ДЕРЖАВНИЙ УНІВЕРСИТЕТ
ІНФОРМАЦІЙНО-КОМУНІКАЦІЙНИХ ТЕХНОЛОГІЙ

НАВЧАЛЬНО–НАУКОВИЙ ІНСТИТУТ ІНФОРМАЦІЙНИХ ТЕХНОЛОГІЙ

КАФЕДРА КОМП'ЮТЕРНИХ НАУК

Кваліфікаційна робота

на тему: «**СИСТЕМА ВІРТУАЛІЗАЦІЇ І КОНТЕЙНЕРИЗАЦІЇ ДЛЯ ХМАРНИХ СЕРВІСІВ»**

на здобуття освітнього ступеня магістра
зі спеціальності __122 Комп'ютерні науки_____
(код, найменування спеціальності)
освітньо-професійної програми __Комп'ютерні науки_____
(назва)

Кваліфікаційна робота містить результати власних досліджень. Використання ідей, результатів і текстів інших авторів мають посилання на відповідне джерело

_____ ___Володимир СЕМЕНЕНКО _____
(підпис) *Ім'я, ПРІЗВИЩЕ здобувача*

Виконав: здобувач вищої освіти гр. ___КНДМ-61___
_____Володимир СЕМЕНЕНКО_____
(Ім'я, ПРІЗВИЩЕ)

Керівник: __д.ф., доцент Юлія БЕРЕЗОВСЬКА__
Науковий ступінь, вчене звання *(Ім'я, ПРІЗВИЩЕ)*

Рецензент: _____
Науковий ступінь, вчене звання *(Ім'я, ПРІЗВИЩЕ)*

Київ – 2024

ДЕРЖАВНИЙ УНІВЕРСИТЕТ
ІНФОРМАЦІЙНО-КОМУНІКАЦІЙНИХ ТЕХНОЛОГІЙ

Навчально-науковий інститут інформаційних технологій

Кафедра Комп'ютерних наук
Ступінь вищої освіти − «Магістр»
Спеціальність підготовки − «122 Комп'ютерні науки»
Освітньо-професійна програма − «Комп'ютерні науки»

ЗАТВЕРДЖУЮ
Завідувач кафедри
Комп'ютерних наук
_____Віктор Вишнівський
"____" _____2024 року

ЗАВДАННЯ
НА КВАЛІФІКАЦІЙНУ РОБОТУ

_____Семененко Володимир Олександрович _____
(прізвище, ім'я, по батькові здобувача)

1.Тема кваліфікаційної роботи: «Система віртуалізації і контейнеризації для хмарних сервісів»
керівник кваліфікаційної роботи __Юлія БЕРЕЗОВСЬКА, д.ф., доцент._____,
(ім'я, ПРІЗВИЩЕ, науковий ступінь, вчене звання)
затверджені наказом Державного університету інформаційно-комунікаційних технологій від «15» жовтня 2024р. №320

2. Строк подання кваліфікаційної роботи «15» грудня 2024р.

3. Вихідні дані до кваліфікаційної роботи: алгоритми побудови віртуальних машин, технологія контейнеризації, Docker, методи оркестрації контейнерів, Docker Swarm, Kubernetes, архітектура хмарних обчислень, алгоритми створення хмарної федерації, технологія Edge Cloud, методи оркестрації контейнерів у граничній хмарі, науково-технічна література по темі роботи

4. Зміст розрахунково-пояснювальної записки (перелік питань, які потрібно розробити)
4.1 Аналіз сучасних методів віртуалізації і контейнеризації
4.2 Аналіз та дослідження оркестрації контейнерів в системах хмарної федерації
4.3 Дослідити способи запровадження ідентифікатора ресурсу та оркестратора

контейнерів у еталонну архітектуру хмарної федерації
4.4 Проаналізувати та дослідити способи організації контейнерів у граничній хмарі
4.5 Аналіз методів управління контейнерними кластерами у граничній хмарі
4.6 Провести моделювання кластерної інфраструктури граничної хмари за допомогою компонентів проекту Edge Run

5. Перелік ілюстративного матеріалу: Демонстраційний матеріал у вигляді ppt-презентації

6. Дата видачі завдання «16» вересня 2024 р.

КАЛЕНДАРНИЙ ПЛАН

№ з/п	Назва етапів кваліфікаційної роботи	Строк виконання етапів роботи	Примітка
1	Підбір науково-технічної літератури	10.09.2024р	вик.
2	Сучасні рішення щодо застосування сучасних методів віртуалізації і контейнеризації	18.09.2024р	вик.
3	Порівняльний аналіз контейнерних технологій	28.09.2024р	вик.
4	Архітектура для ідентифікації ресурсів та управління оркеструванням	01.10.2024р	вик.
5	Оркестрування контейнерів у хмарних федераціях	11.10.2024р	вик.
6	Блок-схеми алгоритму оркестрації контейнерів	21.10.2024р	вик.
7	Лінійна регресія компонента ідентифікатора ресурсу	29.10.2024р	вик.
8	Проект Edge Run	05.11.2024р	вик.
9	Моделювання	08.11.2024р	вик.
10	Розділ 1	10.11.2024р	вик.
11	Розділ 2	16.11.2024р	вик.
12	Розділ 3	22.11.2024р	вик.
13	Оформлення роботи	29.11.2024р	вик.
14	Презентація	01.12.2024р	вик.
15	Підготовка до захисту	15.12.2024р	вик.

Здобувач вищої освіти _____ __Володимир СЕМЕНЕНКО_____
(підпис) (Ім'я, ПРІЗВИЩЕ)

Керівник кваліфікаційної роботи _____ ___Юлія БЕРЕЗОВСЬКА___
(підпис) (Ім'я, ПРІЗВИЩЕ)

РЕФЕРАТ

Текстова частина кваліфікаційної роботи на здобуття освітнього ступеня магістра: 76 стор., 24 рис., 2 табл., 32 джерела.

Мета роботи – дослідити можливість та дати рекомендації щодо покращення ефективності процесів оркестрації контейнерів у системах хмарних сервісів, від хмарної федерації до граничних обчислень.

Об`єкт дослідження – процеси та технології організації контейнерів у системах хмарних обчислень.

Предмет дослідження – методи та способи створення сучасних систем оркестрації контейнерів у хмарних сервісах.

Короткий зміст роботи:

У роботі проведено аналіз шляхів підвищення ефективності побудови сучасних алгоритмів оркестрації контейнерів у системах хмарних сервісів. Наведено огляд різних напрямків досліджень за контейнерною технологією, які актуальні для хмарних обчислень. Проведено порівняльний аналіз контейнерних технологій, показано переваги Docker як основної платформи для розробки, доставки та експлуатації додатків. Проведено порівняння інструментів для оркестрації контейнерів. Проаналізовано та досліджено архітектуру для ідентифікації ресурсів та управління оркеструванням контейнерів у хмарних федераціях. Запропоновано два нових підкомпоненти в еталонну архітектуру NIST. Запропоновано використання техніки лінійної регресії компонента ідентифікатора ресурсу. На основі аналізу блок-схеми алгоритму оркестрації контейнерів виявлено основні особливості контейнерної організації у хмарній федерації. Розглянуто основні риси процесів кластеризації та оркестрації контейнерів у граничній хмарі. У зв'язку з цим було проведено моделювання у кластерній інфраструктурі граничної хмари з використанням платформи проекту Edge Run. На основі цього запропоновано шляхи покращення ефективності механізмів організації контейнерів у хмарній федерації та граничній хмарі.

КЛЮЧОВІ СЛОВА: Контейнер, Docker, Оркестрація, Kubernetes, Хмарна федерація, Гранична хмара.

ABSTRACT

Text part of the master's qualification work: 76 pages, 24 pictures, 2 tables, 32 sources.

The purpose of the work – to explore the possibility and provide recommendations for improving the efficiency of container orchestration processes in cloud service systems, from cloud federation to edge computing.

Object of research – processes and technologies for organizing containers in cloud computing systems.

Subject of research – methods and techniques for creating modern container orchestration systems in cloud services.

Summary of the work:

The thesis analyzes ways to improve the efficiency of building modern container orchestration algorithms in cloud service systems. An overview of various research areas in container technology relevant to cloud computing is provided. A comparative analysis of container technologies is conducted, showing their advantages as the main platform for developing, delivering, and operating applications. A comparison of container orchestration tools is conducted. The architecture for resource identification and container orchestration management in cloud federations is analyzed and investigated. Two new subcomponents are proposed in the NIST reference architecture. The use of the linear regression technique of the resource identifier component is proposed. Based on the analysis of the container orchestration algorithm flowchart, the main features of container organization in cloud federation are identified. The main features of the clustering and container orchestration processes in the edge cloud are considered. In this regard, modeling was carried out in the cluster infrastructure of the edge cloud using the Edge Run project platform. Based on this, ways to improve the efficiency of container organization mechanisms in cloud federation and edge cloud are proposed.

KEYWORDS: Container, Docker, Orchestration, Kubernetes, Cloud Federation, Edge Cloud.

ЗМІСТ

ПЕРЕЛІК УМОВНИХ ПОЗНАЧЕНЬ

VM (Virtual Machine) – віртуальна машина;

IoT (Internet of things) – Інтернет речей;

IaaS (Infrastructure as a Service) – інфраструктура як послуга;

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

SaaS (Software as a Service) – програмне забезпечення як послуга;

OS – операційна система;

CSP (cloud service providers) – постачальники хмарних послуг;

FO (Federation Broker) – брокер федерації;

RM (Resource Manager) – менеджер ресурсів;

RI (Resource Identifier) – ідентифікатор ресурсів;

SDN (software-defined networks) – програмно-визначені мережі;

API (application programming interface) – прикладний програмний інтерфейс;

TOSCA (Topology and Orchestration Specification for Cloud Applications) – Специфікація топології та оркестровки для хмарних програм;

RTT (round-trip-time) – час створення трасування;

CDN (Content Delivery Network) – мережа доставки вмісту;

AR (Augmented Reality) – доповнена реальність.

ВСТУП

Інновації необхідні, щоб долати неминучу хвилю змін, і з цієї причини більшість підприємств прагнуть зменшити витрати на обчислення. Найважливішою відповіддю на вимогу зменшення витрат на обчислення є впровадження хмарних обчислень [1]. Це швидко стає стандартом для розміщення та запуску програмного забезпечення в Інтернеті. Хмарні обчислення — це не нова технологія, а IT-парадигма, яка пропонує скорочення попередньої капіталізації, швидку масштабованість і повсюдну доступність. Хмарні обчислення були розроблені в результаті розвитку інших існуючих рішень, таких як віртуалізація. Це одна з фундаментальних технологій, яка склала хмарну парадигму. Завдяки цій новій моделі сьогодні споживачі можуть більше зосередитися на основних функціях програми, а не на бізнес-аспектах. Такі підходи формувалися на протязі достатньо тривалого проміжку часу. Історія їх починається з початкового моменту виникнення комп'ютерів. У цей час комп'ютери були дуже дорогими, треба було забезпечити роботу багатьох дослідницьких груп. Таким чином виникла необхідність віртуального розбиття реального комп'ютера на кілька робочих машин – віртуальних машин (VM, Virtual Machine). У подальшому технологія подібного вдосконалення успішно розвивалася, виникла концепція контейнерів, яка є менш простою і в той же час більш вдосконаленою з точки зору організації процесів.

Технологія контейнеризації допомагає досягти не тільки кращої мобільності та взаємодії, але й кращої продуктивності та ефективності у різних хмарних обчислювальних схемах. Очікується, що така технологія розширить можливості хмарних федерацій за рахунок покращення мобільності та масштабованості в рамках федерації.

Хмарні технології рухаються у бік більшого поширення у багатохмарних середовищах та включення різних пристроїв, як це видно з інтеграції Інтернету речей (IoT, Internet of Things) та мереж у контексті периферійних (граничних,

хмарних) та туманних обчислень. Як правило, легкі рішення віртуалізації вигідні для цього архітектурного середовища з меншими, але все ще віртуалізованими пристроями для розміщення додатків та платформних сервісів, а також логістикою, яка потрібна для їх управління. Контейнеризація при цьому виступає як легке рішення віртуалізації. Крім переваг у порівнянні з традиційними віртуальними машинами у хмарі з точки зору розміру та гнучкості, контейнери особливо актуальні для проблем платформи, які зазвичай вирішуються у хмарах PaaS (Platform as a Service), таких як упаковка та оркестрування додатків. Для периферійного хмарного середовища оркестрування додатків та сервісів може допомогти керувати та оркеструвати додатки за допомогою контейнерів як механізму пакування додатків.

Таким чином, актуальними стають кілька напрямів досліджень, пов'язаних із технологіями організації контейнерів від хмарної федерації до граничних обчислень. Ці питання вирішуються додаванням нових компонентів у хмарну федерацію з урахуванням моделювання розподілу навантаження методами машинного навчання. Для граничної хмари на першому плані виступає моделювання на основі параметрів конкретних пристроїв, які працюють на периферії. Остання область привертає значну увагу з точки зору організації обчислень у розподілених системах.

Метою роботи є розроблення нових підходів щодо покращення ефективності процесів оркестрації контейнерів у системах хмарних сервісів, від хмарної федерації до граничних обчислень. Для досягнення цієї мети треба вирішити наступні завдання дослідження. Провести порівняльний аналіз контейнерних технологій, інструментів для оркестрації контейнерів. Дослідити архітектуру для ідентифікації ресурсів та управління оркеструванням контейнерів у хмарних федераціях. Провести моделювання та обробку даних по розподілу навантаження в структурах хмарної федерації методами машинного навчання. Розглянути особливості процесів кластеризації та оркестрації контейнерів у граничній хмарі. Загалом, об`єктом дослідження є процеси та технології

організації контейнерів у системах хмарних обчислень. А предметом дослідження є методи та способи створення сучасних систем оркестрації контейнерів у хмарних сервісах

Робота складається зі вступу, трьох розділів, висновків, списку використаних джерел. На основі поставленої мети завдання у роботі проведено аналіз шляхів підвищення ефективності побудови сучасних алгоритмів оркестрації контейнерів у системах хмарних сервісів. У першому розділі наведено огляд різних напрямків досліджень за контейнерною технологією, які актуальні для хмарних обчислень. Далі на основі літературних джерел проведено порівняльний аналіз контейнерних технологій, показано переваги Docker як основної платформи для розробки, доставки та експлуатації додатків. Решта першого розділу присвячена порівнянню інструментів для оркестрації контейнерів. У другому розділі представлено аналіз та дослідження архітектури для ідентифікації ресурсів та управління оркеструванням контейнерів у хмарних федераціях. Далі обговорюється перевага техніки лінійної регресії для компонента ідентифікатора ресурсу. На основі аналізу блок-схеми алгоритму оркестрації контейнерів виявлено основні особливості контейнерної організації у хмарній федерації. З урахуванням нових компонентів у хмарної федерації проведено моделювання розподілу навантаження методами машинного навчання. Третій розділ присвячений дослідженню процесів у граничній хмарі. Для цього розглянуто основні риси процесів кластеризації та оркестрації контейнерів у граничній хмарі. У зв'язку з цим проведено моделювання у кластерній інфраструктурі граничної хмари з використанням платформи проекту Edge Run. На основі цього запропоновано шляхи покращення ефективності механізмів організації контейнерів у хмарній федерації та граничній хмарі.

Кваліфікаційна робота апробована на Міжнародній науково-технічній конференції «Сучасні досягнення компанії Hewlett Packard Enterprise в галузі IT та нові можливості їх вивчення й застосування».

1 АНАЛІЗ ПРЕДМЕТНОЇ ГАЛУЗІ

У цьому розділі ми розглянемо дослідження віртуалізації, контейнеризації, порівняння продуктивності віртуалізації та контейнеризації, хмарних обчислень і переваги використання хмарних служб.

1.1 Віртуалізація

Віртуалізація — це технологія, яка створює абстрактний рівень над апаратним забезпеченням комп'ютера, який дозволяє використовувати апаратне забезпечення одного комп'ютера та розділяти його на кілька віртуальних комп'ютерів, відомих як віртуальні машини. Кожна віртуальна машина працює під керуванням власної операційної системи. Віртуальні машини можна використовувати, якщо вам потрібні різні або кілька однакових операційних систем на одному фізичному комп'ютері.

Віртуалізація — це також технологія, яка швидко розвивається разом із цифровізацією. Цифровізація є фундаментальною рушійною силою, спричиненою Четвертою промисловою революцією та Інтернетом речей (IoT, Internet of things), які змінили наш підхід до бізнес-процесів і діяльності та їхнє мислення [2]. Такі технології, як IoT, різні типи хмарних сервісів, як-от інфраструктура як послуга (IaaS, Infrastructure as a Service), платформа як послуга (PaaS, Platform as a Service) і програмне забезпечення як послуга (SaaS, Software as a Service), допомогли компаніям швидше розробляти та розгортати свої продукти.

Віртуалізація та запровадження центрів обробки даних позитивно вплинули на компанії з фінансової точки зору, обмеживши вимоги щодо придбання обладнання для розміщення їхніх продуктів та підтримки інфраструктури розміщення своїх серверів усередині компанії. Так, у багатьох роботах було виявлено, що невеликі компанії, які мають визначення менше 100 серверів, могли б заощадити більше 200% поточного бюджету на IT-інфраструктуру компаній,

впровадивши хмарні обчислення замість розміщення власної серверної інфраструктури.

1.1.1 Технологія створення віртуального комп'ютера

Оскільки процеси, пов'язані з віртуалізацією, спрямовано на економію ресурсів організацій, то природно, що ця парадигма виникла вже давно. Навіть із назви зрозуміло, що процеси віртуалізації пов'язані з емуляцією та імітацією роботи самої комп'ютерної системи. Іншими словами, необхідно створити таку програму, яка б імітувала роботу самого "заліза".

Якщо порівнюємо підходи до процесів організації VM, слід відзначити, що технології віртуалізації повинні забезпечувати спосіб спільного використання обчислювальних ресурсів між віртуальними машинами за допомогою розділення апаратного/програмного забезпечення, емуляції, розподілу часу та динамічного спільного використання ресурсів [3]. Традиційно операційна система (OS) контролює апаратні ресурси, але технологія віртуалізації додає новий рівень між OS і апаратним забезпеченням. Розподілом ресурсів займається спеціальна програма, так званий гіпервізор (hypervisor) [4]. До кожного цільового призначення процес організації гіпервізорів реалізуються різними способами. Їх прийнято поділити на кілька, як правило, на дві основні технології. Таким чином, відмінності між гіпервізорами, як показано на рисунку 1.1, пов'язані з тим, що вони кардинально різняться за своєю архітектурою.

