або службу, враховуючи обмеження ресурсів або гарантії, локальність даних, вимоги до

якості сервісу, специфікації антиафінності та афінності та інші фактори (див. на рисунок 3.3).

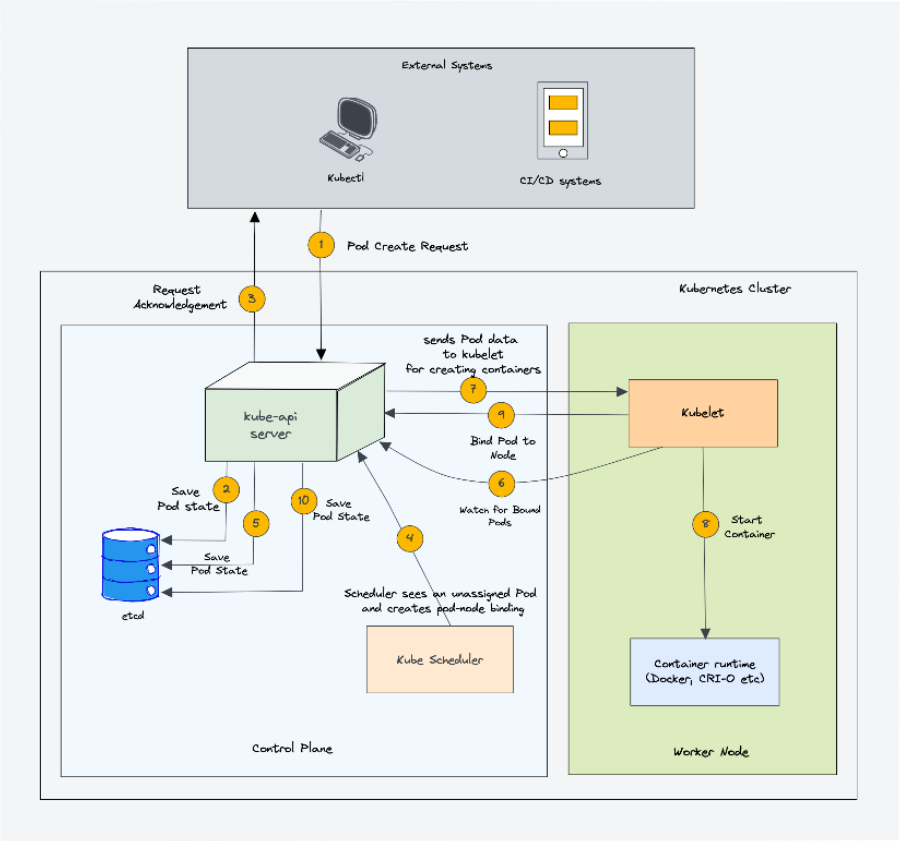


Рисунок 3.3 – Приклад роботи kube-scheduler

Менеджер контролера Kubernetes

В екосистемі Kubernetes є різні контролери, які керують станами кінцевих точок, токенів і облікових записів служб (простори імен), вузлів і реплікації (автомасштабування). Менеджер контролера — іноді його називають диспетчером хмарного контролера або просто контролером — це демон, який запускає кластер Kubernetes за допомогою кількох функцій контролера [13].

Контролер стежить за об’єктами, якими він керує в кластері, коли запускає цикли керування ядром Kubernetes. Він спостерігає за їхнім бажаним і поточним станом через сервер API. Якщо поточний і бажаний стани керованих об’єктів не збігаються, контролер вживає коригувальних заходів, щоб привести стан об’єкта до бажаного стану. Контролер Kubernetes також виконує основні функції життєвого циклу.

ETCD

Розподілена та відмовостійка etcd — це база даних із відкритим вихідним кодом, яка зберігає ключ-значення, у якій зберігаються конфігураційні дані та інформація про стан кластера. etcd можна налаштувати зовні, хоча він часто є частиною рівня керування Kubernetes (див. на рисунок 3.4).

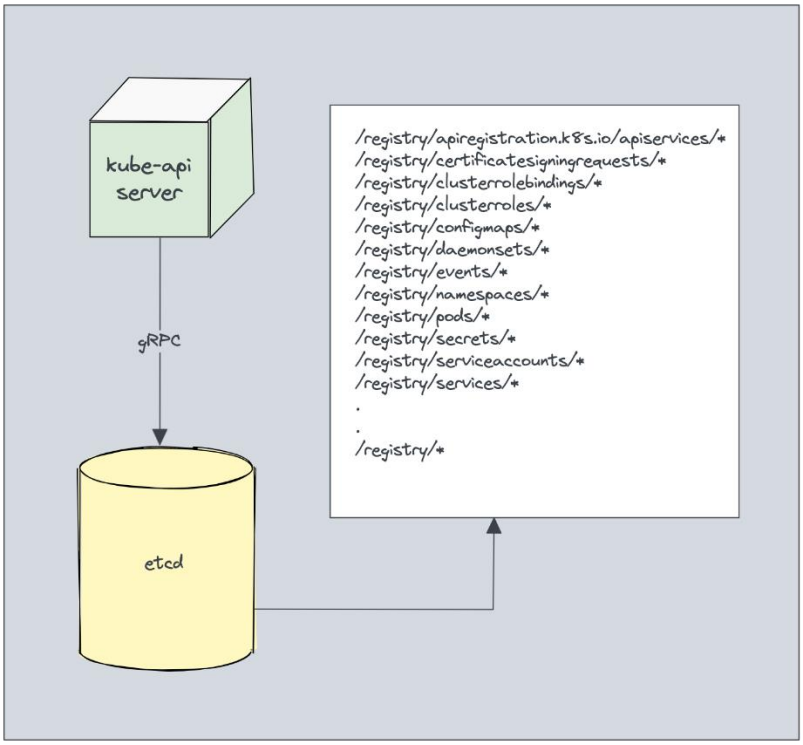


Рисунок 3.4 – ETCD

etcd зберігає стан кластера на основі консенсусного алгоритму Raft. Це допомагає впоратися із загальною проблемою, яка виникає в контексті реплікованих кінцевих автоматів і передбачає узгодження значень кількома серверами. Рафт визначає три різні ролі: лідера, кандидата та послідовника, і досягає консенсусу, обираючи лідера.

Таким чином etcd діє як єдине джерело істини (SSOT) для всіх компонентів кластера Kubernetes, відповідаючи на запити з рівня керування та отримуючи різні параметри стану контейнерів, вузлів і модулів. etcd також використовується для зберігання деталей конфігурації, таких як ConfigMaps, підмережі та секрети, а також дані про стан кластера.

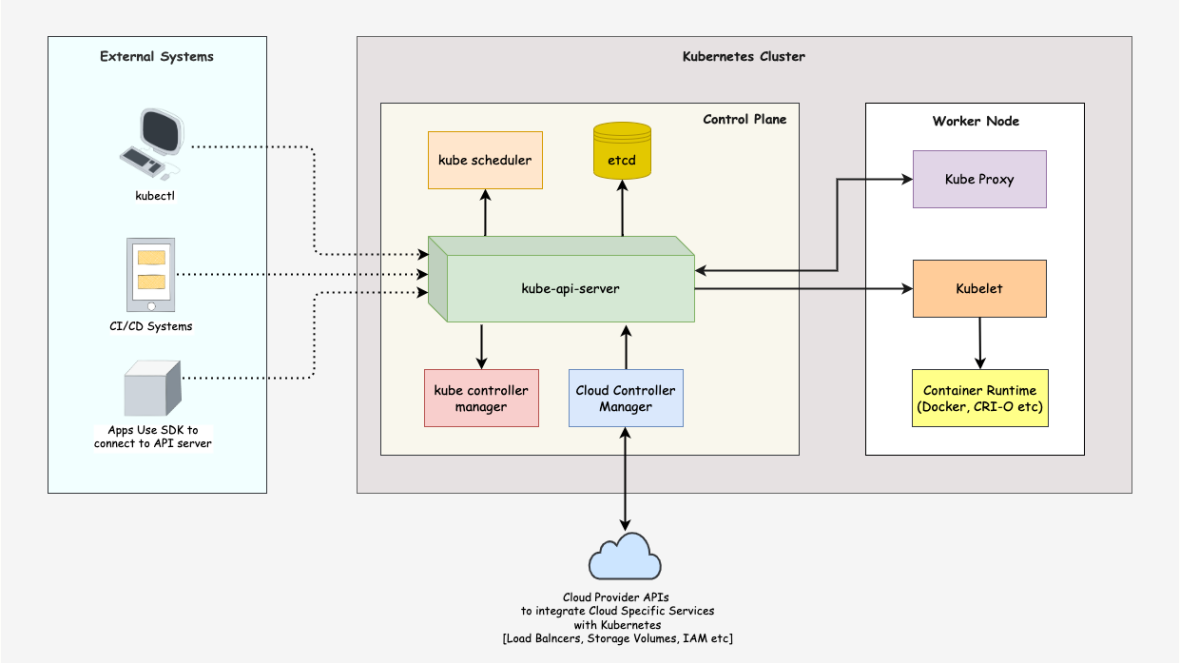


Рисунок 3.5 – Архітектура Kubernetes кластеру

Ось деякі компоненти кластера Kubernetes, які знаходяться в центрі уваги (див. на рисунок 3.5):

Nodes

Кластер Kubernetes повинен мати принаймні один обчислювальний вузол, хоча його може бути багато, залежно від потреби в потужності. Модули оркестровано та заплановано для роботи на вузлах, тому для збільшення потужності кластера потрібно більше вузлів.