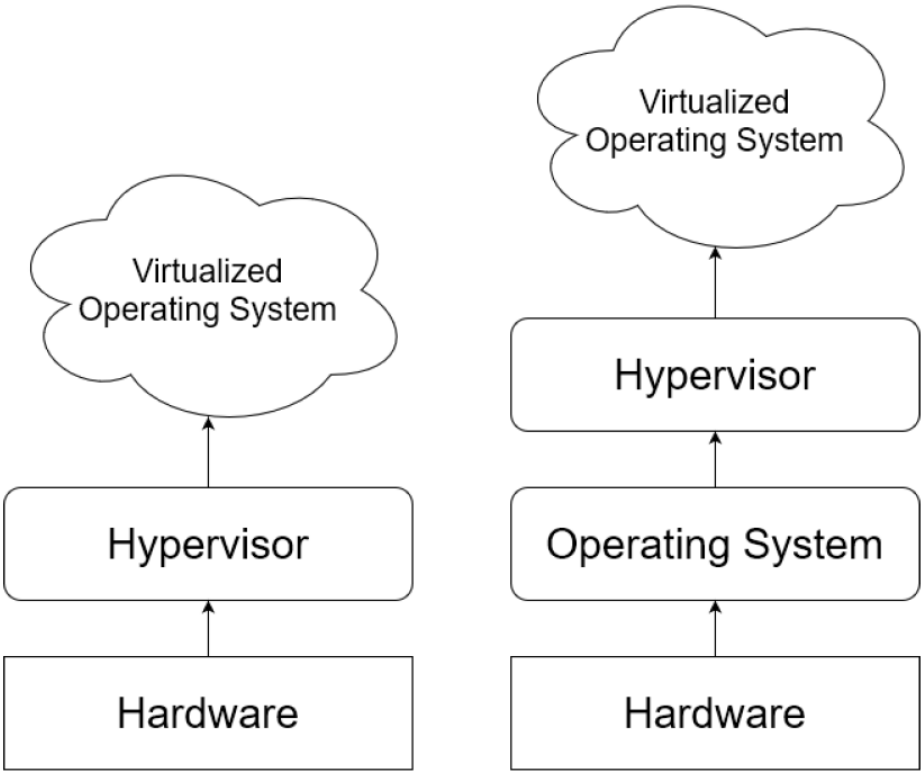


Рисунок 1.1 – Архітектура гіпервізора типу 1 (зліва) і типу 2

Гіпервізор типу 1 розміщує віртуальні машини поверх обладнання і таким чином ефективніший у своєму способі використання доступного обладнання. Гіпервізор типу 2 розміщує віртуальні машини поверх операційної системи хоста, що збільшує накладні витрати до досягнення обладнання.

Поряд з цим існують інші способи організації VM. Так, корпорація Microsoft розробила Hyper-V – гіпервізор типу 1, який входить до складу деяких версій x64 операційних систем Windows [5]. Hyper-V – це мікроядерний гіпервізор, архітектура якого не потребує встановлення будь-яких драйверів для фізичного обладнання в гіпервізорі. Натомість вони використовують батьківський розділ, що містить усі драйвери для фізичного обладнання. У роботі [6] розглядається питання про перевагу мікроядерних гіпервізорів, яке полягає в тому, що драйвери пристроїв не повинні бути поінформовані про структуру гіпервізора, що дає мікроядерним гіпервізорам можливості підтримувати ширший спектр пристроїв. Основний недолік мікроядерних гіпервізорів полягає в тому, що

їм потрібна операційна система у батьківському розділі, перш ніж гіпервізор зможе працювати. Порівняння між різними гіпервізорами є широко дослідженою областю, і багато з них відрізняються пропорцією віртуальних машин, які вони тестують. Щоб підтримати Hyper-V як дійсний вибір для представлення віртуалізації, достатньо розглянути попередні дослідження [7], що порівнюють продуктивність гіпервізорів типу 1.

1.1.2 Види віртуалізації

Технології віртуалізації можна розділити на три категорії: повна віртуалізація, апаратна віртуалізація та паравіртуалізація. Повна віртуалізація – це метод віртуалізації, який дозволяє повністю моделювати базове обладнання VM. Він дозволяє виконувати будь-яке програмне забезпечення, яке може працювати на фізичному обладнанні в VM, та підтримує використання кількох гостьових операційних систем одночасно. Повна віртуалізація складається з підходу, який спирається на двійкову трансляцію для перехоплення в моніторі VM (гіпервізора) та віртуалізації певних конфіденційних і невіртуалізованих інструкцій за допомогою нових послідовностей інструкцій, які мають запланований вплив на віртуальне обладнання [8]. Надаючи віртуальний BIOS, віртуальні пристрої та віртуалізоване керування пам'яттю, повна віртуалізація ізолює гостьову операційну систему від базового обладнання, забезпечуючи підвищену безпеку та простоту міграції. Тим часом код на рівні користувача виконується безпосередньо на процесорі для забезпечення високої продуктивності. При цьому команди, що надходять від гостьової операційної системи, перехоплюються гіпервізором. Таким чином, він є унікальним компонентом, здатним виконувати ці операції.

Одним із основних елементів апаратної віртуалізації першого покоління було впровадження кільцевої архітектури процесора x86, відомої як Ring-1. Це дозволяє гіпервізорам працювати на кільці -1, щоб виконувати гостьові інструкції на кільці 0, так само, як вони зазвичай виконуються на фізичному хості [9]. Він

забезпечує рівень Ring-1 і гіпервізор працює на Ring-1, в той час як операційна система працює на Ring 0. Це не вимагає процесу двійкової трансляції для привілейованих команд, і команда виконується безпосередньо обладнанням через гіпервізор з помітним поліпшенням продуктивності.

Паравіртуалізація — це вдосконалення технології віртуалізації, за якої гостьова операційна система перекомпілюється перед встановленням у віртуальну машину. Він створює рівень інтерфейсу для гостьових операційних систем, який може дещо відрізнятися від базового апаратного забезпечення. Ця потужність мінімізує накладні витрати та оптимізує продуктивність системи завдяки підтримці використання віртуальних машин.

На рисунку 1.2 показано рішення з паравіртуалізацією. Основним обмеженням паравіртуалізації є той факт, що гостьова операційна система має бути налаштована спеціально для роботи поверх монітора віртуальної машини. Однак паравіртуалізація позбавляє віртуальної машини необхідності перехоплювати привілейовані інструкції.

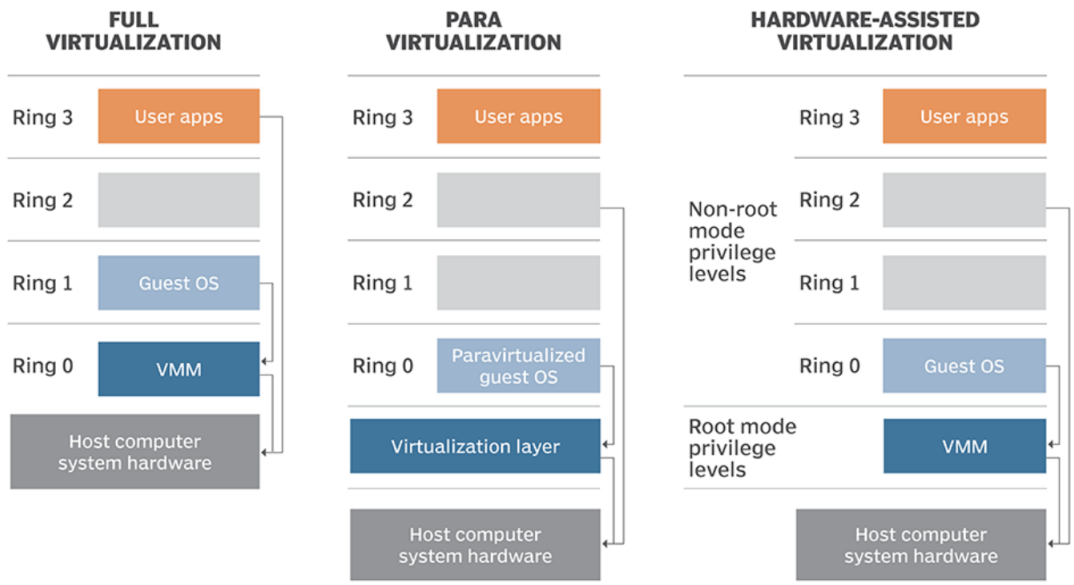


Рисунок 1.2 – Типи віртуалізації (рисунок з [10])

Кожна технологія віртуалізації має свої переваги та недоліки. Вибір

віртуалізації значною мірою залежить від використання та вартості. Розвивається багато технологій, і корпоративні продукти підтримують кілька типів віртуалізації, щоб покращити продуктивність і зменшити витрати ресурсів.

1.2 Контейнеризація

Контейнери мають довгу історію в обчислювальній техніці. На відміну від віртуалізації гіпервізора, де одна або кілька незалежних машин працюють віртуально на фізичному обладнанні через проміжний рівень, контейнери працюють у просторі користувача поверх ядра операційної системи. Тому віртуалізацію контейнера часто називають віртуалізацією на рівні операційної системи. Технологія контейнерів дозволяє запускати кілька ізольованих екземплярів простору користувача на одному хості. В результаті свого статусу як гостей операційної системи контейнери іноді вважаються менш гнучкими: вони, як правило, можуть запускати лише ту саму або подібну гостьову операційну систему, що й основний хост.

1.2.1 Технологія контейнеризації

Контейнеризація — це технологія, яка використовує контрольні групи та простори імен, головним чином, для створення ізольованих легких середовищ для запуску певних програм і коду. Ці контейнери створені спеціально для відповідних програм із лише необхідними бібліотеками та залежностями. Кілька контейнерів можуть спільно використовувати одне ядро, але кожен контейнер може бути обмежений використанням лише певної кількості ресурсів, доступних на хост-машині. Ядро в цьому випадку відноситься до ядра основної операційної системи. Архітектура контейнеризації, яку зображено на рисунку 1.3, може виглядати схожою на гіпервізори типу 2.

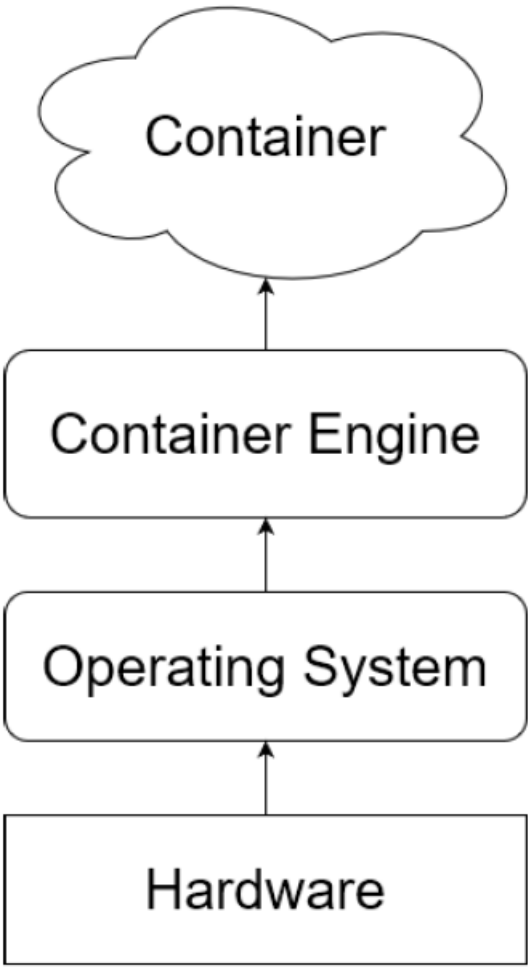


Рисунок 1.3 – Архітектура контейнеризації

Істотна різниця між ними полягає в тому, що механізм контейнеризації може мати багато контейнерів, розміщених через ядро OS, тоді як гіпервізор типу 2 повинен створити повну гостьову OS для кожної віртуальної машини. Контейнери вважаються менш безпечними, ніж повна ізоляція віртуалізації гіпервізора. Як заперечення проти цього аргументу можна навести той факт, що легкі контейнери безпечніші у разі атаки на повну операційну систему, як у випадку з віртуальною машиною, яка потенційно вразлива на рівні гіпервізора. Це випливає з того факту, що контейнери мають меншу поверхню контакту з хост-OS. Незважаючи на ці обмеження, контейнери були розгорнуті в різних випадках використання. Вони популярні для гіпермасштабованого розгортання багатокористувальницьких служб, для полегшення маніпуляцій в пісочниці та,

незважаючи на занепокоєння щодо їх безпеки, як середовища ізоляції процесів. Дійсно, одним із найпоширеніших прикладів контейнера є *chroot* (скорочення від *change root*) *jail*, який створює ізольоване середовище каталогу для запущених процесів. Зловмисники, якщо вони порушують запущений процес у в'язниці (jail), опиняються в пастці в цьому середовищі та не можуть далі скомпрометувати хост.

Останні технології контейнерів включають OpenVZ [11], Solaris Zones [12] і контейнери Linux, такі як LXC [13]. Використовуючи новітні технології, контейнери тепер можуть виглядати як повноцінні хости самі по собі, а не просто середовища виконання. У випадку Docker наявність сучасних функцій ядра Linux, таких як групи керування та простори імен, означає, що контейнери можуть мати сильну ізоляцію, власну мережу та стеки для зберігання, а також можливості керування ресурсами, щоб забезпечити дружнє співіснування кількох контейнерів на хості.

Контейнери зазвичай вважаються економічною технологією, оскільки вони потребують обмежених накладних витрат. На відміну від традиційних технологій віртуалізації або паравіртуалізації, для їх роботи не потрібен рівень емуляції або рівень гіпервізора, а замість цього використовується звичайний інтерфейс системних викликів операційної системи. Це зменшує накладні витрати необхідні для запуску контейнерів і можуть дозволити більшу щільність контейнерів для запуску на хості.

Незважаючи на свою історію, контейнери не досягли широкомасштабного впровадження. Більша частина цього може бути викладена на їх складність: контейнери можуть бути складними самі по собі, трудними в налаштуваннях і складними в управлінні та автоматизації. Docker намагається це змінити [14].

1.2.2 Docker та його переваги

Docker — це механізм із відкритим кодом, який автоматизує розгортання

програм у контейнерах. Він був написаний командою Docker, Inc (раніше dotCloud Inc, ранній гравець на ринку платформи як послуги (PAAS)) і випущений нею під ліцензією Apache 2.0.

Docker додає механізм розгортання додатків до віртуалізованого середовища виконання контейнера. Він розроблений, щоб забезпечити легке та швидке середовище для запуску вашого коду, а також ефективний робочий процес для передачі цього коду з вашого ноутбука у ваше тестове середовище, а потім у виробництво. Docker неймовірно простий. Дійсно, ви можете розпочати роботу з Docker на мінімальному хості, на якому запущено лише сумісне ядро Linux і двійковий файл Docker.

Місія Docker [14] – надати простий та легкий спосіб імплементації процесів контейнеризації. Docker це швидко. Ви можете запакувати вашу програму в контейнер за лічені хвилини. Docker використовує модель копіювання під час запису, тому внесення змін до вашої програми також неймовірно швидко, оскільки змінюється тільки те, що ви хочете змінити. Більшість контейнерів Docker запускаються менше ніж за секунду. Усунення вкладених витрат на створення гіпервізора також означає, що контейнери високопродуктивні, і ви можете упаковувати більше з них у свої хости та максимально ефективно використовувати свої ресурси.

Завдяки Docker розробники піклуються про те, щоб їхні програми запускалися всередині контейнерів, а оператори піклуються про керування контейнерами. Docker розроблено для підвищення узгодженості, гарантуючи, що середовище, у якому ваші розробники пишуть код, відповідає середовищам, у яких розгортаються ваші програми.

Docker має на меті скоротити час циклу між написанням коду та його тестуванням, розгортанням і використанням. Він спрямований на те, щоб зробити ваші програми портативними, легкими для створення та спільної роботи.

Docker також націлений на сервісно-орієнтовані та мікросервісні архітектури. Docker рекомендує [14], щоб кожен контейнер запускав одну

програму або процес. Це пов'язано з просуванням моделей розподілених додатків, де додаток або служба представлені серією взаємопов'язаних контейнерів, що спрощує поширення, масштабування, налагодження та інтроспекцію ваших додатків. Якщо програма має інші цілі, то немає необхідності створювати свої програми тільки таким чином.

1.2.3 Компоненти Docker

Розглянемо основні компоненти, з яких складається Docker. Ці компоненти є подібними до більшості механізмів контейнеризації, тому мають показовий характер. Більш повну інформацію з повною документацією можна знайти на сайті компанії [15].

Отже, по-перше, це Docker клієнт і сервер. Docker – це клієнт-серверний додаток, архітектура якого показано на рисунку 1.4.

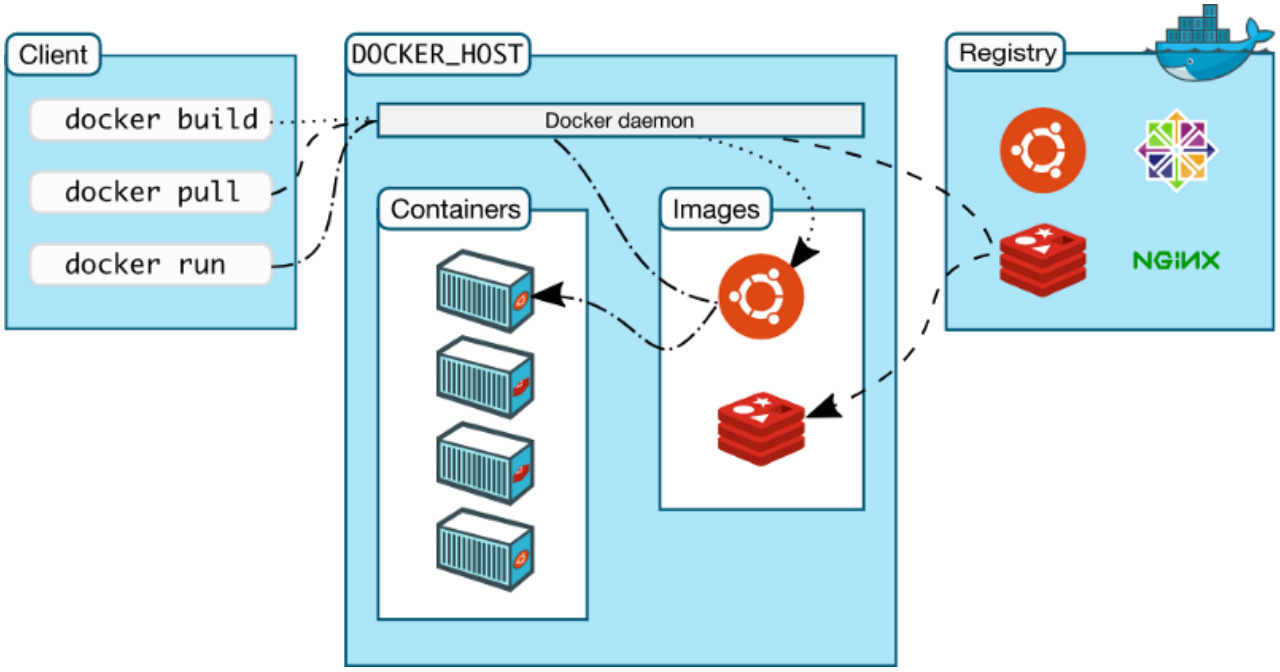


Рисунок 1.4 – Архітектура Docker

Клієнт Docker (Docker Client) – це основний засіб, який використовують для взаємодії з Docker. Так, при роботі з інтерфейсом командного рядка Docker

(Docker Command Line Interface, CLI), в термінал вводять команди, що починаються з ключового слова docker, звертаючись до клієнта. Потім клієнт використовує API Docker для надсилання команд демону Docker. Клієнт Docker взаємодіє з сервером Docker або демоном, який, у свою чергу, виконує всю роботу. Демон Docker (Docker Daemon) – це сервер Docker, який чекає на запити до API Docker.