Вузли виконують роботу для кластера Kubernetes. Вони з’єднують програми та мережеві, обчислювальні ресурси та ресурси зберігання.

Вузли можуть бути хмарними віртуальними машинами (ВМ) або оголеними серверами в центрах обробки даних.

Container Runtime Engine

Кожен обчислювальний вузол запускає та керує життєвими циклами контейнера за допомогою механізму виконання контейнерів. Kubernetes підтримує середовища виконання, сумісні з Open Container Initiative, такі як Docker, CRI-O та rkt.

Сервіс Kubelet

Кожен обчислювальний вузол включає kubelet, агент, який спілкується з площиною керування, щоб забезпечити роботу контейнерів у модулі. Коли рівень керування вимагає виконання певної дії у вузлі, kubelet отримує специфікації модуля через сервер API і виконує дію. Потім він гарантує, що пов’язані контейнери справні та працюють.

Сервіс Kube-проксі

Кожен обчислювальний вузол містить мережевий проксі, який називається kube-проксі, який полегшує роботу мережевих служб Kubernetes. Проксі-сервер kube або сам пересилає трафік, або покладається на рівень фільтрації пакетів операційної системи для обробки мережевих зв’язків як поза, так і всередині кластера.

Проксі-сервер kube працює на кожному вузлі, щоб забезпечити доступність служб для зовнішніх сторін і мати справу з окремими підмережами хостів. Він служить мережевим проксі та балансувальником навантаження на послуги на своєму вузлі, керуючи мережевою маршрутизацією для пакетів UDP і TCP. Насправді kube-proxy направляє трафік для всіх кінцевих точок служби.

Pods

Дотепер ми розглядали концепції, які є внутрішніми та зосередженими на інфраструктурі. На відміну від цього, pods є центральними для Kubernetes, оскільки вони є ключовою зовнішньою конструкцією, з якою взаємодіють розробники.

Под представляє один екземпляр програми та найпростішу одиницю в об’єктній моделі Kubernetes. Однак модулі є центральними та вирішальними для Kubernetes. Кожен пакет складається з контейнера або тісно пов’язаних контейнерів у серії, які логічно йдуть разом, разом із правилами, які контролюють роботу контейнерів [14].

Поди мають обмежений термін служби і врешті-решт гинуть після оновлення або зменшення масштабу. Однак, незважаючи на те, що модулі є

ефемерними, вони можуть запускати програми зі збереженням стану, підключаючись до постійного сховища.

Поди також здатні до горизонтального автомасштабування, тобто вони можуть збільшувати або зменшувати кількість запущених екземплярів. Вони також можуть виконувати поточні оновлення та розгортання Canary.

Поди працюють разом на вузлах, тож вони спільно використовують вміст і сховище та можуть зв’язуватися з іншими модулями через локальний хост. Контейнери можуть охоплювати кілька машин, тому й контейнери також можуть. Один вузол може запускати кілька контейнерів, кожен з яких збирає кілька контейнерів.

Под — це основна одиниця управління в екосистемі Kubernetes і діє як логічна межа для контейнерів, які спільно використовують ресурси та контекст. Відмінності у віртуалізації та контейнеризації пом’якшені механізмом групування модулів, який дає змогу запускати кілька залежних процесів разом.

Досягніть масштабування модулів під час виконання, створивши набори реплік, які забезпечують доступність, постійно підтримуючи попередньо визначений набір модулів, гарантуючи, що розгортання завжди виконує потрібну кількість. Служби можуть надавати зовнішнім або внутрішнім споживачам окремий пакет або набір копій.

Сервіси пов’язують із подами певні критерії, щоб уможливити їх виявлення. Блоки та служби пов’язані за допомогою пар ключ-значення, які називаються селекторами та мітками. Будь-який новий збіг між міткою пода та селектором автоматично виявлятиметься службою.

Порівняння Kubernetes в AWS, GCP, Microsoft Azure

Як говорилося вище, масштабні програмні рішення, що являють собою відокремленні мікросервіси, зручно розміщати в окремих контейнерах, та

використовувати системи оркестрації. Найбільш популярне рішення такого використання в хмарі є Kubernetes. Google Cloud Platform (GCP), Amazon Web Services (AWS) та Microsoft Azure займають ключову роль в управлінні архітектурою Kubernetes.

AWS має три контейнерні середовища: ECS, EKS і Fargate. ECS — найкращий варіант, якщо у вас мало досвіду роботи з контейнерами та ви вже працюєте з AWS для розміщення своїх служб. Це «контейнерне світло» з трьох варіантів. Розгортання в ECS називається «контейнери як послуга», і це вважається гарною відправною точкою для тих, хто хоче визначити, чи підходять вони для організації. Вам не потрібно нічого встановлювати з ECS. Ви просто дозволяєте службі автоматизувати ваші розгортання безпосередньо в хмарі за допомогою Amazon AWS CloudFormation [15].

Розміщений Kubernetes з AWS — це те, що робить його привабливим для розробників, які тільки вивчають усе контейнерне середовище. Якщо вам потрібно поекспериментувати з контейнерами, і ви не впевнені, що вони підходять у ваше середовище розробки, використання AWS і їхнього сервісу Kubernetes стане хорошою відправною точкою. Недоліком є те, що EKS вважається складним у налаштуванні та вимагає певного технічного досвіду роботи з контейнерами, але це також повністю масштабоване та настроюване рішення, яке дозволяє компанії контролювати Kubernetes і те, як він працює з локальною розробкою.

Якщо ви переважно працюєте в середовищі Windows, розгортання в Azure здається природним рішенням. Azure дещо повільніший під час розгортання, але це все ще покращення старих традицій ручного переміщення коду .NET із проміжної стадії чи розробки у робоче середовище. Це найновіший контейнерний сервіс на хмарному ринку, доступний лише з 2015 року, тому Microsoft продовжує вдосконалювати свій сервіс.

Незважаючи на те, що використання Azure звучить так, ніби це природно лише для екземплярів Windows, Azure насправді підтримує образи Linux, а це означає, що ви можете розгортати контейнери Linux на Azure, якщо операційну систему інстальовано з інформаційної панелі Azure. Це означає, що ви не обмежені лише Windows, але у вас є обмеження щодо гібридних контейнерів. Там, де AWS підтримуватиме гібридне розгортання, Azure обмежує вас Linux або Windows.

Недоліком Azure є те, що хоча служба AKS фактично з’явилася перед AWS EKS, Kubernetes набагато ширше запроваджується на AWS і GCP, і Azure зазвичай поступається як AWS, так і Google Kubernetes Engine (GKE), коли виходять останні версії. Однак якщо у вас уже є Azure і ви бажаєте попрацювати над ідеєю контейнерів, Azure спрощує розгортання та надає детальну аналітику, за допомогою якої ви можете визначити, чи це правильна платформа для вас.

Google є оригінальним творцем стандартів Kubernetes, тому робота з цією платформою дає вам перевагу майже в усіх аспектах. Будь-які нові версії та розгортання одразу доступні для вас, тоді як інші платформи повинні наздоганяти. Google чудово пропонує технології великих даних, машинного навчання та штучного інтелекту (ШІ), тож якщо це те, що вам подобається, то GCP – це той сервіс, з яким можна працювати [16].

Основна проблема GCP полягає в тому, що він не найпопулярніший для IaaS. Він не має хмарних пропозицій для малого бізнесу, які пропонують AWS і Azure, тому його платформа в цілому не приваблива для корпорацій, які хочуть інтегрувати хмару у свою внутрішню мережу. Немає інтеграції Active Directory, наприклад Azure або IAM, з AWS.