Демон Docker управляє образами, контейнерами, мережами та томами. Ви можете запустити демон і клієнт Docker на одному хості або підключити локальний клієнт Docker до віддаленого демона, що працює на іншому хості.

Образи (Images) – це будівельні блоки світу Docker. Ви запускаєте свої контейнери із образів. Образи – це частина "складання" життєвого циклу Docker. Вони є багаторівневим форматом, що використовує системи Union file, які будуються крок за кроком за допомогою серії інструкцій. Ви можете розглядати образи як вихідний код для ваших контейнерів. Вони дуже портативні і можуть зберігатися та оновлюватися.

Docker зберігає створені вами образи у реєстрах (registry). Реєстр Docker (Docker Registry) є віддаленою платформою, що використовується для зберігання образів Docker. Під час роботи з Docker образи відправляють до реєстру та завантажують із нього. Такий реєстр можна організувати тими, хто користується Docker. Крім того, постачальники хмарних послуг можуть підтримувати власні реєстри. Наприклад, це стосується AWS та Google Cloud. Існує два типи реєстрів: публічний та приватний. Docker, Inc. керує публічним реєстром образів, що називається Docker Hub. Ви можете створити обліковий запис у Docker Hub та використовувати його для обміну та зберігання власних образів.

Docker Hub також містить, за останніми підрахунками, понад 10 000 образів, які створили та поділилися інші люди. Якщо необхідним є образ Docker для веб-сервера Nginx, системи Asterisk з відкритим вихідним кодом PABX або бази даних MySQL? Все це легко знайти, як і багато іншого.

І основна мета Docker полягає в тому, що він допомагає створювати та

розгортати контейнери, усередині яких можна упаковувати свої програми та сервіси. Контейнери запускаються з образів і можуть містити один або кілька запущених процесів. Ви можете думати про образи як про будівельний або пакувальний аспект Docker, а про контейнери як про запущений або виконуваний аспект Docker. Таким чином, контейнер Docker – це формат образу, набір стандартних операцій та середовище виконання.

1.2.4 Використання Docker

Очевидно, що ізоляція, яку забезпечують контейнери є відмінною можливістю розгортання пісочниць для різних цілей тестування. Крім того, через свою "стандартну" природу вони також є відмінними будівельними блоками для сервісів. Наведемо деякі приклади використання Docker [14]:

• контейнери допомагають зробити локальний робочий процес розробки і збірки більш швидким, ефективним і легким. Локальні розробники можуть створювати, запускати та спільно використовувати контейнери Docker. Контейнери можна створювати в процесі розробки і розвивати в тестових середовищах і, в свою чергу, у виробництві;

• забезпечує послідовне виконання автономних служб та додатків у кількох середовищах – концепція, особливо корисна у сервісно-орієнтованих архітектурах та розгортанні, які значною мірою залежать від мікросервісів;

• використання Docker для створення ізольованих екземплярів для запуску тестів, наприклад, тих, що запускаються пакетом Continuous Integration (CI), таким як Jenkins CI;

• забезпечує створення та тестування складних додатків та архітектур на локальному хості перед розгортанням у виробничому середовищі;

• створення багатокористувацької PAAS;

• надання легких автономних середовищ ізольованого середовища для розробки, тестування та навчання технологій, таких як оболонка Unix або мова

програмування;

• програмне забезпечення як послуга. Високопродуктивне гіпермасштабне розгортання хостів;

Список ранніх проектів, створених на основі екосистеми Docker, можна побачити в численних оглядах, присвячених цьому предмету (дивиться, наприклад, [14]). Тим не менш, якщо порівнювати з іншими технологією, існує безліч проблем, пов'язаних із зростанням обсягу обчислювальних ресурсів. Іншими словами, нині потрібен розвиток концепції контейнеризації, створення нових механізмів. Тема відкрита для досліджень.

1.3 Оркестрація контейнерів

Як було показано в попередньому розділі, контейнери є ідеальним транспортним засобом для багатьох компонентів завдяки низьким накладним витратам і швидкості розгортання. Крім того, вони також підходять для ефективного горизонтального масштабування шляхом розгортання кількох ідентичних контейнерів відповідного компонента. Таким чином, сучасні програми можуть складатися із сотень або навіть тисяч контейнерів, потенційно зі складними взаємозалежностями. Тим не менш, згідно з цією новою тенденцією розробки додатків, використання контейнерних рішень, таких як контейнери додатків, які ми вже досліджували, є досить обмеженим і важким для прийняття. Таким чином, ці проблеми були вирішені шляхом впровадження вищого рівня контейнеризації, який називається оркестровкою контейнерів [16]. Метою цього розділу є дослідження цього нового рівня та широко поширених рішень, враховуючи той факт, що вони також є фундаментальними у виробничих випадках хмарних обчислень.

1.3.1 Необхідність та структура оркестрації

Коли з’явилися контейнерні рішення, не було інструментів, призначених для оркестровки контейнерів. Сьогодні існує багато таких рішень, як-от Docker Swarm, Kubernetes, Mesos та інші, і тому важко зрозуміти, який з них прийняти для використання. На самому базовому рівні всі оркестратори контейнерів роблять те саме: вони автоматизують надання та керування контейнерною інфраструктурою [16]. Вони були створені для того, щоб мати справу з безперервністю планування, координацією та керуванням, зберігаючи контейнерні компоненти та споживані ними ресурси. Також варто відзначити, що оркестратори не обмежуються строго самими контейнерами. Інші інструменти оркестрування, такі як хмарний оркестратор Juju, було розроблено ще до того, як контейнери стали справді популярними.

Однак оркестратори особливо важливі в контейнерному середовищі, оскільки у нас є багато компонентів, і керувати речами вручну, ймовірно, не вдасться. Таким чином, все більш широке впровадження контейнерних рішень стимулювало впровадження нових можливостей, які можна розрізнити за функціональними та нефункціональними якостями. Серед них ми можемо знайти планування, керування ресурсами та керування послугами. Насправді сценарій – це набір машин, ядро яких містить механізм контейнерів для роботи з контейнерними програмами.

Рисунок 1.5 ілюструє багатошарову структуру оркестратора контейнерів.

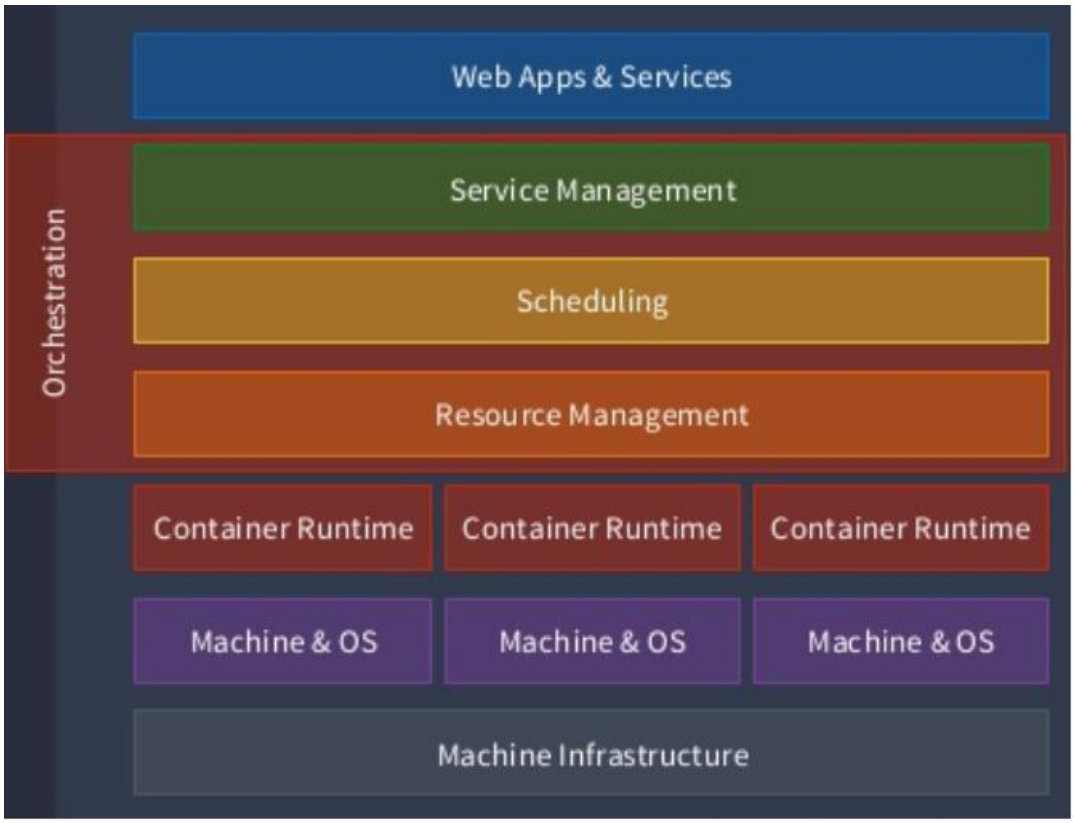


Рисунок 1.5 – Шари оркестровки контейнерів

Оркестровка контейнера дозволяє визначати автоматизоване забезпечення та робочі процеси керування змінами, щоб завжди надавати погоджені політики та рівні обслуговування. На рисунку 1.5 зображено еталонна архітектура механізму багаторівневої оркестровки [17], де набір машин через своє ядро та середовище виконання контейнера реалізує субстрат підтримки. Структура механізму оркестровки знаходиться на вершині та складається з трьох рівнів: керування ресурсами, планування та керування послугами.

Рівень керування ресурсами (Resource Management на рисунку 1.5) керує ресурсами низького рівня; у цьому випадку функціональні елементи – це ресурси, якими можна керувати/компонувати та включають: пам’ять, CPU/GPU, дисковий простір, томи (тобто можливість взаємодії з файловою системою локального хостінгу), постійні томи (тобто, можливість взаємодії також із віддаленою хмарною файловою системою), порт та IP (тобто конфігурація портів UDP/TCP та IP-адрес у віртуальній мережі контейнера). Він спрямований на максимізацію

використання та мінімізацію перешкод між контейнерами, що конкурують за ресурси.

Рівень планування спрямовано на ефективне використання ресурсів кластера. Зазвичай він отримує надані користувачем індикації (наприклад, обмеження розміщення, ступінь реплікації тощо) як вхідні дані, а потім вирішує, як розмістити всі контейнери, що складають програми. Найбільш важливі можливості включають:

- розміщення для прямого управління рішеннями щодо планування;
- реплікацію/масштабування для вираження кількості реплік мікросервісів;
- перевірку готовності включення контейнера лише тоді, коли він готовий відповісти;
- відновлення для повторного запуску швидких довготривалих процесів, робота яких потребує постійної готовності до роботи;
- перепланування для автоматичного перезапуску та планування роботи аварійних контейнерів на несправному вузлі; послідовне розгортання для автоматичного підвищення/зниження версії програми;
- спільне розміщення для затвердження обмежень розгортання, таких як спільне розміщення контейнерів для використання переваг локальної взаємодії між процесами;

Нарешті, рівень управління службами надає функціональні можливості для створення та розгортання (складних) корпоративних додатків. Він керує високорівневими аспектами, що включають:

- мітки для приєднання метаданих до об'єктів контейнерів;
- групи/простори імен для ізоляції контейнерів і підтримки багатокористувацького середовища;
- залежності між мікросервісами;
- балансування навантаження для розділення вхідного навантаження;

- перевірки, щоб зробити додаток доступним у мережі тільки тоді, коли він готовий приймати вхідний трафік;

Далі ми зосередимося на двох оркестраторах контейнерів, а саме Docker Swarm та Kubernetes, які були обрані тому, що вони найпоширеніші на ринку. Слід визнати, що поряд з ними є й інші оркестратори [17]. Так, наприклад, Apache Mesos є дуже значним і основним інструментом цієї області, і, нарешті, Cattle, який є еталонним оркестратором для Rancher. Rancher – це не оркестратор контейнерів, а радше повноцінна платформа управління контейнерами, яка використовується для розгортання та запуску експериментальних результатів.

1.3.2 Docker Swarm

Docker Swarm – це інструмент кластеризації та планування для контейнерів Docker [18]. Це дозволяє IT-операторам керувати кластером вузлів Docker як єдиною віртуальною системою. Це важливий аспект, оскільки він створює кооперативну групу систем, які забезпечують надмірність і включають механізм стійкості до відмови, якщо один або кілька вузлів виходять з ладу. Крім того, інструмент оркестрування надає адміністраторам централізацію, за допомогою якої можна керувати та контролювати всю систему.

Цей оркестратор засновано на моделі головний/підлеглий, у якій головний вузол є основним компонентом кластера. Саме цей вузол відповідає за планування контейнерів, тоді як підлеглий вузол відповідає за запуск отриманих контейнерів. Крім того, Docker Swarm надає можливість додавати або віднімати ітерації контейнерів у міру зміни обчислювальних потреб. Це, очевидно, важливо у хмарних середовищах, де еластичність є однією з найважливіших характеристик. Більше того, Docker Swarm продовжує використовувати стандартний інтерфейс програмування програм Docker для взаємодії з іншими інструментами, такими як Docker Machine. Це означає, що користувач Docker не бачить жодної різниці між роботою з однією машиною або всім кластером.

На рисунку 1.6 показано типову архітектуру Docker Swarm.

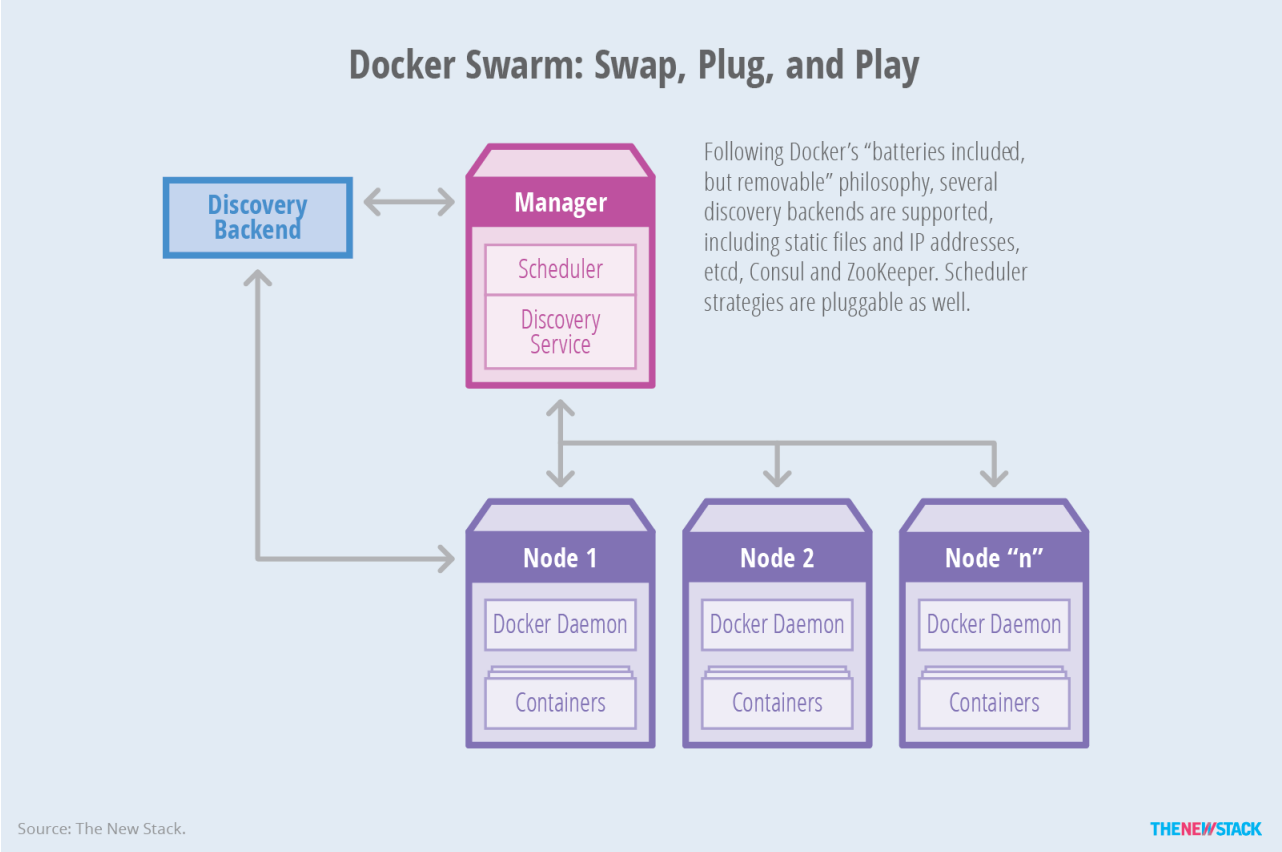


Рисунок 1.6 – Архітектура Docker Swarm (The New Stack)

Кластер — це група серверів та інших ресурсів, які діють як єдина система, що забезпечує високу доступність і, в деяких випадках, балансування навантаженням та паралельною обробкою. Оскільки кластер — це логічна, а не фізична одиниця, розмір кластера може змінюватися. Робота з розподільними системами передбачає довгі затримки і несподівані збої. Створення кластера — це рішення, яке можна використовувати для вирішення цих проблем, використовуючи більш надійне обладнання та найкращі мережеві з'єднання.

З Docker ця стратегія вимагає глибоких обґрунтувань, оскільки організація контейнерів для роботи на безлічі машин – нетривіальне завдання, особливо коли розгортаються різні частини програмного забезпечення на різних машинах. Крім того, з контейнерами Linux для ізоляції і Docker для управління контейнерами, основними проблемами, що залишилися, є ефективність використання ресурсів, характеристики продуктивності обладнання та локальність мережі.

Кластер Swarm складається з двох типів машин: системи, на якій Swarm працює в режимі управління, що називається менеджером, і ще однією, яка називається вузлом Docker, на якій працює агент Swarm. Обидва типи вузлів такі самі, як і будь-які інші машини Docker. Ці агенти не вимагають спеціальної установки або привілейованого доступу до машин, але вони працюють у контейнерах Docker. Для створення кластера потрібно вказати машину, яка має працювати як Swarm manager. Це означає, що певний агент буде розміщено таким чином, щоб увімкнути додаткові функції для забезпечення режиму кластера. Після цього кожен підлеглий вузол потрібно приєднати до менеджера через компонент агента, який в цьому задіяний. Крім того, кожному типу машини в кластері Swarm потрібен спосіб знайти та ідентифікувати кластер, до якого він приєднується.

Як і інші проекти Docker, Swarm дотримується принципу “swap, plug and play” (дивиться рисунок 1.6). Таким чином, можна замінити попередньо визначений сервер планування за допомогою готового рішення. Ця функція є досить важливою, враховуючи, що вона підходить для більшості випадків використання та для розгортання великомасштабного виробництва для більш потужних серверних програм, таких як Mesos.

1.3.3 Kubernetes

Відомо, що зі збільшенням кількості компонентів додатків, які можна розгортати, стає важко керувати ними всіма. Google була першою компанією, яка зрозуміла, що їй потрібен набагато кращий спосіб розгортання програмних компонентів та інфраструктури з метою керування ними, якщо вони збираються вийти на глобальний рівень. Однак це було зумовлено тим фактом, що компанія постійно стикається з виконанням різноманітних завдань на великій кількості серверів. Це спонукало їх створити рішення, які б зробили розробку та розгортання тисяч програмних компонентів керованими та економічно

ефективними. Спочатку вони розробили внутрішню систему під назвою Borg (пізніше змінену на Omega), яка допомагала як розробникам програм, так і системним адміністраторам керувати величезною кількістю програм і служб. Зберігаючи Borg і Omega в таємниці протягом цілого десятиліття, у 2014 році Google представила Kubernetes [19], систему з відкритим кодом, засновану на досвіді, зібраному через Borg, Omega та інші внутрішні системи Google.

Kubernetes – це система управління контейнерними програмами в кластері машин. Вона була розроблена для усунення невідповідності між тим, як спроектовані сучасні кластерні інфраструктури, та деякими припущеннями, які більшість програм мають щодо своїх середовищ. Таким чином, вона дозволяє запускати програмні продукти на тисячах серверних вузлів так, якби всі ці вузли були одним комп'ютером. Користувачам не потрібно бачити рівень інфраструктури, оскільки платформа його абстрагує. Таким чином, розгортання додатків через Kubernetes завжди однаково, немає різниці, якщо розмір кластера постійно змінюється. Як і Docker Swarm (дивиться розділ 1.3.2), Kubernetes засновано на архітектурному шаблоні master/slave («головний-підлеглий»), в якому розробник відправляє список додатків master, а потім платформа піклується про їхнє розгортання на робочих вузлах.

З цієї причини Kubernetes розглядається як операційна система для кластера, оскільки вона показує користувачам увесь набір ресурсів як єдину та центральну точку керування. Крім того, розробникам програм не потрібно впроваджувати певні служби, пов’язані з інфраструктурою, оскільки вони можуть покладатися на Kubernetes для надання цих послуг. Це включає такі функції, як виявлення сервісів, масштабування, балансування навантаженням, самовідновлення, а також вибір лідера. Це робить Kubernetes найпопулярнішим рішенням на ринку оркестровки. Насправді, завдяки цьому корисному набору функцій розробники можуть зосередитися на бізнес-ядрі та не витрачати час на те, як інтегрувати його в інфраструктуру.

Kubernetes — це платформа з відкритим кодом для розгортання програм і

керування ними на основі контейнерів, які виконуються на кластері машин. Навіть якщо система оркестровки контейнерів та весь проект має складну архітектуру, яка спрямована на надання користувачам API, орієнтованих на контейнери, їм не потрібно піклуватися про управління інфраструктурою та компоненти низького рівня, такі як обчислення, сховища та мережі. Архітектура (дивиться рисунок 1.7) заснована на архітектурному шаблоні «головний-підлеглий», а також розроблена з інтерфейсом відкритого рівня.

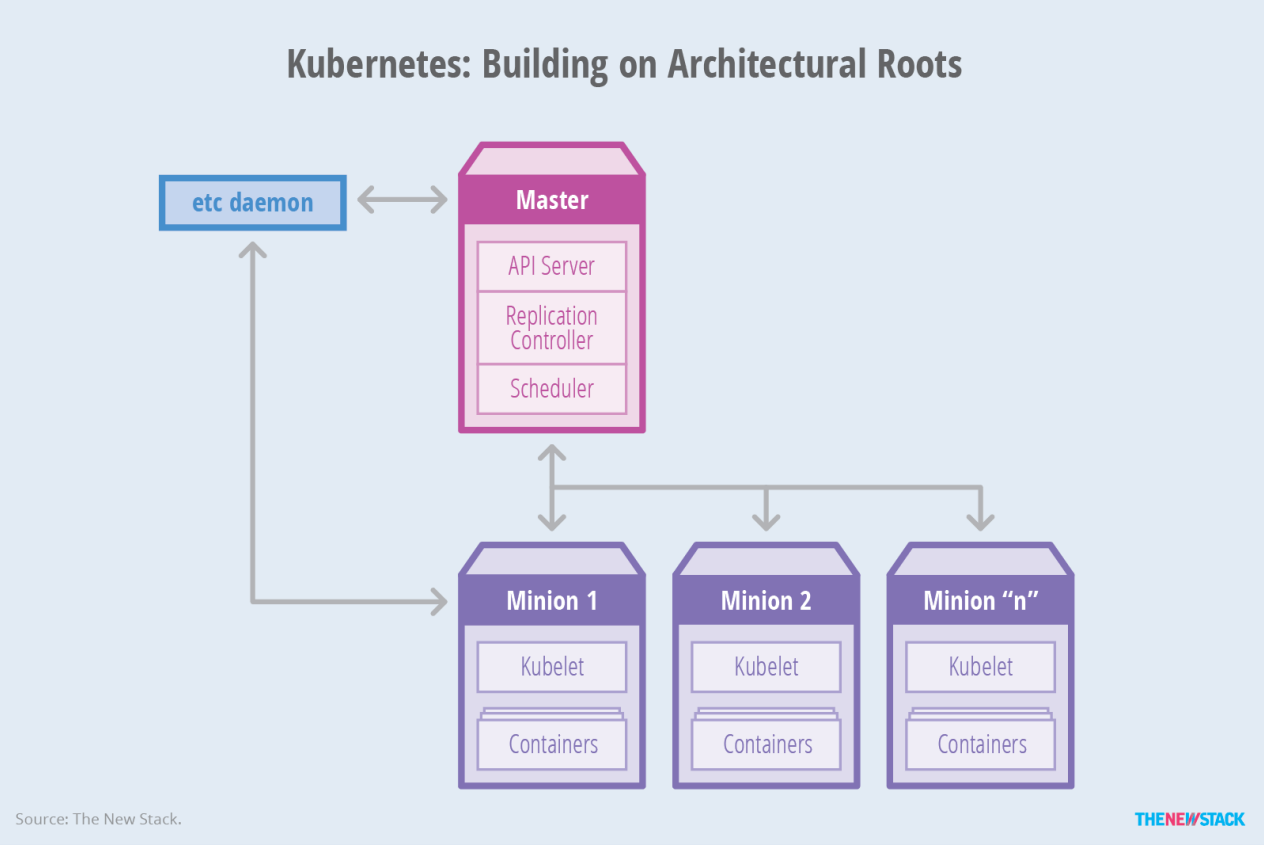


Рисунок 1.7 – Архітектура Kubernetes (з ресурсу The New Stack)

Таким чином, можна налаштувати платформу, розширивши поведінку усіх компонентів. Для цього компоненти не взаємодіють напряму; вони розроблені таким чином, щоб мати відокремлені взаємодії.

Як можна побачити на рисунку 1.7, існує три основні сутності: головний вузол, один або кілька підлеглих вузлів і система постійного зберігання даних. Головний представляє рівень керування кластером, і його можна тиражувати, щоб гарантувати високу доступність і відмовостійкість. Він складається з різних

компонентів, кожен з яких використовується для реалізації певної функції. Одним із них є модуль, який відкриває REST-API. Вони призначені для забезпечення чотирьох основних функцій постійного зберігання, відомих як створення, читання, оновлення, видалення (CRUD, Creating, Read, Update, Delete). Користувачі отримують доступ до цих API і, пройшовши етап автентифікації, можуть працювати з об'єктами Kubernetes, щоб оркеструвати їх у всьому кластері. Таким чином, рівень REST представляє спільну точку, до якої кожен користувач має доступ для роботи з платформою. З огляду на те, що Kubernetes можна легко налаштувати, можливо змінити деякі конкретні частини, такі як механізм контейнера або модуль реплікації.

Іншим фундаментальним компонентом головного вузла є сховище даних стану кластера. Крім того, для забезпечення бажаного стану існує серверний компонент, який називається Controller-Manager. Основні функціональні можливості програми включені в цей компонент. Це окремий процес, і його обов'язки включають життєвий цикл і управління бізнес-логікою. На відміну від інших компонентів, він більш монолітний.

Kubernetes зосереджується на багатоконтейнерних програмах. Концепція мультиконтейнера реалізована на об'єкті платформи, який називається «pod». За дизайном Kubernetes надає контейнеризовану програму як набір контейнерів, кожен з яких є специфічним для окремого мікросервісу. Pod – це набір контейнерів, які залучаються до розгортання як найменшої складової одиниці архітектури. Це означає, що контейнери, розміщені в модулі, будуть розташовані на одній машині кластера. Насправді платформа піклується про весь пакет, а обраний міньйон отримає всю структуру об'єкта Kubernetes. Нагадаємо, що міньйон (Minion) — це робочий вузол, який виконує завдання, делеговані master. Міньйони можуть запустити один або кілька pod. Крім того, вони надають «віртуальний хост» для конкретного додатка в контейнерному середовищі. У кожного міньйона є агент, так званий Kubelet, який керує самим міньйоном і зв'язує його з master.

Користувачі повідомляють Kubernetes, що модуль має бути розташований на машині, яка працює в кластері. Планувальник, ще один компонент головного вузла, приймає рішення про розміщення модуля. Крім того, структура підтримує надані користувачем планувальники, які корисні, коли клієнти хочуть адаптувати поведінку планування до своїх потреб.

Вузол-міньйон складається з трьох основних компонентів: Kubelet, середовища виконання контейнера, Kube Proxy. Kubelet є найважливішим компонентом кожного підпорядкованого вузла, і це агент, мета якого полягає в тому, щоб виконувати дії, пов’язані з платформою. Без присутності цього компонента проект не може працювати як система оркестровки кластера.

За останні роки хмарні обчислення набули великої популярності як серед споживачів, так і серед компаній. Однак ці високі рівні популярності також призвели до зростання онлайн-загроз. Необхідно вжити певних запобіжних заходів, щоб зупинити такі загрози та захистити особисті дані. Дані, що зберігаються в хмарі, захищені за допомогою контейнерної технології. Процедура шифрування та дешифрування даних користувача включена в запропоновану парадигму. Це можна зробити, запустивши контейнер для кожного користувача окремо. Це гарантує ефективне використання ресурсів і збалансованість попиту на різні сервери.

Таким чином, є деякі можливості для оркестрів контейнерів, які можуть бути обрані користувачами. Кожен проект оркестровки має свої переваги та недоліки, і важливо вибрати кращий. Зазвичай рекомендуємо вибрати Docker Compose або Swarm для невеликого розвертання або розробникам для тестування. Для більшого розвертання слід вибрати Kubernetes, який набагато складніший, але використовується для всієї інфраструктури та завдяки модульності може бути розширений іншими опціями. Оркестратори очікують подібного розвитку на шляху, який пов'язаний з проектом OpenStack. Крупні компанії починають випускати власні дистрибутиви, які реалізують кращу підтримку їх існуючих продуктів і послуг. У Kubernetes вже є свій дистрибутив від Canonical і CoreOS.

Redhat побудував всю платформу контейнерів OpenShift на Kubernetes. Віртуалізація контейнерів буде продовжувати змінюватися і розвиватися.

2 АНАЛІЗ ТА ДОСЛІДЖЕННЯ ОРКЕСТРАЦІЇ КОНТЕЙНЕРІВ У ХМАРІ

Технологія контейнеризації допомагає досягти не тільки кращої мобільності та взаємодії, але й кращої продуктивності та ефективності у різних хмарних обчислювальних схемах. Очікується, що така технологія розширить можливості хмарних федерацій за рахунок покращення мобільності та масштабованості в рамках федерації. У цьому розділі ми пропонуємо та досліджуємо архітектуру хмарної федерації, додавши підкомпоненти до еталонної архітектури NIST для ідентифікації ресурсів та управління оркеструванням контейнерів. Пропоновані два нових підкомпоненти забезпечують ідентифікацію ресурсів та оркестрування контейнерів серед членів хмарної федерації. Цими двома компонентами є ідентифікація ресурсів та оркестрація контейнерів відповідно. Компонент з ідентифікації ресурсів визначає відповідного члена хмарної федерації для розподілу завдань на основі попереднього досвіду та поточного стану. Оркестрація контейнерів полегшує управління та оркестрування контейнерів лише на рівні федерації. Ми також визначили кілька методів, які можна використовувати для визначення ресурсів. Серед них обрано метод лінійної регресії для надання ресурсів та ідентифікації членів федерації. Крім того, очікується, що ці методи навчатимуться на основі файлів журналів попередніх виконань та розставлятимуть пріоритети ресурсів на основі поточного стану ресурсів та попереднього досвіду.

2.1 Хмарні обчислення

Парадигма хмарних обчислень [1] виступає за централізований контроль за ресурсами у взаємопов'язаних центрах обробки даних під управлінням одного постачальника послуг. Цей підхід пропонує економічні вигоди за рахунок економії на масштабі пропозиції, зниження дисперсії використання ресурсів за рахунок агрегації попиту, а також зниження витрат на управління

інформаційними технологіями одного користувача за рахунок архітектури, побудованої на декількох орендарях.

2.1.1 Хмарний стек

Хмарні обчислення розрізняють моделі обслуговування: Інфраструктура як послуга (IaaS), Платформа як послуга (PaaS) та Програмне забезпечення як послуга (SaaS) [4] [5]. IaaS пропонує інфраструктурні послуги, такі як Compute Clouds, хмарне сховище, черги повідомлень і т. д. PaaS пропонує повні платформи, стеки рішень та середовища виконання, в той час як SaaS – це модель постачання програмного забезпечення, керована архітектурою, яка розрахована на оренду з багатьма користувачами (multi-tenancy architecture).

Основні моделі послуг IaaS, PaaS і SaaS пов’язані одна з одною та можуть бути організовані як стек. Рівень IaaS представляє найнижчий рівень стеку та дуже близький до основного обладнання. Усередині рівня IaaS можна виділити два типи послуг: обчислювальні та служби зберігання. Типовими представниками інфраструктурних послуг є EC2 від Amazon і S3 від Amazon (дивиться таблицю 2.1).

Таблиця 2.1 – Огляд хмарних продуктів

Ім'я	URL
Amazon’s EC2	http://aws.amazon.com/ec2/
Amazon’s Elastic Beanstalk	http://aws.amazon.com/elasticbeanstalk/
Amazon’s S3	http://aws.amazon.com/s3/
AppScale	http://code.google.com/p/appscale
CloudKick	http://www.cloudkick.com/
Google App Engine	http://code.google.com/appengine/
Microsoft Azure	http://www.microsoft.com/windowsazure/
SalesForce’ Force.com	http://www.salesforce.com/platform/
ScaleUp	http://www.scaleupcloud.com/
SpotCloud	http://spotcloud.com
Zimory	http://www.zimory.com

PaaS представляє другий рівень у стеку. Відомі приклади це Azure від

Microsoft, App Engine від Google, Force.com від SalesForce і Elastic Beanstalk від Amazon. Elastic Beanstalk наразі перебуває на стадії бета-тестування та безпосередньо базується на пропозиціях Amazon IaaS.

Верхні рівні, такі як SaaS (наприклад, Google Docs) і Human as a Service (HuaaS), прямо чи опосередковано базуються на IaaS або PaaS. Деякі додаткові послуги, як-от моніторинг, облік, автентифікація, вимірювання або конфігурація та керування необхідні на кількох рівнях стеку.

2.1.2 Хмарне програмне забезпечення та хмарні продукти

1) Існує широкий спектр програмного забезпечення для приватних хмарних обчислень з відкритим кодом, яке імітує власні системи Amazon, Google і Co. Наприклад, Eucalyptus, AppScale, typhoonAE та OpenNebula (табл. 2.1). Користувачі можуть інсталювати програмне забезпечення з відкритим кодом «вдома» як приватні хмарні рішення. Оскільки таке приватне хмарне рішення частково сумісне з інтерфейсами, протоколами, моделями програмування та параметрами розгортання приватних публічних хмар, це може бути підхід до створення сумісних гібридних хмар, композиції приватних і публічних хмар.

2) Хмарні торгові майданчики та пропозиції федерації: поки ринки, такі як Zimory або SpotCloud, дозволяють торгувати, хмарні ресурси, такі пропозиції, як CloudKick і ScaleUp приймають деякі функції федерації, наприклад, моніторинг і управління підтримкою кількох хмар (табл. 2.1).

2.1.3 Архітектура хмарних обчислень

В основному всю систему можна розділити на основний стек і систему керування. У базовому стеку є три рівні: (1) ресурс, (2) платформа і (3) додаток (дивиться рисунок 2.1).

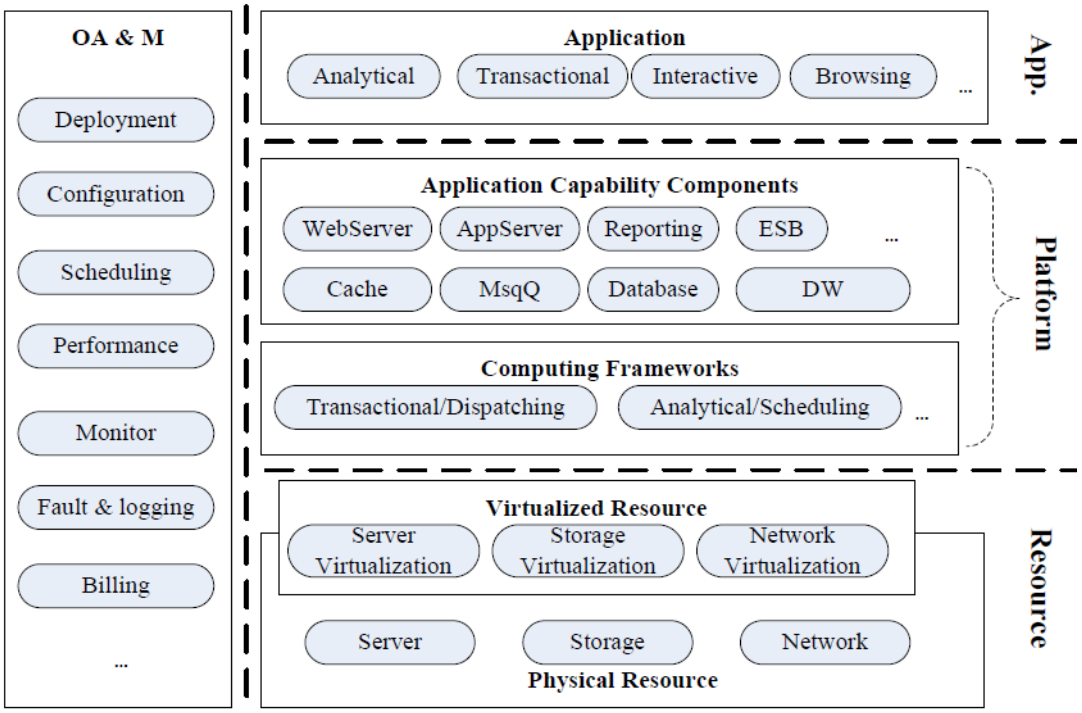


Рисунок 2.1 – Еталонна архітектура

Рівень ресурсів – це рівень інфраструктури, який складається з фізичних і віртуалізованих обчислювальних ресурсів, ресурсів зберігання та мережевих ресурсів.

Рівень платформи є найскладнішою частиною, яку можна розділити на багато підрівнів. Наприклад, обчислювальна структура керує диспетчеризацією транзакцій та/або плануванням завдань. Підрівень зберігання забезпечує необмежену можливість зберігання та кешування. Сервер додатків та інші компоненти підтримують ту саму загальну логіку додатків, що й раніше, з можливістю роботи на вимогу гнучкого керування, так що жоден компонент не буде вузьким місцем усієї системи. На основі основного ресурсу та компонентів програма може підтримувати великі та розподілені транзакції по керуванню величезними обсягами даних. Усі рівні надають зовнішні послуги через веб-служби або інші відкриті інтерфейси.