Таюлиця 3.1

Порівняльна характеристика хмарних платформ

Сервіс	Недоліки	Переваги
AWS	Безпека, надійність і	Потенційно дорожче,

	масштабованість	ніж інші варіанти
	Параметри для легких служб, щоб випробувати Kubernetes і контейнери, перш ніж перемістити локальне середовище в хмару	Важко використовувати новим розробникам, які не знайомі з контейнерними службами та Kubernetes
	Fargate дає розробникам можливість розгортати контейнери без розуміння серверної інфраструктури	
GCP	Оригінальний творець Kubernetes, тому впровадження нових функцій відбувається швидше	Не інтегрується з вимогами хмари IaaS
	Ідеально підходить для розробників, які хочуть працювати зі ШІ та великими даними	
Azure	Більш інтуїтивно зрозумілий для розробників Windows	Повільніше під час розгортання
	Supports Windows and Linux containers	Не підтримує гібридні контейнери

Однозначної відповіді, який сервіс краще немає. Все залежить від потреб. Якщо треба працювати зі ШІ, в цьому випадку краще підійде GCP. Якщо треба отримати більше додаткових програм – тоді вибір за Azure. AWS краще підходить для більшого інтегрування з хмарою.

Для подальшого розгортання масштабованого застосунку розглянемо платформу AWS, оскільки він більш надійний і краще масштабується.

Amazon Web Services

Платформа Amazon Web Services (AWS) надає понад 200 повнофункціональних послуг із центрів обробки даних, розташованих по всьому світу, і є найповнішою у світі хмарною платформою

Платформа, яка пропонує кілька операцій на вимогу, як-от обчислювальна потужність, зберігання бази даних, доставка контенту тощо, щоб допомогти компаніям масштабуватись і рости.

Для керування сервісами можна використовувати AWS Management Console (див. рис. 3.4).

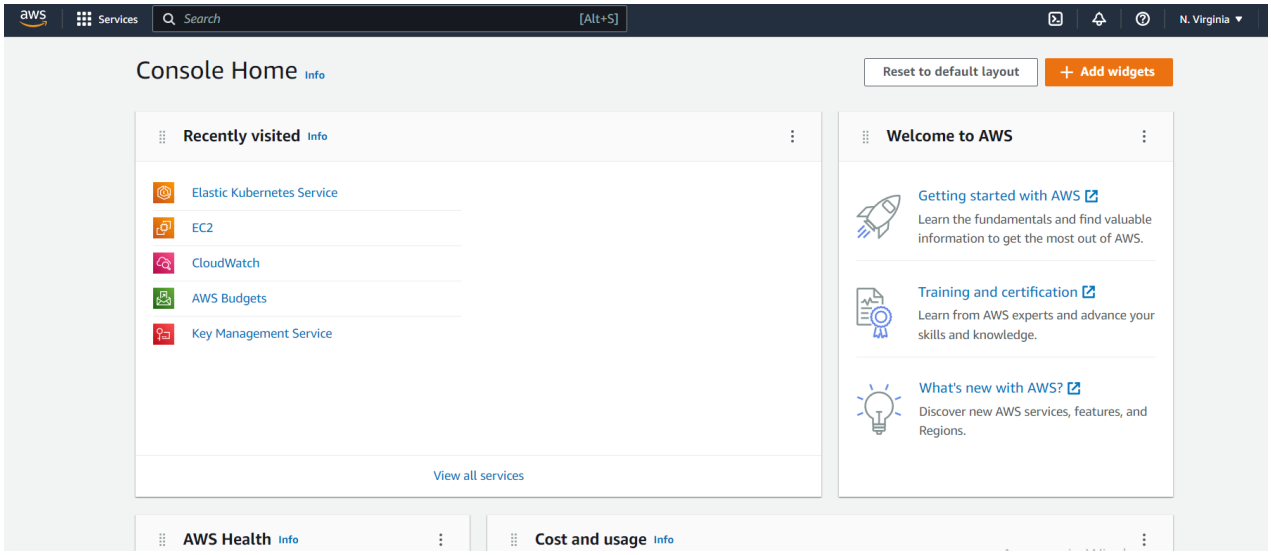


Рисунок 3.4 – AWS Management Console

Консоль керування AWS — це веб-застосунок, який містить і посилається на широкий набір сервісних консолей для керування ресурсами AWS. Коли ви вперше входите, ви бачите домашню сторінку консолі.

Домашня сторінка надає доступ до кожної сервісної консолі та пропонує єдине місце для доступу до інформації, необхідної для виконання завдань, пов’язаних із AWS. Це також дозволяє налаштовувати роботу консолі Home, додаючи, видаляючи та змінюючи порядок віджетів, таких як Recently visited, AWS Health, Trusted Advisor тощо.

Однією з головних переваг AWS є його гнучкість. З його інфраструктурою на вимогу, в основному немає обмежень щодо того, скільки ви можете використовувати. AWS надає вам різні варіанти.

Ви можете вибрати бажану ОС, платформу веб-застосунків, мову програмування та багато іншого. Платформа спрощує завантаження будь-якої служби чи програмного забезпечення у віртуальну екосистему. Це спрощує міграцію з існуючих платформ і полегшує розгортання нових. Окрім гнучкості, ви також маєте доступність.

Уявіть собі широкий спектр доступної вам інфраструктури та платформ за доступною ціною. Це те, що AWS надає та забезпечує. Отже, ви можете використовувати його за потреби, не турбуючись про накопичення витрат, коли ви цього не робите.

Можна подумати, що вартість усіх цих інфраструктур буде зашкалювати. Не так. У всякому разі, це цілком доступно. На відміну від будь-яких інших служб, які змусять вас платити за всі послуги - необхідні і непотрібні -, AWS цього не робить.

Натомість усі послуги, які пропонує компанія, є доступними та оплачуються залежно від використання. Немає авансових платежів або контрактів. Це максимально просто [17].

І це стане в нагоді малим підприємствам із обмеженим бюджетом, які прагнуть розвиватися, не платячи невеликих грошей за веб-послуги, і вважається ще однією з переваг AWS, якими ви можете скористатися.

Висновки до розділу 3

Kubernetes показав себе, як одна із найпопулярніших послух для оркестрації контейнерів та має попит в управлінні кластерами. AWS, GCP, Azure – найвідоміші клауд провайдери, які мають широкий спектр сервісів та можливість багатьох інтеграцій.

Kubernetes є чудовим рішенням для масштабування, підтримки різноманітних і відокремлених робочих навантажень із збереженням стану та без збереження стану, а також забезпечення автоматизованих відкатів і розгортань.

РОЗДІЛ 4
ОПТИМІЗАЦІЯ ПРОЦЕСУ РОЗГОРТАННЯ КЛАСТЕРНОГО ЗАСТОСУНКА В СИСТЕМІ KUBERNETES

Процес розгортання веб застосунка в мережі

Більшість веб-застосунків мають бути доступними для запитів, оновлення налаштувань та збереження даних користувача постійно. Існує дуже багато варіантів розгорнути застосунок в мережі, найпопулярніший принцип — наявність фізичного простору для зберігання даних (віртуальний хостинг, віртуальний виділений сервер, фізичний сервер, VPS-хостинг або cloud провайдери).

Отже для того щоб розгорнути деякі застосунки в Інтернеті, необхідно перевірити наявність доменних імен, вибрати відповідний простір імен та зарезервувати його. Вибрати надійний хостинг, зареєструватися та оплатити послуги, завантажити всі необхідні файли, зазвичай, через FTP-клієнт, на сервер. Загалом послідовність кроків нескладна, але основна проблема пов’язана з надійністю. Такий варіант розгортання застосунку є досить нестабільним у використанні. Зі збільшенням функціоналу веб-застосунку та відповідно збільшенням навантаження на нього виникає проблема з масштабу-ванням. Також, є ризик втрати даних, оскільки звичайний хостинг не передбачає створення бекапів та резервного копіювання.