2.2 Концепція хмарної федерації

Хмарна федерація включає в себе сервіси від різних постачальників, об'єднаних в один пул, що підтримує три основні функції взаємодії – міграцію ресурсів, вихід ресурсів і комбінацію додаткових ресурсів відповідних сервісів. Міграція дозволяє переміщати ресурси, таких як образи віртуальних машин, елементи даних, вихідний код тощо, з одного домену сервісу в інший. У той час як надмірність дозволяє одночасно використовувати аналогічні функції сервісу в різних доменах, а комбінації додаткових ресурсів і сервісів дозволяють об’єднувати різні типи в агрегованих сервісах. Дезагрегація сервісів тісно пов'язана з хмарною федерацією, оскільки федерація спрощує модулярізацію сервісів для забезпечення більш ефективної та гнучкої роботи загальної системи. Ми виділяємо два основні виміри хмарної федерації: горизонтальний і вертикальний. У той час як горизонтальне об’єднання відбувається на одному рівні хмарного стеку, наприклад стеку програм, вертикальне об’єднання охоплює кілька рівнів. Далі ми зосередимося на горизонтальній федерації, яка демонструє деякі особливості, пов'язані з процесами оркестрації контейнерів.

Два типи сценаріїв можна пов’язати з горизонтальною федерацією:

- **надмірність** (redundancy): використовується щоразу, коли існує підмножина належним чином організованих пропозицій послуг, які забезпечують кращу корисність для клієнта, ніж будь-яка окрема пропозиція послуги x_i, тобто існує $X \subseteq \bigcup_{i=1}^{n} x_i$, де для будь-якого x_i тривалість $u(X) > u(x_i)$, принаймні, у найближчій перспективі, постійна, оскільки користувач цілеспрямовано використовує кількох постачальників послуг одночасно;
- **міграція** (migration): може бути ініційована, коли нова пропозиція послуг пропонує клієнту більшу корисність, ніж раніше. Таким чином, існує нова пропозиція x_{new}, така що для любої пропозиції x_i

тривалість $u(x_{new}) > u(x_i) + u(x_s)$, де x_s – загальні витрати на перемикання;

Рисунок 2.2 ілюструє поведінку двох сценаріїв використання послуг хмарної федерації протягом тривалого часу.

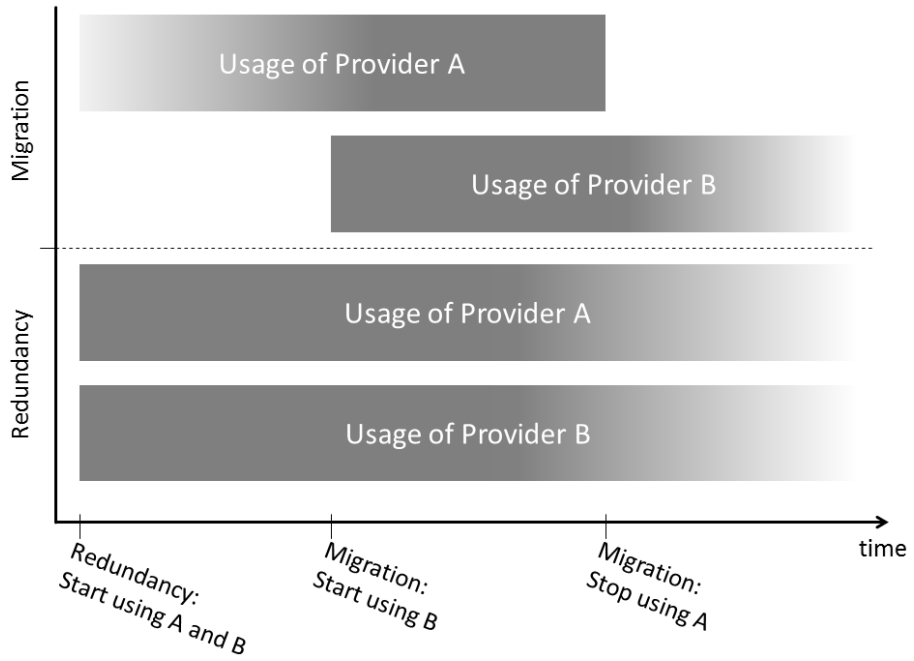


Рисунок 2.2 – Міграція проти надмірності

Застосування різноманітних технологій для хмарної федерації пов'язано з можливістю реалізації цих двох сценаріїв.

2.3 Оркестрування контейнерів у хмарних федераціях

Однією з ключових проблем, яку прагне вирішити хмарна федерація, є масштабованість ресурсів. Певний постачальники хмарних послуг (CSP, cloud service providers) може відчувати брак ресурсів, тоді як інші CSP мають додаткові ресурси, які не використовуються повністю. Вважається, що розподіл ресурсів між CSP може кардинально змінити ситуацію в хмарних федераціях. Серед фреймворків оркестровки ресурсів, які існують на ринку, виділимо фреймворк MiCADO [20]. Нагадаємо, що MiCADO – це мультихмарна оркестровка та

фреймворк автоматичного масштабування для кластерів додатків контейнерів Docker, що працюють на Kubernetes. Однак MiCADO призначено лише для багатохмарних середовищ, а не для хмарних федерацій. Технологія контейнеризації може ефективно підвищити масштабованість, безпеку та управління організованими ресурсами [21]. Контейнери є ресурсозберігаючими одиницями, оскільки для їх роботи потрібні мінімум ресурсів. Образ контейнера інкапсулює всі вимоги та залежності програми в упакований блок. Це робить їх дуже масштабованими та портативними. Хоча керування контейнерами зазвичай є завданням оркестратора контейнерів (наприклад, Kubernetes і Docker Swarm), ці оркестровки працюють лише в межах ресурсів CSP і не можуть поширювати свою функціональність на інші CSP. Як наслідок, це обмежує повне використання технології контейнеризації на рівні федерації. Крім того, впровадження оркестровки контейнерів на рівні федерації має вплив на покращення мобільності та масштабованості додатків. Більше того, вдосконалення можливостей машинного навчання може ефективно оптимізувати ідентифікацію ресурсів. Далі ми вивчаємо архітектуру, яка використовує технологію контейнеризації для хмарних федерацій і дозволяє ідентифікувати ресурси та керувати контейнерами.

2.3.1 Архітектура для організації контейнерів у хмарних федераціях

Щоб використовувати технологію контейнеризації в хмарних федераціях, керування контейнерами вимагає:

- керування контейнерними кластерами, розгорнутими на різних CSP. Це необхідно для підтримки життєвого циклу оркестровки контейнерів (тобто створення, розгортання, оновлення, масштабування та завершення) і покращення моніторингу;
- регулярне відстежування екземплярів контейнерів, що працюють на різних CSP. Необхідно перевірити їх працездатність і вжити

необхідних заходів (наприклад, замінити екземпляри, які вийшли з ладу);

- еластичне забезпечення ресурсів, щоб можна було ефективно керувати та розраховувати навантаження та використання ресурсів;
- наявність компонента, який дозволяє безпечно керувати екземплярами клієнтів, які працюють у федерації;

Виконання цих вимог дає змогу незалежно від постачальника успішно оркеструвати контейнери в межах об'єднання та мати чітке уявлення для клієнтів про те, де працюють їхні контейнери. Крім того, це полегшує керування, моніторинг і виставлення рахунків контейнерними кластерами та примірниками, що працюють у федерації. Еталонна архітектура NIST добре підходить для застосування оркестровки контейнерів для досягнення портативності системи. З огляду на це, застосування контейнерної технології покращує портативність, масштабованість, ефективність і стійкість сервісу в хмарних федераціях.

Національний інститут стандартів і технологій (NIST) визначив чотири компоненти в еталонній архітектурі хмарної федерації (дивиться рисунок 2.3).

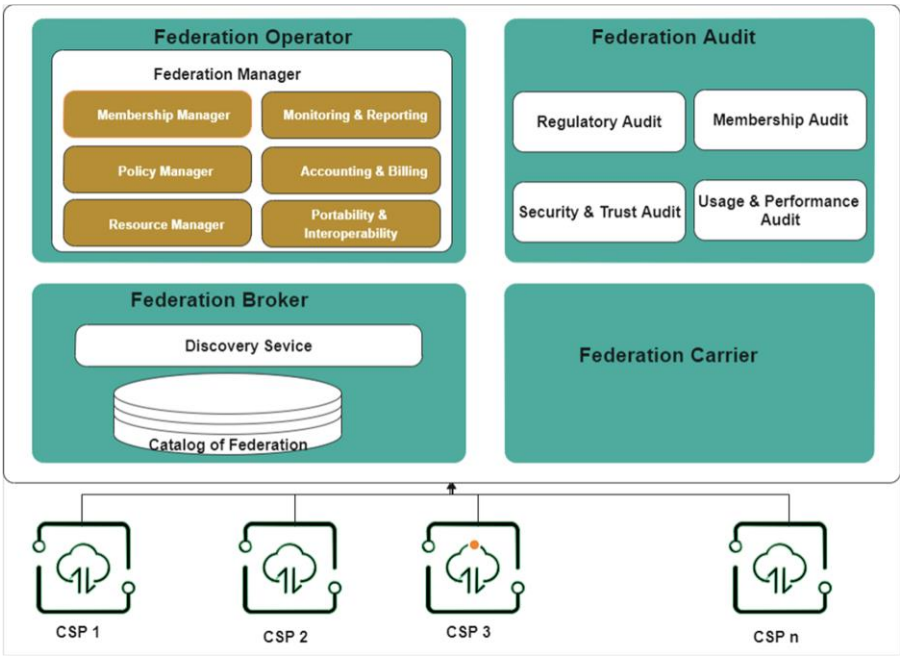


Рисунок 2.3 – Еталонна архітектура хмарної федерації NIST

Це брокер федерації (FO, Federation Broker), оператор федерації (Federation

Operator), аудитор федерації (Federation Audit) і перевізник федерації (Federation Carrier). FO включає в себе шість підкомпонентів, а саме: менеджер членства, менеджер політики, менеджер ресурсів, компонент моніторингу та звітності, компонент обліку та виставлення рахунків і компонент переносимості та взаємодії.

Ці підкомпоненти забезпечують завантаження завдань у федерації; докладні пояснення кожного компонента можна знайти в [22]. В цьому розділі архітектура NIST адаптована для централізованого (тобто центрального брокера) розташування. Ми також припускаємо, що у кожного CSP є інтерфейс менеджера ресурсів (RM, Resource Manager), через який поточні ресурси CSP можуть бути відправлені центральному брокеру.

Для досягнення наших цілей і задоволення вимог пропонується два компоненти для архітектури NIST: ідентифікатор ресурсів (RI, Resource Identifier) і оркестратор контейнерів.

2.3.2 Ідентифікатор ресурсів

Метою даного підкомпонента є виявлення ресурсів та активізація діяльності RM (дивиться рисунок 2.4) шляхом застосування різноманітних технік.

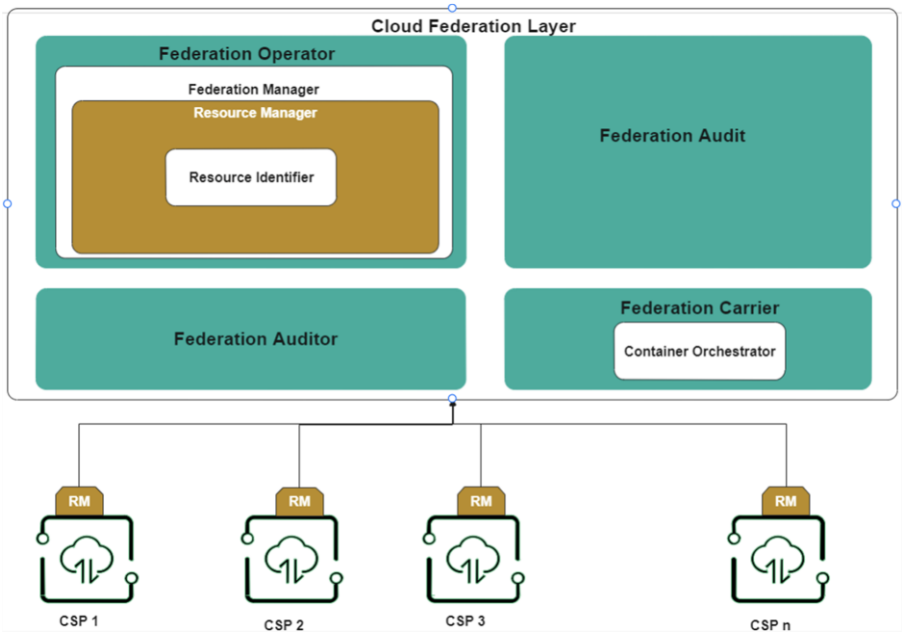


Рисунок 2.4 – Ідентифікатор ресурсу та оркестратор контейнерів у хмарній федерації архітектури NIST

Найпоширенішими методами забезпечення ресурсами є нейронні мережі, лінійна регресія та опорні векторні машини [23]. Раніше використовувались методи машинного навчання для аналізу багатовимірної проблеми розподілу хмарних ресурсів і було підтверджено, що лінійний алгоритм розподілу досягає найкращої продуктивності в багатьох експериментах. Тому в якості відправної точки ми також пропонуємо техніку лінійної регресії для компонента ідентифікатора ресурсу. Ми припускаємо, що надана інформація про ресурси та інформація користувачів.

Інформація про ресурси. Ми припускаємо, що CSP пропонують n типів ресурсів, кожний з яких складається із таких як обчислювальні ресурси, ресурси зберігання та мережеві ресурси. Наприклад, CPU, пам'ять і сховище. Ємність представлена у вигляді вектора $C = (c_1, c_2, ..., c_n)$. Таким чином, це будуть масив із n екземплярів по три властивості в кожному. Враховуючи ці дані на вході, вартість одиниці кожного ресурсу представлена у векторі $P = (p_1, p_2, ..., p_n)$. Наприклад, p_2 означає вартість другого ядра CPU за годину.

Інформація користувачів. Припустимо, що в наборі $U=(1,2,...,m)$ є m користувачів. Окремий користувач $i \in U$. Користувач може подати вимоги, позначені вектором $\mathbf{k}^i=(k_1^i,k_2^i,...,k_n^i)$, де k_s^i представляє ресурси типу s (наприклад, обчислення, сховище, мережа), запитувані i-м користувачем. Цей користувач встановлює ціновий поріг t^i, який представляє готовність користувача стільки платити за вимоги $\mathbf{k}^i$. Загальна інформація про подання i-м користувачем представлена вектором $\mathbf{R}^i=(\mathbf{k}^i,t^i)$. Наприклад, вимога першого користувача – 4 ядра CPU, 16 ГБ пам'яті та 150 ГБ сховища, при цьому користувач готовий заплатити 9$. Ми визначаємо $\mathbf{R}^1=(\mathbf{k}^1,t^1)$, $\mathbf{R}^1=(4,16,150,9)$. У механізмі аукціону постачальник ресурсу переслідує максимізацію суспільного добробуту. Оцінку кожного користувача t^i можна вважати частиною соціального добробуту. Ми позначимо загальний суспільний добробут як V і сформулюємо задачу розподілу ресурсів у хмарних обчисленнях наступним чином:

$$V=\max\left(\sum_{i\in U}(t^i-h_\beta(\mathbf{k}_i))\square x^i\right), \tag{2.1}$$

$$\sum_{i\in U} r_s^i \square x^i \le c_s, \quad \text{де } s=1,2,...n, \tag{2.1a}$$

і $x^i=\{0,1\}$. З часом користувач подає різні вимоги до ресурсів $\mathbf{k}^i$ і відповідні порогові значення ціни t^i, які можна враховувати у функції гіпотези наступним чином

$$h_\beta(k^i)=\beta_0+\beta_1 k_1^i+\beta_2 k_2^i+...+\beta_n k_n^i, \tag{2.2}$$

де $\beta_0,\beta_1,...,\beta_n$ позначають прогнозовану ціну за одиницю з n типів ресурсів, що запитуються i-м користувачем, а β_0 – невизначеність. Іншими словами, ці

параметри виконують роль навчальних параметрів [24]. Змінна $h_\beta(\mathbf{k}^i)$ може розумітися як цінове вираження ресурсів, затребуваних і-м користувачем. Ця функція гіпотези передбачає готовність платити β_i одиниць за кожну одиницю ресурсу. Рівняння (2.1a) вказує на те, що розподіл ресурсів не може перевищувати потужність будь-якого типу ресурсу. Бінарна змінна $x^i = \{0,1\}$ вказує на те, що задоволення вимоги і-го користувача представлено $x^i = 1$ і нуль в іншому випадку. Наведені вище задачі можна трансформувати в задачу про пакування рюкзака (knapsack problem), яка є, як відомо, NP-складною. Назву ця проблема отримала від максимізаційного завдання укладання якомога більшої кількості потрібних речей у рюкзак за умови, що загальний обсяг (або вага) всіх предметів, здатних поміститися в рюкзак, обмежений.

2.3.3 Алгоритми розподілу ресурсів на основі машинного навчання

Ми використовуємо регресію і класифікацію машинного навчання для розробки багатомірних алгоритмів розподілу обласних ресурсів на аукціонах. Основна ідея полягає в тому, щоб вибрати частину вимог від усіх користувачів, вирішити для оптимального розподілу рішень і оптимальної ціни оплати, використовувати лінійну регресію та логістичну регресію для підтримки оптимального розподілу рішень і, нарешті, застосувати вивчену модель для прогнозування всіх вимог користувачів. Було запропоновано [24] два види алгоритмів машинного навчання для підгонки оптимального рішення розподілу. Це алгоритм прогнозування розподілу ресурсів на основі лінійної регресії (Linear-ALLOC) і алгоритм прогнозування розподілу ресурсів на основі логістичної регресії (Logistic-ALLOC).

Під час аукціону користувач виставляє різні вимоги до ресурсів $\mathbf{k}^i$ і відповідні пропозиції t^i, які в даному випадку одержані на основі вивчення ринку хмарних послуг. Іншими словами, передбачається, що сформована база даних

щодо функціонування хмарної федерації.. Як особливості функції гіпотези розглядаємо вимоги користувача до різних типів ресурсів. Далі визначаємо параметри навчання $\theta_0, \theta_1, ..., \theta_n$, а функція гіпотези виглядає як (2.2) $h_\theta(\mathbf{k}^i)$. Суспільний добробут (2.1) підкаже нам розподіл змінної $x^i = \{0,1\}$. Насправді це експериментальні дані. які відображені у базі даних. Далі, як зазвичай у методах машинного навчання, необхідно визначити функцію, яка відображала експериментальні дані та функцію гіпотези. Функція витрат визначається наступним чином (2.3):

$$J(\theta) = \frac{1}{2m} \sum_{i=1}^{m} x^i \left(t^i - h_\theta(k^i) \right)^2, \tag{2.3}$$

З метою мінімізації $J(\theta)$, необхідно вирішити θ, щоб визначити функціональну залежність $h_\theta(k^i)$. Структура алгоритму машинного навчання зображено на рисунку 2.5.