Тому розглянемо процес розгортання кластерного застосунку, що передбачатиме масштабування в майбутньому, з використанням контейнеризації та хмарних технологій. Представимо порівняльну таблицю послуг, що є для хостингу та cloud провайдерів, для розгортання застосунку (табл. 4.1):

Таблиця 4.1

Порівняння можливостей cloud та хостингу

	Cloud	Хостинг
Масштабованість	Автоматично, в залежності від навантаження	Можливо, але складно в реалізації
Виділення додаткових ресурсів	Миттєво	Тривалий процес
Віртуалізація	Контейнери, розподілені між декількома фізичними серверами	Віртуальні машини на одному сервері
Ресурси процесора та пам’яті	Ізольовані	Розділювані
Вибір OS	Будь-яка	Пропонується хостом
Надійність	Висока	Непередбачувана
Вартість	Середня	Низька

Отже, нехай, наявний застосунок з мікросервісною архітектурою, кожен мікросервіс може бути написаний на різній мові, для зручності оперування в подальшому, потрібно написати для кожно-го сервісу Dockerfile. Dockerfile — це набір описаних команд для створення docker image. Кожна команда створює шар, тим самим збільшуючи загальний розмір image. Тобто потрібно дотримуватися певних рекомендацій для того, щоб зменшити розмір такого image, наприклад: не встановлювати непотрібні пакети, використовувати .dockerignore, використовувати multi-stage білди. З docker image створюється контейнер. Цей контейнер є портативним і може використовуватися в будь-якій інфраструктурі в будь-якому середовищі, яке підтримує технологію контейнерів, наприклад Kubernetes [18].

Створення кластера AWS EKS

Перш ніж розгорнути застосунок, потрібно створити кластер. Перед початком використання Amazon EKS потрібно встановити та налаштувати такі компоненти:

– AWS CLI – хоча можна використовувати AWS Console для створення кластера в EKS, AWS CLI простіше.

– Kubectl – використовується для зв’язку з API-сервером кластера.

– AWS-IAM-Authenticator – дозволяє автентифікацію IAM за допомогою кластера Kubernetes.

Наш перший крок — налаштувати нову роль IAM із дозволами EKS. Відкрити консоль IAM, вибрати «Ролі» ліворуч, а потім натиснути кнопку «Створити роль» у верхній частині сторінки. У списку служб AWS вибрати EKS (див. рис. 4.1).

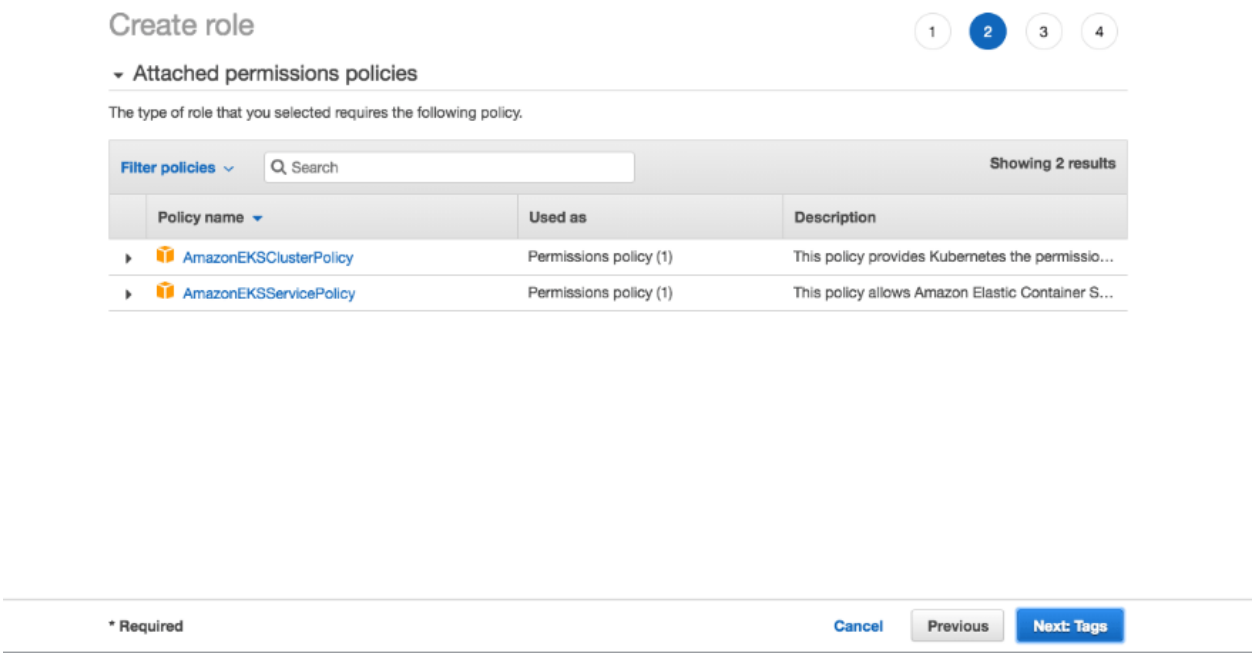


Рисунок 4.1 – Створення ролі

Ввести назву ролі (наприклад, eksrole) і натисніть кнопку «Створити роль (див. рис. 4.2).

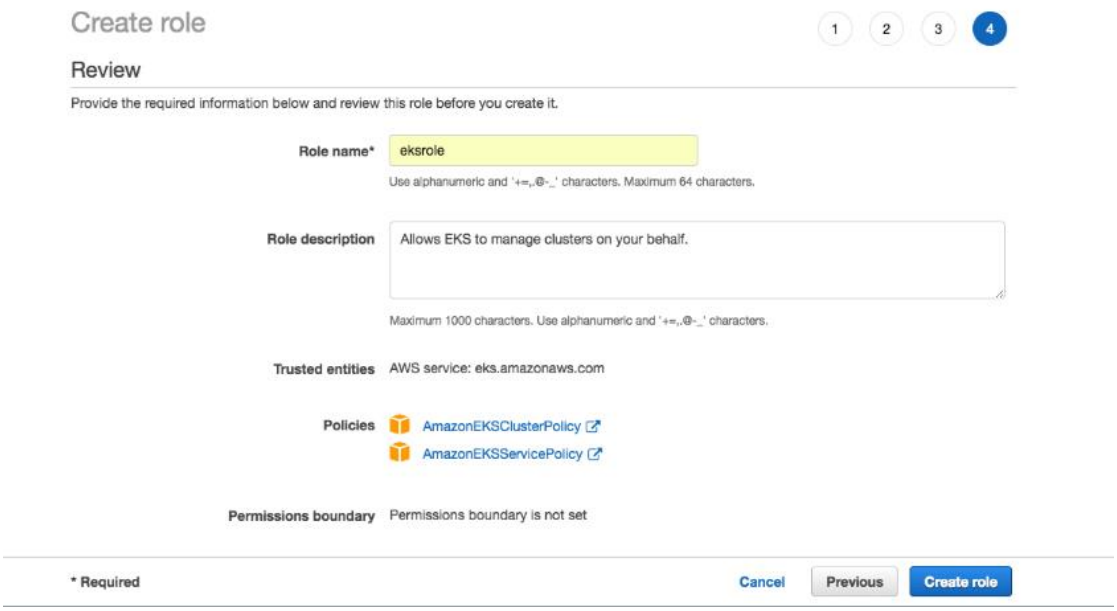


Рисунок 4.2 – Створення ролі

Далі створюємо окремий VPC — віртуальну приватну хмару, яка захищає зв’язок між робочими вузлами та сервером AWS Kubernetes API — для нашого кластера EKS [19]. Для цього скористаємося шаблоном CloudFormation, який містить усі необхідні компоненти EKS для налаштування VPC. Відкрити CloudFormation і натисніть кнопку «Створити новий стек» (див. рис. 4.3).