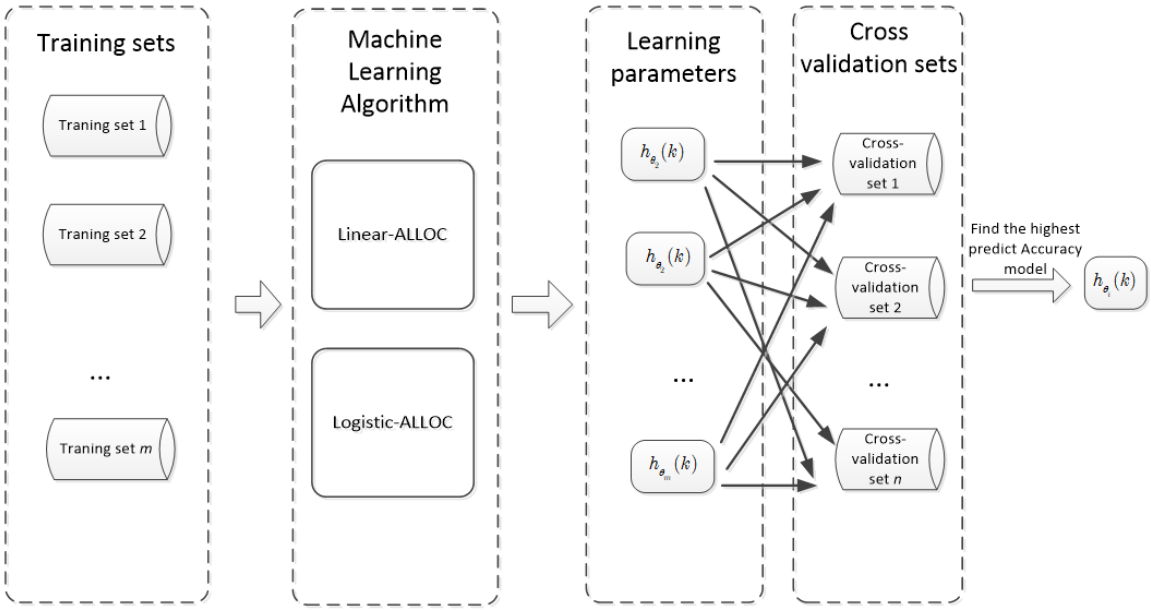


Рисунок 2.5 – Алгоритм машинного навчання менеджера ресурсів (RI)

2.3.4 Результати моделювання

Після отримання оптимальних моделей прогнозування для двох алгоритмів

в роботі [24] випадковим чином згенеровано чотири тестові набори, які включали 1000, 2000, 3000 або 5000 вимог користувачів і відповідні значення ставок.

На рисунку 2.6 ми бачимо, що алгоритми Linear-ALLOC і Logistic-ALLOC у вирішенні соціального добробуту мають близькі показники.

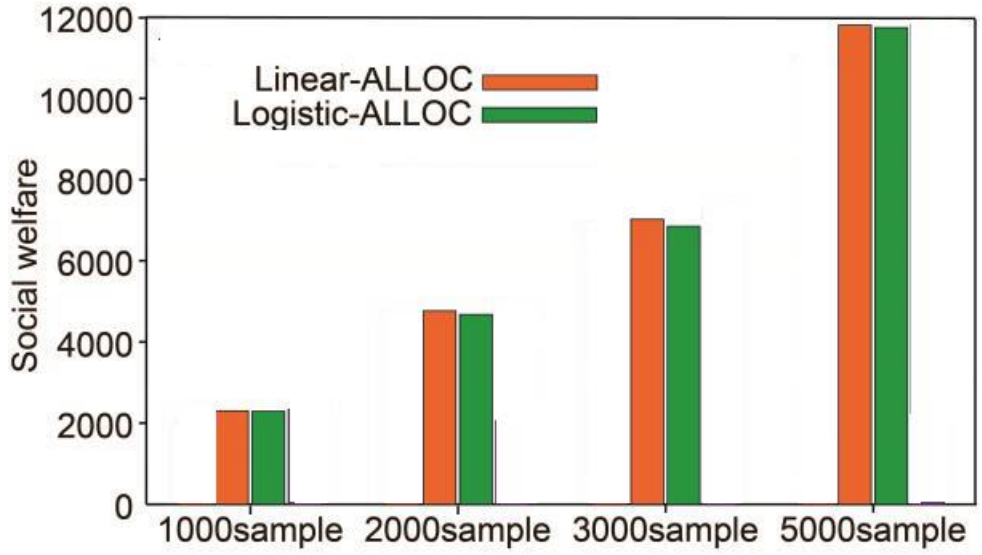


Рисунок 2.6 – Порівняння суспільного добробуту (2.1)

Доведено, що оптимальне рішення розподілу має потенційну модель, яку можна підібрати за допомогою алгоритму машинного навчання. Соціальний добробут, отриманий за допомогою алгоритму Linear-ALLOC, дуже близький до оптимального рішення розподілу.

На основі інформації про ресурси та інформації про користувача мета полягає в тому, щоб визначити відповідні CSP для обробки потрібних запитуваних ресурсів. Компонент RI ініціалізується, коли клієнт отримує запит на додаткові ресурси разом із ціною, яку клієнт готовий платити. Потім компонент RI зв’язується з усіма CSP, щоб дізнатися про їхній поточний статус ресурсу. Після визначення статусу CSP список доступних CSP буде відфільтровано. Якщо перенаправлений список порожній, що означає, на даний момент немає доступного CSP, тоді запитувачу буде надіслано повідомлення про недоступність усіх CSP. Але якщо список не порожній, тоді з бази даних буде витягнуто файл реєстру, який містить історичну інформацію про запитувані ресурси,

запропоновану ціну клієнтом та інформацію про виділений CSP для цього запиту. Після цього будуть перевірені файли журналу, щоб перевірити, чи виконувався той самий запит раніше. Після ідентифікації інформація CSP, яка була раніше вибрана для подібного запиту, буде перенаправлена для фільтрації списку на основі географічно найближчого розташування. Цей крок виконується для економії обчислювальної потужності. Але якщо подібний запит не вдалося знайти в базі даних журналу, то компонент використовує рівняння (2.1), щоб обчислити та передбачити, чи прийме користувач доступні ресурси CSP. Це має місце, якщо ціна наявних ресурсів CSP нижча за розрахункову. Після прогнозування ціни за одиницю та визначення списку CSP, які можуть надавати свої послуги на основі ціни. Потім зі списку вибирається географічно найближчий CSP. Нарешті, ідентифікатор ресурсу повертає вибрану інформацію CSP і оновлює свій файл журналу. На рисунку 2.7 представлено детальну покрокову діяльність компонента RI.

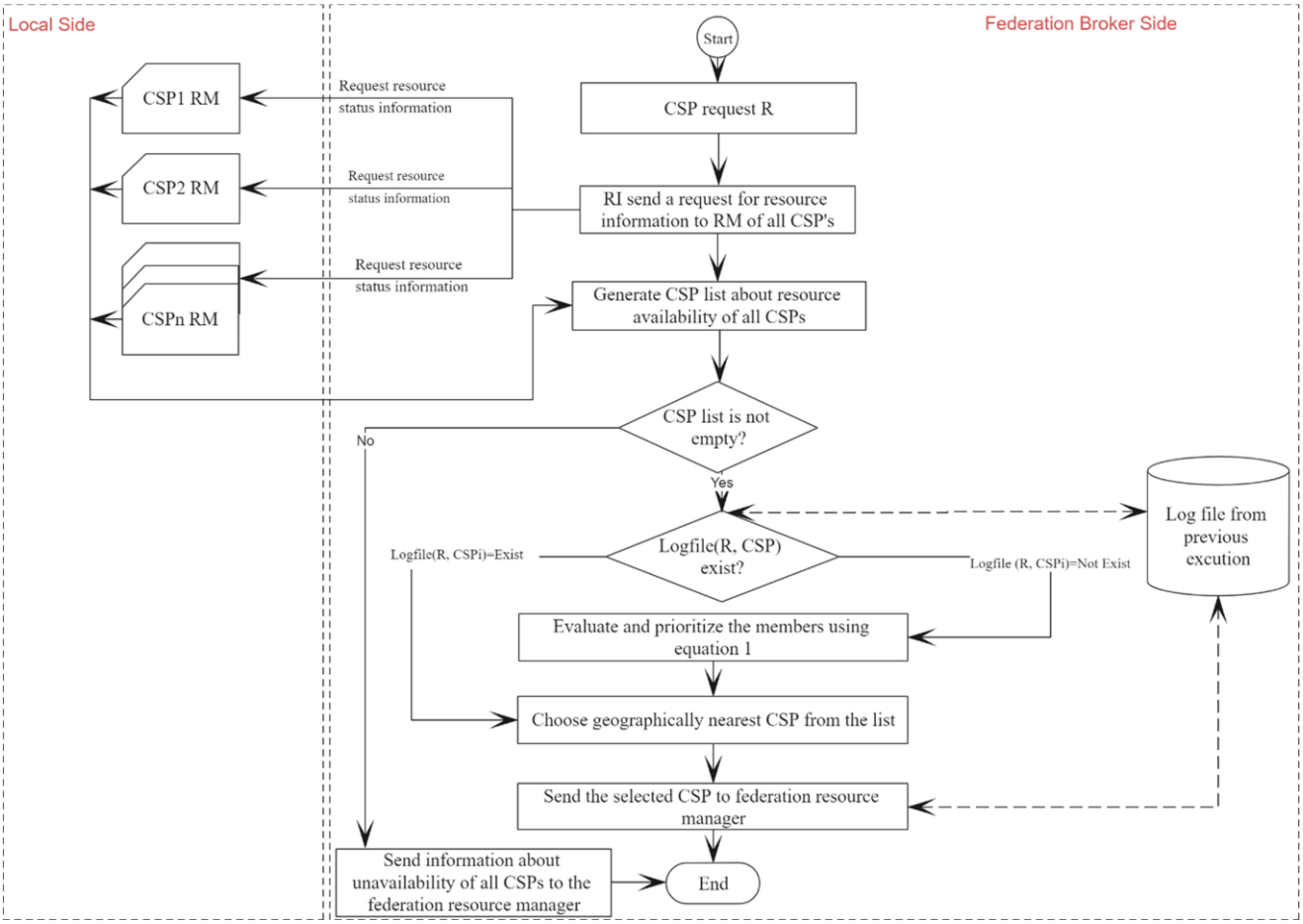


Рисунок 2.7 – Блок-схема алгоритму менеджера ресурсів (RI)

Іншими словами, це завдання тісно пов'язане з проблемою розподілу ресурсів на аукціоні. У такій постановці завдання треба визначити параметри хмарної федерації, що надають найбільш підходящі пропозиції. І ціна пропозиції надається вигодами від її використання. Звичайно, для більш повноцінного аналізу потрібно буде зібрати великий набір даних, часто суто конфіденційних. Тому треба будувати моделі за допомогою методів машинного навчання [24] та аналізувати результати.

2.3.5 Оркестратор контейнерів

Цей підкомпонент пропонується як частина компонента перевізника федерації (дивиться рисунки 2.3 та 2.4). Це полегшує організацію контейнерів у федерації. Щоразу, коли координатор об’єднання отримує запит, наприклад, на звичайні хмарні обчислення, контейнери керуються та контролюються оркестровим пристроєм (наприклад, Docker Swarm, Kubernetes). Однак, оскільки хмарне об’єднання має справу з абсолютно незалежними об’єктами, керування кластерами контейнерів і оновлення головних вузлів, які зазвичай призводять до перебудови образу контейнера, може бути складним завданням з багатьох аспектів. Щоб подолати такі проблеми, оркестратор контейнерів федерації бере повний контроль над кластерами контейнерів, розгорнутими між членами федерації, і керує всім їх життєвим циклом. Іншими словами, створення, розгортання, масштабування та завершення роботи контейнерів в об’єднанні є відповідальністю оркестратора контейнерів.

Оркестратор контейнера ініціалізується запитом від CSP разом із маркером контейнера. Маркер використовується для унікальної ідентифікації кожного образу контейнера. Після отримання запиту цей компонент перевіряє, чи існує образ в сховищі. Якщо так, оркестратор контейнера отримує копію образу з сховища контейнерів. Якщо ні, він отримує доступ до образу контейнера шляхом запиту до CSP. Після отримання образу контейнера від CSP або пов'язаного з CSP

репозиторію, який вибрано ідентифікатором ресурсу для розгортання контейнера.

Рисунок 2.8 візуалізує роботу компонента оркестратора контейнера.

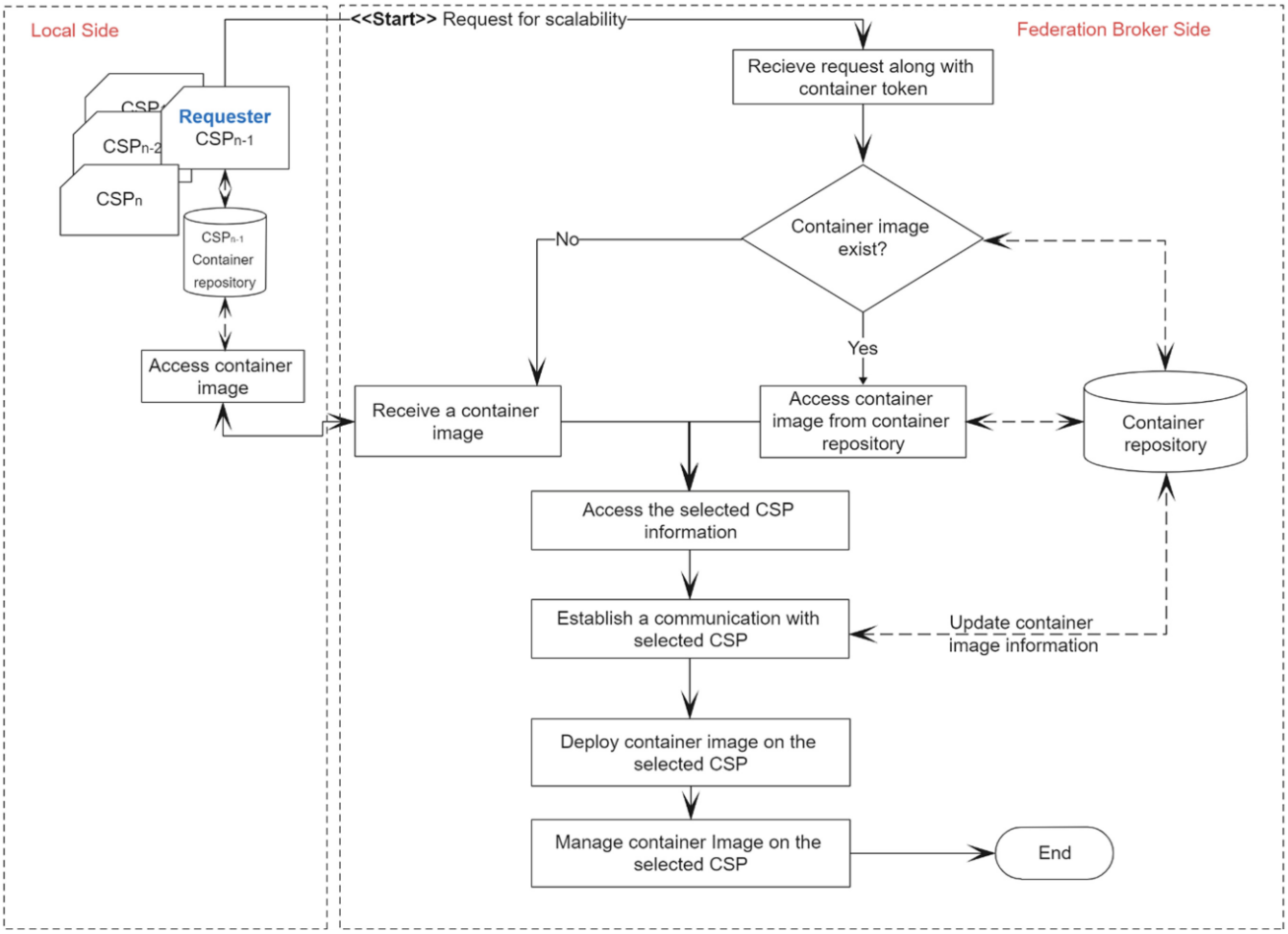


Рисунок 2.8 – Блок-схема алгоритму оркестратора контейнерів

2.4 Особливості контейнерної організації у хмарній федерації

Хмарне об’єднання забезпечує недорогий спосіб максимального використання ресурсів за рахунок підвищеної гнучкості ресурсів, масштабованості та ефективного надання послуг. Оптимізація ідентифікації ресурсів і використання переваг технології контейнеризації в управлінні ресурсами об’єднаного середовища може додати більше переваг хмарним клієнтам і CSP. На додаток до цих переваг: (1) техніка ідентифікації ресурсів відповідає необхідним ресурсам клієнтів у межах бюджету, який вони можуть собі дозволити. У результаті клієнти економлять час і зусилля, оскільки їм не

потрібно шукати та порівнювати ціни на потрібні їм ресурси; (2) Технологія контейнеризації максимізує ефективність використання ресурсів, оскільки контейнери вимагають менше системних ресурсів порівняно з віртуальними машинами, і, що більш важливо, вони дуже портативні, незалежно від того, де вони розгорнуті; (3) Поєднання ідентифікації ресурсів і оркестровки контейнерів у федерації прискорює цикли розгортання та виробництва. Додаткові структури спрямовані на досягнення цієї мети, пропонуючи компонент для ідентифікації ресурсів і компонент для оркестровки контейнерів у хмарних федераціях.

Оркестровка контейнерів вирішує проблему сумісності та мобільності програм, які вже реалізовані в інших хмарних системах. Запропонована архітектура включає оркестратор контейнерів, який є підкомпонентом диспетчера розподілу, відповідає за оркестрування контейнерів між членами федерації. У той час як існуючий оркестратор контейнерів працює на рівні провайдера, запропонований керує кластерами контейнерів на рівні федерації. На високому рівні це дає федерації перевагу максимально ефективного використання ресурсів. На низькому рівні це додає додатку більше гнучкості та підтримує портативність. Додатковою перевагою є те, що він дозволяє автоматизовано керувати додатками, а отже, зменшує накладні витрати, необхідні для керування всім життєвим циклом контейнерів, розгорнутих у федерації. Оскільки оркестровка контейнерів у федерації не була запропонована в хмарній федерації, цей підкомпонент може надати додаткові переваги для федеративних хмар щодо масштабованості, гнучкості та портативності.

Покращуючи еталонну архітектуру NIST, ми додали два підкомпоненти: ідентифікатор ресурсів та оркестратор контейнерів. Ці два підкомпоненти відстежують доступні ресурси в об’єднаній хмарі та використовують цю інформацію для встановлення пріоритетів і прийняття рішення про те, які члени об’єднання мають запускати контейнер. Використовуючи переваги технології контейнеризації, ці підкомпоненти дозволяють керувати та оркеструвати контейнери між членами федерації та, як наслідок, підвищити портативність та

масштабованість програм у федеративних середовищах.