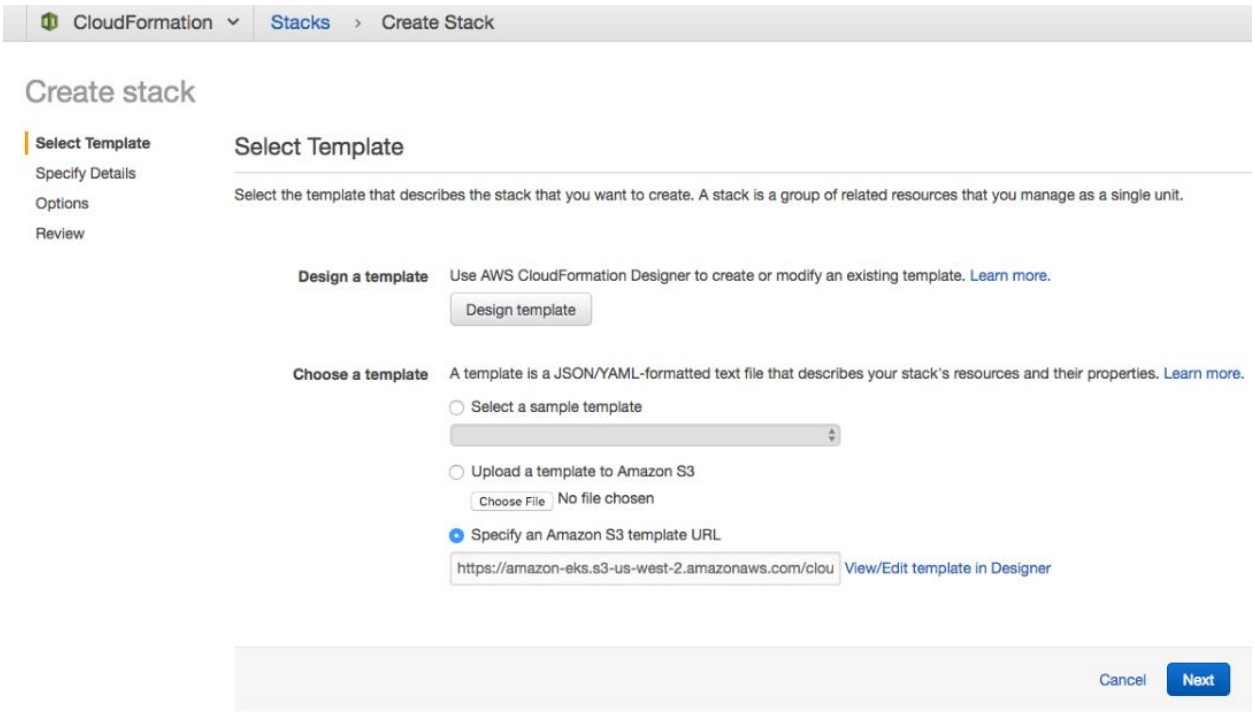


Рисунок 4.3 – Створення vpc

На сторінці перегляду просто натисніть кнопку «Створити», щоб створити VPC (див. рис. 4.4).

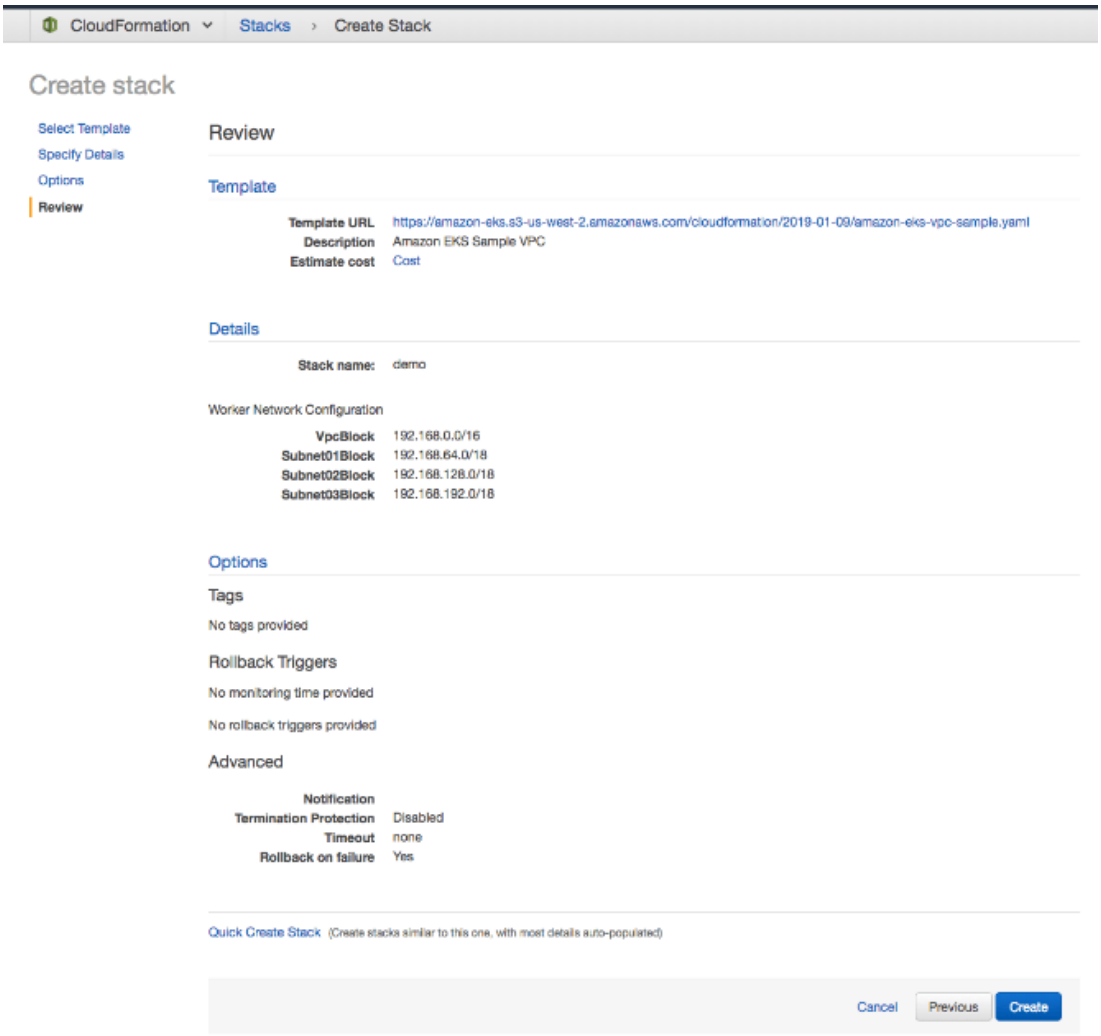


Рисунок 4.4 – Підтвердження параметрів vpc

CloudFormation почне створювати VPC. Після цього обов’язково треба запам’ятати різні створені значення — SecurityGroups, VpcId та SubnetIds.

Як згадувалося вище, будемо використовувати AWS CLI для створення кластера Kubernetes. Для цього виконуємо таку команду: *aws eks --region <region> create-cluster --name <clusterName> --role-arn <EKS-role-ARN> --resources-vpc-config subnetIds=<subnet-id-1>,<subnet-id-2>,<subnet-id-3>,securityGroupIds=<security-group-id>*

Для створення кластера потрібно близько 5 хвилин. Ви можете перевірити статус команди за допомогою цієї команди CLI: *aws eks --region us-east-1 describe-cluster --name demo --query cluster.status*

Або можна відкрити сторінку кластерів у консолі EKS (див. рис. 4.5):

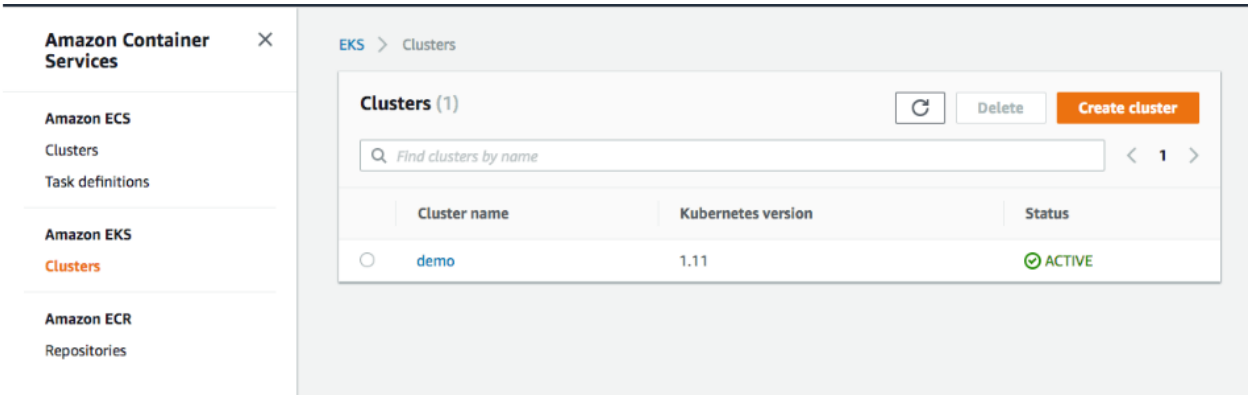


Рисунок 4.5 – Статус створення кластера в консолі

Коли статус зміниться на "Active", ми можемо продовжити оновлення нашого файлу kubeconfig інформацією про новий кластер, щоб kubectl міг з ним спілкуватися [20].