Далі було перетворено проблему розподілу ресурсів хмарних обчислень у задачу регресії або класифікації машинного навчання. Вивчаючи навчальні дані, алгоритм може зробити можливе рішення дуже близьким до оптимального з точки зору точності розподілу забезпечення та використання ресурсів. Перевірено, що справді існує потенційна модель оптимального розподілу ресурсів, яка представляє нове рішення для багатовимірного розподілу ресурсів хмарних обчислень. У цьому розділі ми також обговорювали алгоритм розподілу ресурсів, заснований на машинному навчанні, який задовольняє стратегії механізму аукціону по розподілу хмарних ресурсів федеративних об'єднань.

3 ОРГАНІЗАЦІЯ КОНТЕЙНЕРІВ У ГРАНИЧНІЙ ХМАРІ

Хмарні технології рухаються у бік більшого поширення у багатохмарних середовищах та включення різних пристроїв, як це видно з інтеграції Інтернету речей (IoT, Internet of Things) та мереж у контексті периферійних (граничних) хмарних та туманних обчислень. Для периферійного хмарного середовища оркестрування додатків та сервісів може допомогти керувати та оркеструвати додатки за допомогою контейнерів. Ми розглядаємо вимоги периферійної хмари та обговорюємо придатність контейнерної та кластерної технології.

Хмарні обчислення переходять від централізованих великомасштабних центрів обробки даних до більш розподілених багатохмарних налаштувань, що складаються з мережі вузлів віртуальної інфраструктури. Віртуалізація досягає мережі та дозволяє інтегрувати інфраструктуру IoT. Ці архітектури та їх налаштування часто називають граничними хмарами, периферійними або туманними обчисленнями. У зв’язку з розповсюдженням нам потрібні більш легкі рішення, ніж поточна технологія віртуалізації на основі віртуальної машини (VM). Крім того, ще одним завданням є оркестровка полегшених віртуалізованих середовищ виконання.

Зазначимо, що немає принципової різниці між концепціями звичайної та граничної хмари. В попередньому розділі ми досліджували контейнери, які є легкою концепцією віртуалізації, тобто забирають менше ресурсів і часу. Хоча VM та контейнери обидва є методами віртуалізації, але вони вирішують різні проблеми. Контейнери — це рішення для більш сумісного пакетування додатків у хмарі, тому вони повинні вирішувати насамперед проблеми PaaS.

Відповідно до вимог, які пред'являють граничні обчислення, нам знадобиться більш легкий розподіл упакованих додатків для розгортання та керування. Знову ж таки, рішенням може бути контейнеризація, але її потрібно розширити, щоб задовольнити потребам оркестровки. Щоб оцінити проблеми, ми розглядаємо керування кластерами контейнерів та їх оркестровку в хмарному

середовищі на периферії.

3.1 Граничні хмари

Хмарні периферійні обчислення витісняють обчислювальні програми, дані та послуги з центрів обробки даних на межі основної мережі [25]. Мета полягає в тому, щоб дозволити службам аналітики та генерації знань розміститись там, де джерело даних. Хмарні обчислення на межі мережі підключаються до IoT. Основна хмара забезпечує глобальне керування; периферійні хмари відповідають за локалізовані види (для розміщення служб поблизу або в кінцевих точках мереж), які направлено на пристрої та до рівня приватної хмари. Ми можемо класифікувати розподілені хмари за трьома архітектурними моделями, від тісно пов’язаних до сильно розсіяних:

- хмари з кількома центрами обробки даних із кількома, але тісно пов’язаними центрами обробки даних під контролем одного провайдера;
- слабозв’язані мультисервісні хмари об’єднують сервіси від різних хмарних провайдерів;
- децентралізовані граничні хмари використовують граничні ресурси для надання даних і обчислювальних ресурсів у дуже розпорошений спосіб;

Граничні обчислення мають кілька потенційних переваг:

- затримка мережі зменшується, покращуючи час відгуку;
- зменшені вимоги до пропускної здатності мережі та використання;
- покращена доступність через меншу залежність від підключення до хмари;
- покращений контроль над даними для безпеки та конфіденційності;
- зниження витрат на хмарну обробку;

Концепція туманних обчислень (fog computing) тісно пов’язана з

периферійними обчисленнями. У туманних обчисленнях хмара розширюється до краю мережі шляхом розгортання тих самих інструментів віртуалізації, які використовуються в хмарі, як в інфраструктурі, розташованій поза центрами обробки даних. Подібною технологією, яка викликала великий інтерес дослідників протягом останнього десятиліття, є «хмарлети» (cloudlets), які розширюють можливості мобільних пристроїв шляхом розгортання віртуальних машин на хмарних серверах, розташованих поблизу користувачів.

Іншою парадигмою периферійних обчислень є Mobile Edge Computing (MEC), де периферійні додатки запускаються на мобільних базових станціях з інтегрованою обчислювальною потужністю, перетворюючи операторів мобільних мереж на постачальників хмарних послуг. Мобільні базові станції широко поширені та розташовані поблизу користувачів, що робить їх перспективними для забезпечення граничної пропускної здатності.

Віртуалізація мережевих функцій (NFV, Network Function Virtualization) відокремлює послуги, що надаються мережевою інфраструктурою, від апаратного забезпечення за допомогою віртуалізації. Це привносить гнучкість хмари в мережеву інфраструктуру: наприклад, послуги, які раніше надавалися декількома мережевими пристроями, можна консолідувати в одному апаратному забезпеченні за допомогою NFV, зменшуючи капітальні витрати. Крім того, це забезпечує гнучке розгортання нових служб, таких як служби, які наближають послуги потокового відео до користувачів для покращення якості. Наявність інфраструктури, здатної виконувати віртуалізовані робочі навантаження, створює можливість реалізувати MEC.

3.1.1 Застосування периферійних обчислень

Багато областей застосування потенційно можуть скористатися перевагами периферійних обчислень; вони були досліджені, серед іншого, [26]. У цьому підрозділі представлено підмножину цих програм.

Мережа доставки вмісту (CDN, Content Delivery Network) — це популярна технологія для масштабування веб-сервісів шляхом зберігання даних на межі мережі та уможливила розгортання веб-додатків до глобального масштабу. CDN складається з розподіленої мережі серверів, які кешують веб-вміст. Вміст зазвичай зберігається в центральному місці, звідки його отримує глобально поширений CDN. Основні переваги CDN включають зменшення споживання пропускної здатності, оскільки кілька користувачів можуть спільно використовувати локально кешований вміст; скорочення часу відгуку завдяки розміщенню контенту ближче до споживача; і підвищення надійності, коли центральний сервер недоступний. CDN може відтворювати статичний вміст, такий як веб-сторінки, зображення та відео, а також динамічний вміст, такий як виконуваний код і обчислювальні середовища для запуску програм.

Автомобільні програми, наприклад, уникнення зіткнень та інші рішення для допомоги водієві, і особливо безпілотні транспортні засоби, вимагають надійної обробки даних майже в режимі реального часу і тому не можуть покладатися на підключення до хмари. Крім того, обсяг даних, зібраних в автономному транспортному засобі, наприклад, камерами, занадто великий, щоб відправляти їх у хмару для обробки. Тому єдиний вибір – обробити його на краю мережі.

Відеоаналітика – це область застосування, яка має можливість скористатися перевагами економії пропускної здатності та аспектів конфіденційності периферійних обчислень. Наприклад, відеоспостереження передбачає обробку каналів з великої кількості камер, і надсилання цих даних у хмару може бути дорогим або недоцільним.

Доповнена реальність (AR, Augmented Reality), яка інтегрує віртуальні об’єкти з фізичним світом у реальному часі для створення насиченого інтерактивного середовища, передбачає суворі вимоги до затримки. Крім того, портативні та мобільні пристрої, які використовуються для впровадження систем AR, не мають достатніх можливостей, щоб добре створювати ці середовища. Граничні обчислення можна використовувати в AR, дозволяючи мобільним

пристроям розширювати свої обчислювальні можливості з меншими затримками, ніж це можливо за допомогою хмари.

3.1.2 Вимоги до технології Edge Cloud

Щоб підтримувати саме граничні хмари, нам потрібні, наприклад, визначення місця розташування та розміщення обчислень, реплікація та відновлення. Наприклад, розглянемо програму аналізу вмісту, яка обробляє цифровий мультимедійний вміст, розміщений в Інтернеті. Граничні ресурси знадобляться як для обчислень, так і для зберігання даних, щоб вирішити проблему широкого поширення даних. Необхідні периферійні ресурси можуть бути виділеними ресурсами, розподіленими по мережах розповсюдження контенту (дивиться рисунок 3.1).

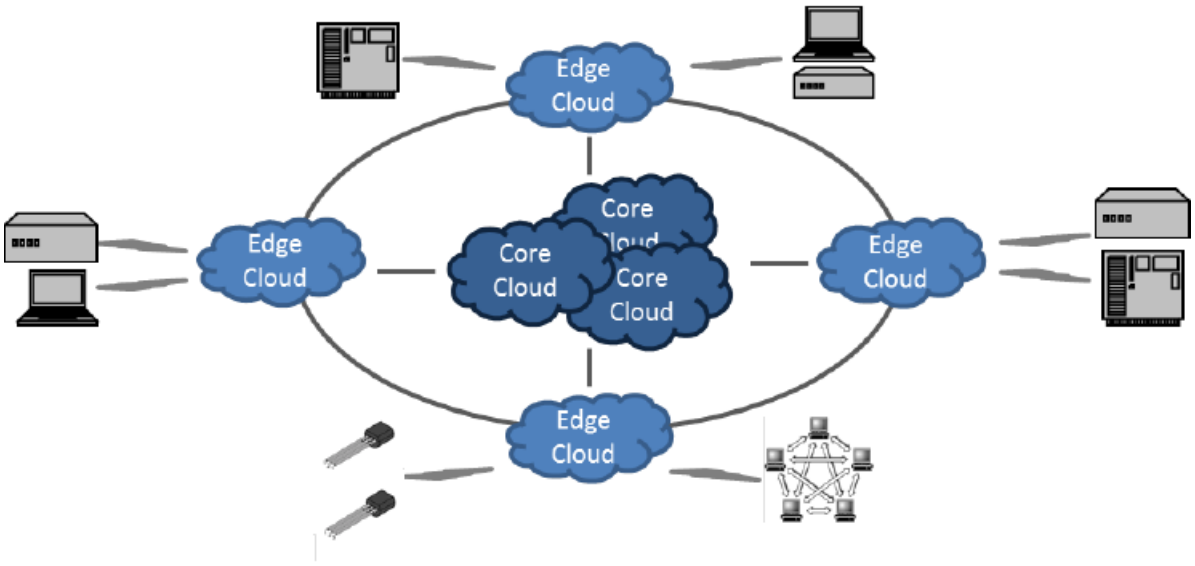


Рисунок 3.1 – Архітектура ресурсів як кластерна контейнерна архітектура

У цьому налаштуванні існують різні віртуалізовані ресурси, які можуть підтримувати периферійні хмари. Для мережі також потрібна певна ємність віртуалізації, як, наприклад, у програмно-визначених мережах (SDN, software-

defined networks). При цьому необхідно підтримувати передачу даних між віртуалізованими ресурсами та надавати обчислювальні та мережеві ресурси між кінцевими пристроями та традиційними центрами обробки даних хмарних обчислень.

Конкретними вимогами, що випливають із цього, є:

1. усвідомлення місцезнаходження, низька затримка та підтримка мобільності для керування хмарними кінцевими точками з віртуалізованими послугами;
2. вузли потребують легкої технології віртуалізації для ефективного розгортання портативних служб на межах;
3. інфраструктури повинна забезпечувати доступ кінцевого користувача через приватні периферійні хмари (які технічно є міні-хмарами);

Їх потрібно налаштувати та оновити безпечним способом – це особливо стосується керування послугами. Нам також знадобиться верхній рівень розробки, щоб надавати та керувати додатками в цих інфраструктурах. Рішення тут можуть включати загальні шаблони топології, контроль життєвого циклу програми та простий у використанні API. Правильні абстракції для управління, орієнтованого на периферійну хмару, на типовому рівні PaaS будуть корисними.

3.1.3 Виклики архітектурних принципів – розробка та експлуатація

Архітектуру, яка вирішує ці завдання, можна організувати в кілька рівнів (знизу вгору):

- унизу мережа розумних речей (мережа розумних датчиків, бездротові мережі датчиків і приводів, мобільні та спеціальні мережі);
- польова мережа (наприклад, 3/4G, LTE, WIFI), а потім основна інфраструктура IP;
- віртуальна обчислювальна/сховище/мережева хмара з додатками поверх;

Експлуатація та керування цією архітектурою дозволяють постачальникам послуг виштовхувати (тобто розгортати) сервіс у відповідних пакетах додатків (наприклад, контейнерах) у кластеризованих периферійних хмарах. Хоча існують деякі рішення, такі як архітектури контейнерів Docker для хмар, все ще існує потреба у специфікації топології та похідних планах оркестрування/хореографії. Існуючі рішення в цій галузі включають Kubernetes, але залишають деякі питання оркестрування без відповіді.

Передбачається, що вимоги до архітектури включають розвертання додатків у кількох оболонках за допомогою облегшеної упаковки додатків, їх подальшого поширення та підтримки вимог до специфікацій топології та управління. Основна ціль міститься в тому, щоб дозволити на цих ресурсах у віртуальній формі розвертати додаткові служби, такі як служби безпеки та аналізу [27]. Зокрема, розробку цих рішень необхідно підтримувати за допомогою оркестровки на основі шаблонів топології, що відображають загальну та еталонну архітектури.

Існує кілька технологій, які можуть сприяти вирішенню цих викликів:

- упаковка програм за допомогою контейнеризації для поширення сервісів та програм на периферію. Docker вже використовувався для цього, надаючи модулі для зв'язку з агентами, які можуть бути середовищами виконання Docker;
- оркестрування може підтримуватися за допомогою специфікації топології на основі шаблонів TOSCA (Topology and Orchestration Specification for Cloud Applications) (дивиться, наприклад, [1]). Загалом композиція сервісів оркестрування має охоплювати весь життєвий цикл — розгортання, виправлення, завершення роботи. Операції зіставляються з керуванням хмарною інфраструктурою, а механізм TOSCA працює поверх цієї периферійної хмарної інфраструктури;

Далі розглядаються периферійні хмари та туманні обчислення з погляду PaaS, зважаючи на легку упаковку додатків та специфікацію топології.

3.2 Кластеризація і оркестровка контейнерів на периферії

Наступне завдання – полегшити перехід від одного контейнерного хоста до кластерів контейнерних хостів, призначених для запуску контейнерних додатків на кількох кластерах у кількох хмарах з метою задоволення вимог периферійної хмари [20]. Вбудована сумісність контейнерів може уможливити це.

3.2.1 Контейнерні кластери у граничній хмарі

Архітектура кластера на основі контейнерів групує хости у кластери. Основні принципи побудови для граничної хмари залишаються такими ж, як і для звичайних хмарних обчислень [16]. Рисунок 3.2 ілюструє архітектурну структуру, що використовує загальні концепції контейнерів та кластерів. Хости контейнерів пов'язані у конфігурацію кластера. Центральними концепціями є кластери, контейнери, служби додатків, томи та посилання.

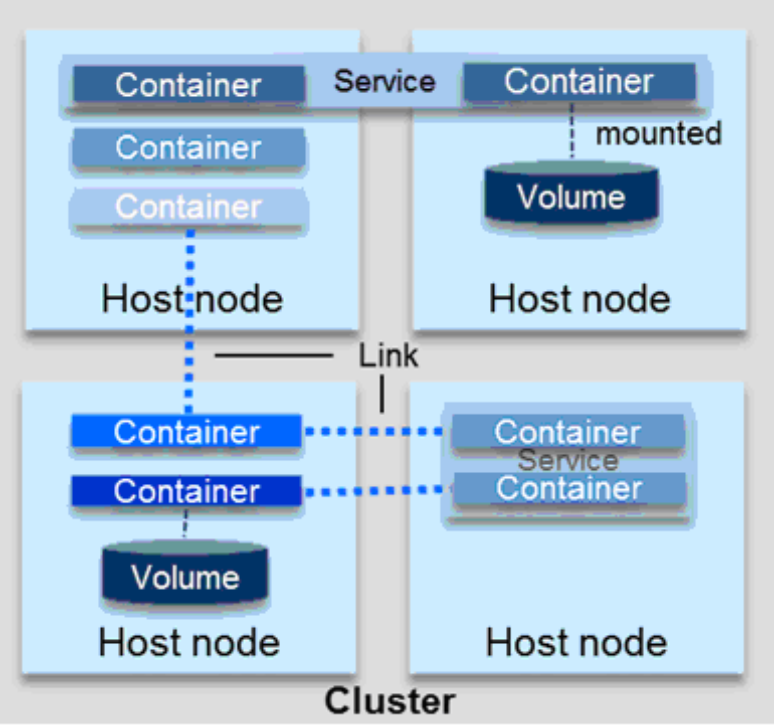


Рисунок 3.2 – Загальна концепція кластерної архітектури

Кожен кластер складається з кількох (хостових) вузлів, де вузли є віртуальними серверами на гіпервізорах. Кожен вузол містить кілька контейнерів із загальними службами, такими як планування, балансування навантаження та самі програми. Кожен контейнер у кластері може містити постійно надані служби, такі як служба корисного навантаження, так звані завдання, є одноразовими службами (наприклад, друк) або функціональними компонентами (служба проміжного програмного забезпечення). Далі служби додатків — це логічні групи контейнерів з одного образу. Служби додатків дозволяють масштабувати додаток між вузлами. Томи використовуються для програм, які вимагають збереження даних. Контейнери можуть монтувати томи. Дані, що зберігаються в цих томах, зберігаються навіть після завершення роботи контейнера. Нарешті, посилання дозволяють двом або більше контейнерам з'єднуватися та спілкуватися.

Результатом цього архітектурного сценарію є рівень абстракції для керування службами на основі кластера, який відрізняється від функцій контейнера, наданих, наприклад, Docker. Архітектура керування кластером має такі компоненти: вузол обслуговування (кластер), прикладний програмний інтерфейс (API, application programming interface), менеджер служби платформи, агент керування життєвим циклом і службу головного вузла кластера.

Розгортання розподілених програм через контейнери підтримується за допомогою віртуального масштабованого сервісного вузла (кластера) з високою внутрішньою складністю (підтримкою масштабування, балансування навантаження, відновлення після відмови) і зниженою зовнішньою складністю. API дозволяє керувати кластерами – від створення служб і наборів контейнерів до інших функцій життєвого циклу. Менеджер служби платформи піклується про пакування та керування програмним забезпеченням. Агент керує життєвими циклами контейнера (на кожному хості). Служба головного вузла кластера — це головний компонент, який отримує команди ззовні та передає їх контейнерам. Це дозволяє розробляти, наприклад, архітектуру периферійної хмари без урахування базової топології мережі та дозволяє уникнути ручного налаштування.

Архітектура кластера складається з механізмів для спільного виявлення служб (наприклад, через спільні розподілені сховища значень ключів) і оркестровки/розгортання (балансування навантаження, моніторинг, масштабування, а також зберігання файлів, розгортання, надсилання, витягування). Полегшена віртуалізована кластерна архітектура, побудована на контейнеризації, повинна забезпечувати низку функцій керування як частину абстракції поверх хостів контейнерів.