Для цього ми використаємо команду AWS CLI update-kubeconfig : *aws eks --region us-east-1 update-kubeconfig --name demo*

Можна перевірити конфігурацію виконавши команду: *kubectl get svc*

Коли вже налаштовано кластер і мережу VPC можна запускати робочі вузли Kubernetes. Для цього користуємося CloudFormation (див. рис. 3.6)

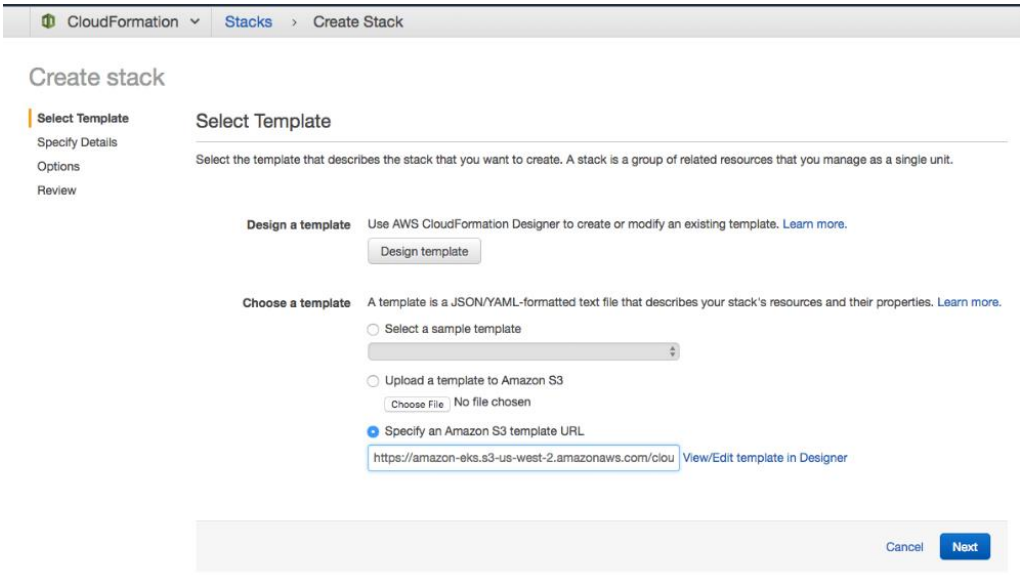


Рисунок 4.6 – Вибір темплейту для node group

Наступним кроком треба задати параметри node group (див. рис. 4.7)

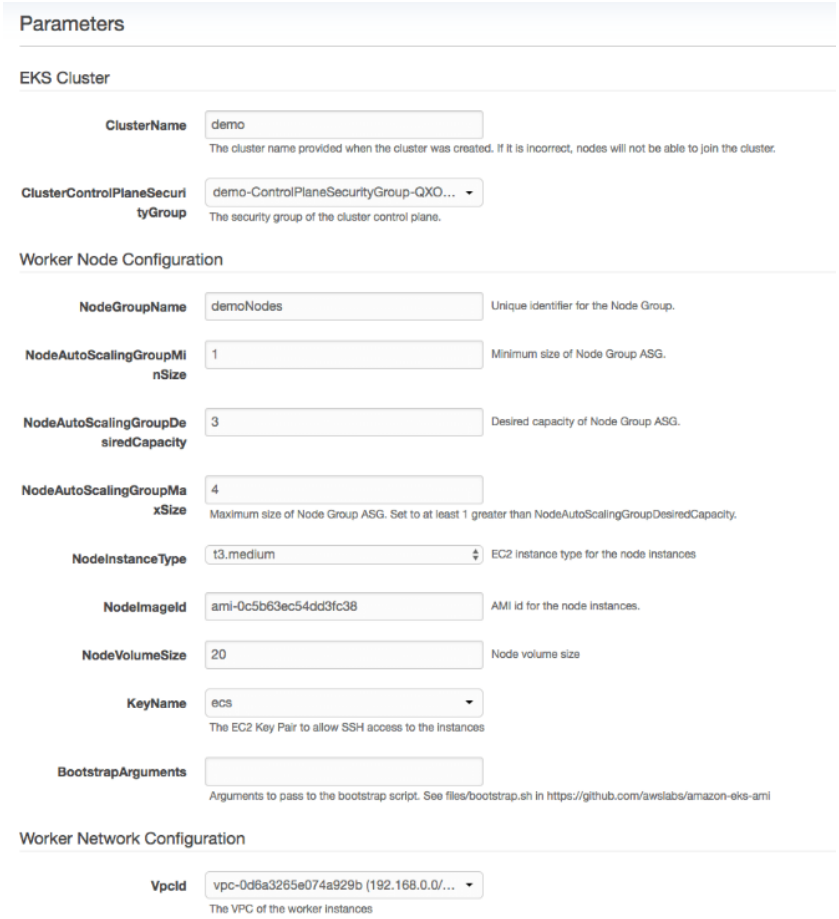


Рисунок 4.6 – Параметри для node group

CloudFormation створює робочі вузли з налаштуваннями VPC, які ми ввели — три нових екземпляри EC2.

Розгортання застосунка

Для налаштування ресурсів в Kubernetes використовуються фали під назвою маніфести. Вони декларативно допомагають описувати всі необхідні параметри у форматі yaml. Переходимо до розгортання застосунку. Для цього описуємо файл маніфесту ресурса ReplicaSet (див. рис. 4.8):

```yaml
apiVersion: apps/v1
kind: ReplicaSet
metadata:
  labels:
    app: node-nginx
  name: node-nginx
spec:
  replicas: 10
  selector:
    matchLabels:
      app: node-nginx
  template:
    metadata:
      labels:
        app: node-nginx
    spec:
      affinity:
        nodeAffinity:
          requiredDuringSchedulingIgnoredDuringExecution:
            nodeSelectorTerms:
            - matchExpressions:
              - key: purpose
                operator: In
                values:
                - stateless
      containers:
      - command:
        - yarn
        - start
        env:
        - name: CONFIG
          value: production
        image: 968199160822.dkr.ecr.us-east-2.amazonaws.com/k8s-web/node:a44afbc
        imagePullPolicy: IfNotPresent
```

Рисунок 4.8 – Опис маніфесту

Також треба зібрати імадж застосунку, який буде зберігатися в aws ecr. Для цього використовуємо описані конструкції в Dockerfile та виконуємо наступні дії. Автентифіковуємо свій клієнт Docker у реєстрі Amazon ECR. Командою aws ecr get-login-password. Під час передавання токена автентифікації до команди входу в докер використовуємо значення AWS для імені користувача та вказуємо URI реєстру Amazon ECR, у якому ви хочете автентифікуватися. У разі автентифікації в кількох реєстрах потрібно повторити команду для кожного реєстру. Позначаємо своє зображення реєстром Amazon ECR, репозиторієм і додатковою комбінацією імен тегів зображення для використання. Ім’я сховища має відповідати сховищу, яке створено для свого образу.

Щоб примінити конфігурацію виконуємо команду: *kubectl apply –f ./deploy.yml*

Далі потрібно налаштувати сервіс лоадбалансер, для доступу з світу до застосунка. Треба описати ресурс service, де буде зазначено дані порту та протокол маршрутизації. Застосовуємо зміни командо: *kubectl apply –f ./service.yml.*

Подальші налаштування треба, щоб застосунок був доступний за доменним ім’ям в мережі. Створюємо маніфест Ingress контроллер (див. рис. 4.9). Додаємо хост нейм, сертифікат. Вказуємо порти які прослуховуються та налаштовуємо правила внутрішної комуніукації.

```yaml
ingress:
  commonLabels:
    app: ingress
    project: frontend
  annotations:
    kubernetes.io/ingress.class: alb
    alb.ingress.kubernetes.io/scheme: internet-facing
    alb.ingress.kubernetes.io/certificate-arn: arn:aws:acm:us-east-2:968199160822:certificate/79b1a83d-33ce-4d45-bd18-77ddc6069f8d
    # Use this annotation (which must match a service name) to route traffic to HTTP2 backends.
    alb.ingress.kubernetes.io/listen-ports: '[{"HTTP":80}, {"HTTPS":443}]'
    alb.ingress.kubernetes.io/ip-address-type: ipv4
    alb.ingress.kubernetes.io/group.name: 'frontend'
    alb.ingress.kubernetes.io/load-balancer-name: 'eks-frontend'
    alb.ingress.kubernetes.io/actions.ssl-redirect: '{"Type": "redirect", "RedirectConfig": { "Protocol": "HTTPS", "Host": "my-apps.xyz", "Port": "4
    alb.ingress.kubernetes.io/healthcheck-protocol: HTTP
    alb.ingress.kubernetes.io/healthcheck-path: /health-check
    alb.ingress.kubernetes.io/healthcheck-interval-seconds: '20'
    alb.ingress.kubernetes.io/healthcheck-timeout-seconds: '8'
    alb.ingress.kubernetes.io/success-codes: 200,301,302,308
  tlsEnabled: true
  hosts:
  - my-apps.xyz
  rules:
  - host: my-apps.xyz
    http:
      paths:
      - path: /
        pathType: Prefix
        backend:
          service:
            name: ssl-redirect
            port:
              name: use-annotation
```

Рисунок 4.9 – Ingress controller

Останнім кроком потрібно налаштувати HPA. Тепер у нас є зразок програми? як частина нашого розгортання, а служба доступна через порт 80. Щоб масштабувати наші ресурси, ми використовуватимемо HPA для збільшення масштабу, коли трафік зростає, і зменшення ресурсів, коли трафік зменшується (див. рис. 4.10).

```yaml
apiVersion: autoscaling/v1
kind: HorizontalPodAutoscaler
metadata:
 name: hpa
spec:
 scaleTargetRef:
   apiVersion: apps/v1
   kind: ReplicaSet
   name: node
 minReplicas: 1
 maxReplicas: 10
 targetCPUUtilizationPercentage: 50
```

Рисунок 4.10 – Horizontal Pod Autoscaler

Наведений скрін показує, що HPA підтримує від 1 до 10 реплік модулів, що керовани hpa. 50% - це середнє використання ЦП, яке має підтримувати HPA.

Використання кластерів для деплою застосунків має ряд переваг. Перш за все, автоматичне масштабування є легко налаштовуваним, як описано вище. Крім того, сервіс оркестрації контейнерів надає можливість відстежувати роботу контейнерів, створювати журнали обліку навантаження, керувати безпекою та виконувати багато інших важливих функцій для моніторингу і керування додатком.

Хоча переваги такого розгортання можна важко спостерігати під час використання невеликих тестових застосунків, вони стають особливо корисними для великих сервісів. Кожен наступний контейнер може бути розгорнутий з меншими зусиллями, що робить цей метод оптимальним для масштабування застосунків.

Рекомендації щодо розгортання контейнерного застосунку

Одним з завдань роботи є розроблення рекомендацій щодо розгортання застосунку з використанням платформи Kubernetes. Оскільки цей процес кластеризації є складним і вимагає значних зусиль, перш за все необхідно визначити межі проекту і визначити, чи є необхідність використовувати цю

методологію для розгортання веб-сервісу. Можливо, використання послуг веб-хостингу або просто хмарного хостингу повністю задовольнить потреби проекту. Крім того, хмарні провайдери пропонують широкий спектр послуг PaaS та навіть SaaS, які можуть значно спростити розробку застосунку і прискорити процес отримання готового рішення. Після оцінки всіх доступних варіантів, можна зробити рекомендації щодо оптимального розгортання застосунку.

Враховуючи велику кількість даних у проекті, варто розгялнути спосіб використання мікросервісів, що вже містяться в контейнерах, і можуть бути впорядкованим, ефективним контейнерами системи оркестрації, як-от Kubernetes. План розгортання складається з наступних етапів:

- розгортання контейнерів за допомогою оркестрації, наприклад Kubernetes,
- створення кластера у вихідній групі ресурсів,
- розміщення контейнерів у кластері,
- налаштування конфігурації розгортання реплік подів,
- встановлення параметрів мережі та прав доступу,
- розгортання кластера за допомогою конфігурації,
- моніторинг і управління роботою кластерів в системі оркестрації.

Цей метод забезпечує високу мобільність, надійність, продуктивність і швидкість проекту, а головне повну автоматичну стабільність і ефективне використання ресурсів.

Висновки до розділу 4

В цьому розділі було показано, як створити кластер Kubernetes в хмарному середовищі AWS, порівняно два способи деплою веб-застосунка, з використанням хмарних технологій та хостингу. Розглянуто спосіб розгортання застосунка з можливістю масштабування в залежності від

навантаження по CPU. Наведено рекомендації по розгортанню контейнерного застосунку.

ВИСНОВКИ

В ході виконання магістерської роботи було досліджено можливості, які можуть бути реалізовані за допомогою хмарних технологій. В розглянуто використання Kubernetes, який є одним з провідних постачальників хмарних послуг. Описано оптимізовану схему роботи з контейнерними додатками та розглянуто процес розгортання застосунку на платформі оркестрації контейнерів.

У першому розділі досліджено головні переваги хмарних обчислень в сучасному інформаційному світі, включаючи уніфікацію мережі інтернет в хмарну систему, стандартизацію зв'язків між елементами, основні послуги, які надаються хмарними провайдерами, такі як підтримка віртуалізації та контейнеризації, надання фізичних та програмних ресурсів та підтримка безпеки даних. Також відзначено проблему швидкого масштабування застосунків через високу доступність даних.

У другому розділі наведено основні переваги та недоліки безсерверної, серверної та монолітної архітектури.

У третьому розділі проведено порівняльний аналіз існуючих рішень для швидкого та ефективного розгортання багатокомпонентних сервісів, які потенційно можуть розширюватись. Розглянуто рішення від провідних провайдерів хмарних послуг.

У четвертому розділі описано оптимізоване розгортання застосунку в системі Kubernetes та порівняно цей процес з розгортанням на звичайному веб-хостингу. Розроблено конфігурації автомасштабування застосунку та надано рекомендації щодо вибору методу розгортання застосунку.

ПЕРЕЛІК ДЖЕРЕЛ ПОСИЛАННЯ

1. Гонтаренко С. С., Крайник Я. М. Процес розгортання кластерного застосунку в Kubernetes. Інформаційні технології та інженерія : тези доп. Всеукр. наук.-практ. конф. Миколаїв, 07–10 лют. 2023 р. Миколаїв : Чорном. нац. ун-т ім. Петра Могили, 2023. С. 12–14.

2. Nikhil Buduma Fundamentals of Deep Learning. Designing nextgeneration machine intelligence algorithms. O’Reilly Media, 2017. 277 с.

3. Юрія Казакова та Михайла Головача, “Оптимізація розгортання та масштабування мікросервісних застосунків з використанням Kubernetes” 2013 (Дата звернення: 18.10.2022).

4. Андрія Григоренка, Дмитра Коваленка, “Інструменти для оркестрування масштабованих застосунків: аналіз Docker Swarm, Kubernetes та Mesos” 2014 (Дата звернення: 19.10.2022).

5. Amazon, Microsoft Or Google: Which Is The Best Play On Surging Cloud Infrastructure Demand, URL: https://www.forbes.com/sites/greatspeculations/2020/05/14/amazon- microsoft-or-google-which-is-the-best-play-on-surging-cloud-infrastructure- demand/ (Дата звернення: 18.10.2022).

6. Comparing Kubernetes Services on AWS vs. Azure vs. GCP, URL: https://www.sumologic.com/blog/kubernetes-aws-azure-gcp/ (Дата звернення: 22.10.2022).

7. AWS EKS, URL: https://www.edx.org/course/aws-eks-scalable-microservices-with-kubernetes/ (Дата звернення: 30.10.2022).

8. Deploy a containerized application on AWS Kubernetes Service, URL: https://docs.microsoft.com/en-us/learn/modules/aks-deploy-container-app/ (Дата звернення: 30.10.2022).

9. AWS production envoronment, URL: https://kubernetes.io/docs/setup/production-environment/turnkey/aws/ (Дата звернення: 1.11.2022).

10. Developers bring their ideas to life with Docker, URL: https://www.docker.com/why-docker (Дата звернення: 1.11.2022).

11. DevOps, URL: https://www.ibm.com/cloud/learn/devops-a-complete-guide (Дата звернення: 1.11.2022).

12. Kubernetes blog, URL: https://cloudacademy.com/blog/category/kubernetes/ (Дата звернення: 11.11.2022).

13. Kubernetes documentation, URL: https://kubernetes.io/docs/ (Дата звернення: 11.11.2022).

14. Kubernetes on AWS, URL: https://aws.amazon.com/ua/kubernetes/ (Дата звернення: 13.11.2022).

15. Kubernetes vs. Docker: A Primer, URL: Kubernetes vs. Docker: A Primer (Дата звернення: 13.11.2022).

16. Why (and when) you should use Kubernetes, URL: : https://hackernoon.com/why-and-when-you- should-use-kubernetes8b50915d97d8 (Дата звернення: 13.11.2022).

17. Мікросервісна архітектура, URL: : https://medium.com/@IvanZmerzlyi/microservices- architecture-461687045b3d (Дата звернення: 15.11.2022).

18. Хмарні обчислення, URL: : https://en.wikipedia.org/wiki/Cloud_computing (Дата звернення: 15.11.2022).

19. Хмарні обчислення, URL: : https://en.wikipedia.org/wiki/Cloud_computing (Дата звернення: 15.11.2022).

20. Марка Бетті, Івана Педретті, "Контейнеризація та оркестрування масштабованих застосунків з використанням Docker та Kubernetes" 2011 (Дата звернення: 23.10.2022).

Додаток А

Програмний код

Лістинг коду deploy.yml

```
apiVersion: apps/v1
kind: ReplicaSet
metadata:
  labels:
    app: node-nginx
  name: node-nginx
spec:
  replicas: 10
  selector:
    matchLabels:
      app: node-nginx
  template:
    metadata:
      labels:
        app: node-nginx
    spec:
      affinity:
        nodeAffinity:
          requiredDuringSchedulingIgnoredDuringExecution:
            nodeSelectorTerms:
            - matchExpressions:
              - key: purpose
                operator: In
                values:
                - stateless
      containers:
      - command:
        - yarn
        - start
        env:
        - name: CONFIG
          value: production
        image: 968199160822.dkr.ecr.us-east-2.amazonaws.com/k8s-
web/node:a44afbc
        imagePullPolicy: IfNotPresent
        livenessProbe:
          failureThreshold: 3
          periodSeconds: 20
          successThreshold: 1
          tcpSocket:
            port: node
          timeoutSeconds: 10
        name: node
        ports:
        - containerPort: 3000
          name: node
          protocol: TCP
```

```
        resources:
          limits:
            cpu: 400m
            memory: 2Gi
          requests:
            cpu: 200m
            memory: 1Gi
      - command:
        - nginx
        - -g
        - daemon off;
        image: 968199160822.dkr.ecr.us-east-2.amazonaws.com/k8s-
web/nginx:a44afbc
        lifecycle:
          preStop:
            exec:
              command:
              - /usr/sbin/nginx
              - -s
              - quit
        livenessProbe:
          failureThreshold: 3
          periodSeconds: 20
          successThreshold: 1
          tcpSocket:
            port: http
          timeoutSeconds: 10

        name: nginx
        ports:
        - containerPort: 80
          name: http
          protocol: TCP
        resources:
          limits:
            cpu: 100m
            memory: 64Mi
          requests:
            cpu: 50m
            memory: 32Mi
        startupProbe:
          exec:
            command:
            - sh
            - -c
            - |
              curl --fail http://localhost:80/health-check
          failureThreshold: 15
          initialDelaySeconds: 10
          periodSeconds: 20
          successThreshold: 1
          timeoutSeconds: 10
        volumeMounts:
        - mountPath: /var/log
          name: logs
```

```
        - mountPath: /etc/nginx/nginx.conf
          subPath: nginx.conf
```

Лістинг коду service.yml

```
apiVersion: v1
kind: Service
metadata:
  name: frontend
  labels:
    app: node-nginx
spec:
  ports:
  - port: 80
    targetPort: 80
  type: NodePort
  selector:
    app: node-nginx
```

Лістинг коду ingrres.yml

```
ingress:
  commonLabels:
    app: ingress
    project: frontend
  annotations:
    kubernetes.io/ingress.class: alb
    alb.ingress.kubernetes.io/scheme: internet-facing
    alb.ingress.kubernetes.io/certificate-arn: arn:aws:acm:us-east-
2:968199160822:certificate/79b1a83d-33ce-4d45-bd18-77ddc6069f8d
    # Use this annotation (which must match a service name) to route traffic
to HTTP2 backends.
    alb.ingress.kubernetes.io/listen-ports: '[{"HTTP":80}, {"HTTPS":443}]'
    alb.ingress.kubernetes.io/ip-address-type: ipv4
    alb.ingress.kubernetes.io/group.name: 'frontend'
    alb.ingress.kubernetes.io/load-balancer-name: 'eks-frontend'
    alb.ingress.kubernetes.io/actions.ssl-redirect: '{"Type": "redirect",
"RedirectConfig": { "Protocol": "HTTPS", "Host": "my-apps.xyz", "Port":
"443", "StatusCode": "HTTP_301"}}'
    alb.ingress.kubernetes.io/healthcheck-protocol: HTTP
    alb.ingress.kubernetes.io/healthcheck-path: /health-check
    alb.ingress.kubernetes.io/healthcheck-interval-seconds: '20'
    alb.ingress.kubernetes.io/healthcheck-timeout-seconds: '8'
    alb.ingress.kubernetes.io/success-codes: 200,301,302,308
  tlsEnabled: true
  hosts:
  - my-apps.xyz
  rules:
  - host: my-apps.xyz
    http:
      paths:
      - path: /
        pathType: Prefix
        backend:
```

```
      service:
        name: ssl-redirect
        port:
          name: use-annotation
```

Лістинг коду hpa.yml

```
apiVersion: autoscaling/v1
kind: HorizontalPodAutoscaler
metadata:
 name: hpa
spec:
 scaleTargetRef:
   apiVersion: apps/v1
   kind: ReplicaSet
   name: node
 minReplicas: 1
 maxReplicas: 10
 targetCPUUtilizationPercentage: 50
```

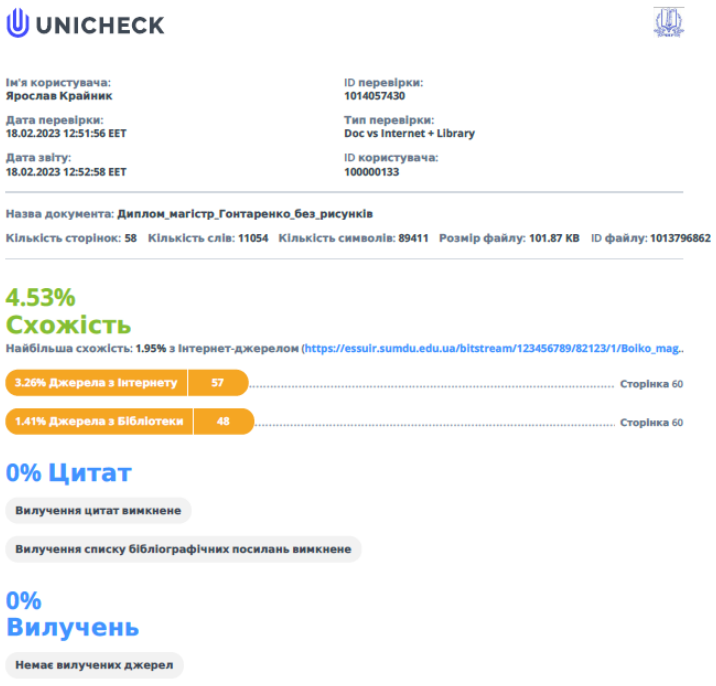
Додаток А

Перевірка на унікальність

ДОВІДКА

про перевірку на унікальність пояснювальної записки
кваліфікаційної магістерської роботи
на тему: «Аналіз технологій контейнеризації та оптимізація розгортання масштабованого застосунка на платформі Kubernetes»
студента спеціальності 123 «Комп’ютерна інженерія», групи 605
Гонтаренка Сергія Сергійовича
прізвище, ім’я, по-батькові

Перевірку тексту здійснено сервісом: онлайн-сервіс Unicheck.
Результат перевірки тексту магістерської роботи: схожість складає 4,53 %.


UNICHECK

Ім'я користувача: Ярослав Крайник
ID перевірки: 1014057430
Дата перевірки: 18.02.2023 12:51:56 EET
Тип перевірки: Doc vs Internet + Library
Дата звіту: 18.02.2023 12:52:58 EET
ID користувача: 100000133

Назва документа: Диплом_магістр_Гонтаренко_без_рисунків
Кількість сторінок: 58 Кількість слів: 11054 Кількість символів: 89411 Розмір файлу: 101.87 KB ID файлу: 1013796862

4.53%
Схожість
Найбільша схожість: 1.95% з Інтернет-джерелом (https://essuir.sumdu.edu.ua/bitstream/123456789/82123/1/Boiko_mag..

3.26% Джерела з Інтернету 57 Сторінка 60
1.41% Джерела з Бібліотеки 48 Сторінка 60

0% Цитат
Вилучення цитат вимкнене
Вилучення списку бібліографічних посилань вимкнене

0%
Вилучень
Немає вилучених джерел