Подібно до Docker, Diego, Warden та інших додатків у контейнерному просторі, кілька подібних продуктів з'явилися і в кластерному просторі. Однією з таких платформ керування кластерами є Mesos – проект Apache, який об'єднує розподілені апаратні ресурси в єдиний пул ресурсів. Ці ресурси можуть використовуватися фреймворками програм для керування розподілом робочого навантаження. Це ядро розподіленої системи, що дотримується тих самих принципів, що й ядро Linux, але на іншому рівні абстракції. Ядро Mesos працює на всіх машинах у кластері. Він полегшує додатки з API для керування ресурсами та планування в хмарних середовищах. З точки зору сумісності, він спочатку підтримує LXC, а також Docker.

Ще одне рішення для управління кластеризацією в периферійних хмарах, хоч і на вищому рівні, ніж Mesos, – це архітектура Kubernetes [22]. Kubernetes, підтримуваний Google, можна налаштувати для оркестрування контейнерів Docker Mesos. Kubernetes заснований на процесах, що працюють на хостах Docker. Вони зв'язують хости в кластери та керують контейнерами. Kubernetes конкурує з іншими платформами оркестровки на основі контейнерів. Cloud Foundry, наприклад, використовує Diego як новий механізм оркестровки для контейнерів.

Як і в централізованих хмарних обчисленнях, кластеризовані контейнери в розподілених системах на периферії вимагають розширеної мережевої підтримки. Контейнери і тут надають можливості, які роблять кожен контейнер автономною одиницею. Традиційно контейнери надаються в мережі через адресу загального

хоста. Кожна група контейнерів у Kubernetes отримує власну унікальну IP-адресу, доступну з будь-якої іншої групи в кластері, незалежно від того, розміщені вони на тій же фізичній машині чи ні. Для цього потрібні розширені функції маршрутизації на основі мережевої віртуалізації.

Розподілене керування контейнерами також має вирішувати проблему зберігання даних. Керування контейнерами в кластерах Kubernetes може викликати труднощі з точки зору гнучкості та ефективності через необхідність спільного розміщення модулів Kubernetes з їхніми даними. Необхідно об'єднати контейнер з томом зберігання, який слідує за ним до фізичної машини, незалежно від розташування контейнера в кластері. Для цього необхідно звернутися до топологічних побудов кластерної архітектури.

3.2.2 Оркестровка і топологія

Рішення для управління, яке надають кластерні рішення, необхідно поєднувати з розробкою та підтримкою архітектури. Multi-PaaS на основі контейнерних кластерів – це рішення для керування розподіленими програмними додатками в хмарі, але ця технологія все ще стикається з проблемами. До них відноситься відсутність відповідних формальних описів або визначених користувачем метаданих для контейнерів, окрім позначення образів простими ідентифікаторами. Необхідно визначити топологію архітектур розподіленого контейнера, а також організувати її розгортання та виконання, як зображено на рисунку 3.3.

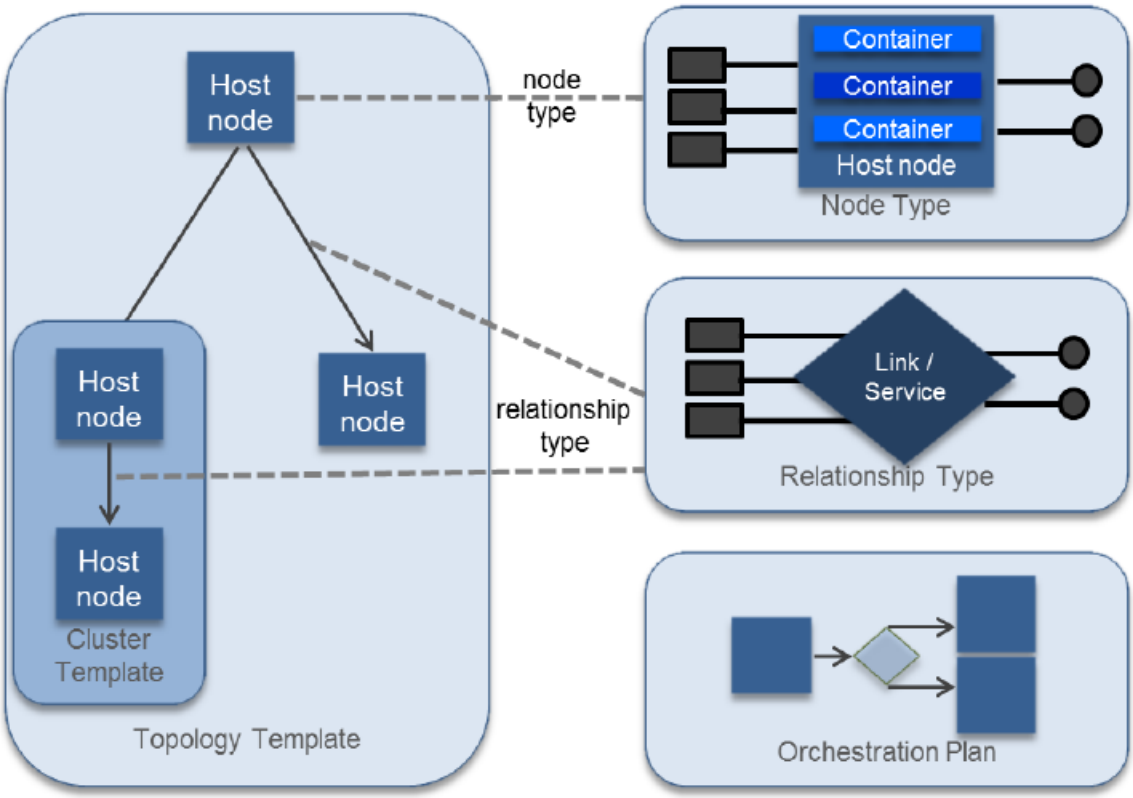


Рисунок 3.3 – Еталонна структура для оркестровки топології кластера

TOSCA при цьому підтримує такі функції:

– взаємодіючий опис хмарних служб додатків та інфраструктури – тут реалізовано як контейнери, розміщені на вузлах крайової хмари;

– зв’язки між частинами сервісу, як показано на рисунку 3.2;

– операційну поведінку цих служб (наприклад, розгортання, виправлення або завершення роботи) у плані оркестровки;

Перевагою цього є те, що він не залежить від постачальника, який створює послугу, а також від будь-якого конкретного хмарного постачальника чи технології хостингу. TOSCA також можна використовувати для пов’язування операційної поведінки вищого рівня з керуванням хмарною інфраструктурою. Шаблони TOSCA можна використовувати для визначення кластерів контейнерів, абстрактних типів вузлів і зв’язків, а також шаблонів стеків програм.

Хмарні додатки можуть працювати в контейнерах TOSCA. Специфікація

топології базується на вузлах (наприклад, сервері або датчику/пристрої) і має інтерфейси життєвого циклу, що дозволяє архітектору визначати, як створювати, налаштовувати, запускати, зупиняти та видаляти ресурси. План оркестровки визначається в YAML і використовується для організації розгортання програм, а також процесів автоматизації після розгортання. План оркестровки описує програми та їхній життєвий цикл, а також зв'язки між компонентами. Сюди входять зв’язки між програмами та місцем їх розміщення. За допомогою TOSCA ми можемо описати інфраструктуру, рівень проміжного програмного забезпечення платформи та прикладний рівень на додаток до них. Продукт PaaS, як-от Cloudify, що підтримує TOSCA, бере план оркестровки TOSCA, а потім запроваджує його за допомогою робочих процесів, які перетинають графік компонентів і видають команди агентам. Ці агенти створюють компоненти програми та склеюють їх разом.

Фреймворки і механізми TOSCA забезпечують оркестровий інструмент, який дозволяє описувати складні топології та розгортання. Визначення спеціальної топології та оркестровки дозволяє вказати, як кожна служба розгортається найбільш ідеальним способом. Для простих мікросервісів без складного внутрішнього стану найкращим підходом є, наприклад, використання Docker і Kubernetes. Проект TOSCA можна використовувати для більш складної топології, яка потребує більшої оркестровки. Прикладами тут може бути реплікований і шардований кластер Mongo DB. Хоча проект TOSCA також можна використовувати для базового випадку контейнера, наприклад, для створення кількох екземплярів образу Docker.

3.3 Аналіз методів управління контейнерними кластерами у граничній хмарі

Для оцінки методів керування кластерами периферійної хмари дослідники часто покладаються на створення спеціальних програм (testbed, випробувальних

стендів), які розроблені для оцінки конкретної техніки. За відсутності загальнодоступної периферійної інфраструктури тестові стенди, поряд з моделюванням і емуляцією, є єдиним способом оцінити системні підходи [27]. Відтворювані експерименти для кластерів периферійних хмар допомагають оцінити методи керування кластерами та включають наступні етапи: (1) налаштування випробувального стенда, (2) створення шаблонів робочого навантаження, (3) керування розгортанням додатків, (4) оркестрування експериментів, (5) прилади та моніторинг, (6) аналіз даних експерименту. Експериментальна та аналітична структура дозволяє дослідникам оцінювати їхні системні підходи, одночасно відкриваючи інші можливості, які важливі для розробки та тестування методів управління кластерами. Експериментальна структура збирає дані в спрощений спосіб і значно скорочує ручні дії та дозволяє масштабувати програми та розміри кластерів.

З цією метою використовується Edge Run [28]. Це набір проектів, які допомагають розробникам створювати, контролювати та тестувати додатки для континуума периферійних хмар. Частиною проекту Edge Run є Galileo – розподілене середовище навантажувального тестування [29].

3.3.1 Експериментальна інфраструктура

У цьому розділі описується мета структури для проведення експериментів для вивчення характеристик кластерів периферійної хмари.

Galileo було створено для розробки та оцінки методів керування кластерами за допомогою Symmetry, спеціальної системи керування кластерами. Основними завданнями Galileo є координація клієнтів для створення запитів, а також моніторинг і зберігання даних моніторингу в примірнику MySQL. Galileo вимагає постійної адаптації та імплементації розширень. Зокрема, для цього проведено його інтегрування в сучасний сервіс оркестрування контейнерів, такий як Kubernetes. Це дозволяє розгортати компоненти фреймворку у кластері,

скорочуючи ручні кроки, необхідні для запуску експериментів.

Експериментальна та аналітична структура пропонує спрощений підхід до виконання та вимірювання контрольних показників. Ми розглядаємо два типи експериментів: сценарій та профілювання.

Експерименти зі сценаріями — це виконання дій з керування ресурсами (наприклад, масштабування). Ці експерименти дозволяють користувачам вказувати кілька екземплярів програми на вузлах. Генерація запитів базується на профілях робочого навантаження, які виконують клієнти Galileo.

Експерименти профілювання подібні до сценаріїв, але пропонують обмежену функціональність і зосереджені на виконанні тільки одного типу програми на одному вузлі. Експерименти з профілювання спрямовані на швидке тестування нових програм для оцінки використання ресурсів і продуктивності на різних пристроях.

Спеціальні програми вимагають реалізації методів виклику відповідної служби та фабрики Kubernetes Pod для образу контейнерів. Це гарантує гнучкість щодо майбутніх додатків і вимагає мінімальних зусиль з кодування.

Методи управління кластерами повинні враховувати мобільність користувачів, оскільки це важлива характеристика периферійних хмарних систем. Наприклад, клієнти пересуваються містом і підключаються до радіовеж, які є першою точкою входу. Для цього є служба оркестровки контейнерів для налаштування кластера. Затримка мережі емулюється за допомогою інструментів Linux і імітує поведінку георозподілених кластерів. Відтворюваність гарантується завдяки використанню також інструментів Linux і служб оркестровки.

Архітектура полегшує генерацію запитів, розміщення програм, розповсюдження повідомлень та зберігання даних. Для цього використовується легковажний дистрибутив Kubernetes для розміщення додатків та клієнтів. Всі вузли приєднуються до одного кластера, що дозволяє уникнути накладних витрат на налаштування федерації кластерів. Кожен кластер має один вузол контролера, який діє як балансир навантаження. На рисунку 3.4 зображена описана система.

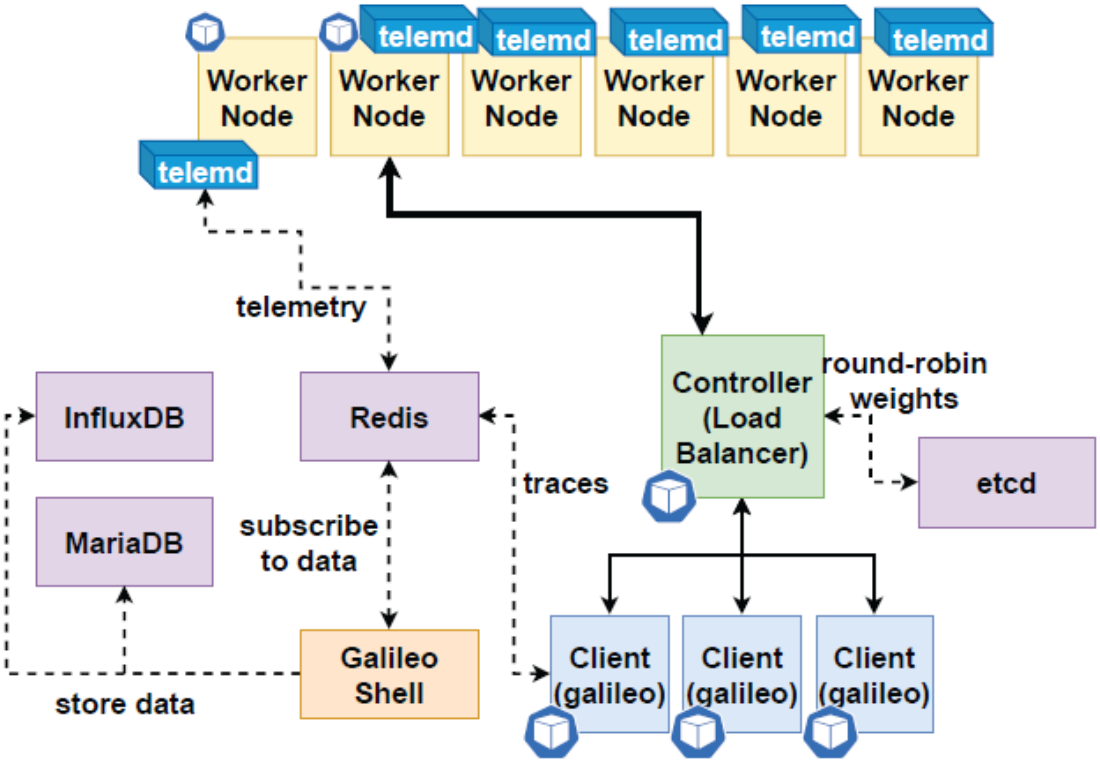


Рисунок 3.4 – Загальний огляд експериментальної установки

Робочі вузли розміщують додатки, в той час як клієнтські вузли запускають Galileo і чекають запитів на генерацію робочого завантаження, опублікованих через Redis. Клієнти відправляють HTTP-запити балансувальника завантаження, який пересилає їх екземплярам додатків. Контролер розміщує екземпляр балансувальника навантаження. Хоча користувачі можуть вибрати використання будь-якого іншого рівня балансування навантаженням. Наша експериментальна структура встановлює початкові ваги для балансувальників навантаження. Користувачі можуть змінювати ваги через etcd, сховище даних ключів, які використовуються Kubernetes для зберігання стану. Кожен екземпляр балансувальника навантаження відстежує зміни екземпляра etcd і таким чином оновлює свої внутрішні ваги. Оболонка galileo є компонентом, який ініціює експеримент і підписується на Redis, а також зберігає дані в InfluxDB і MariaDB для аналізу після експерименту.

Далі пояснюється, як користувачі можуть визначати та запускати

експерименти. На рисунку 3.5 експеримент розбивається на три фази: до експерименту, під час виконання та після експерименту.

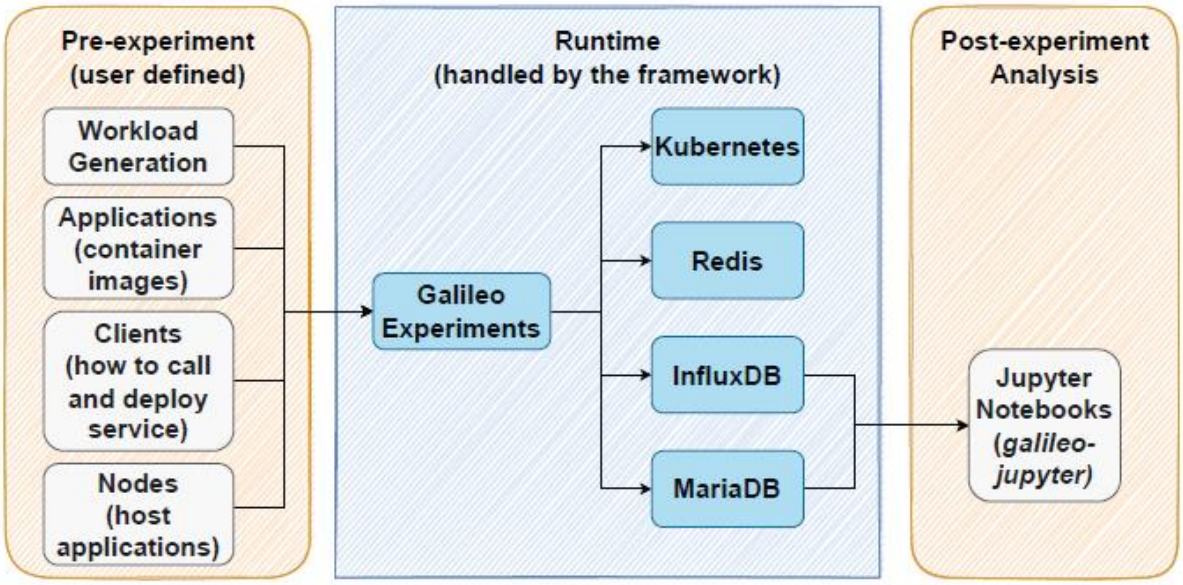


Рисунок 3.5 – Робочий процес експерименту

Користувачі повинні взаємодіяти лише на етапах до та після експерименту, тоді як фреймворк керує системою на етапі виконання. Користувачі мають надати такі вхідні дані: робоче навантаження (тобто список часу між надходженнями), програми (тобто образ контейнера), клієнти (тобто реалізувати метод для створення запиту HTTP), вузли На етапі перед експериментом користувачі можуть створювати робочі навантаження. Фреймворк підтримує два типи робочих навантажень: параметризовані та на основі профілів. Параметризований підхід вимагає трьох аргументів: n кількість запитів, час між надходженнями, і кількість клієнтів. Клієнти є асинхронними та надсилають n запитів із фіксованим часом між надходженнями. Отже, експерименти припиняються, не дочекавшись відповіді.

Варіант на основі профілю очікує масив, який містить час між прибуттями, кожен масив представляє одного клієнта. Наприклад, [0.5, 1, 0.5] надішле три запити: один через 0,5 секунди, інший через 1 секунду та останній через ще 0,5

секунди.

Експерименти з профілю та зі сценаріями мають різні параметри. Другий приймає обидва типи робочого навантаження, тоді як перший підтримує лише підхід на основі профілів. Крім того, експерименти з профілюванням приймають образ контейнера програми, кількість екземплярів програми, хост для профілю та те, в якому кластері клієнти генерують запити. В експериментах зі сценаріями користувачі створюють зіставлення між вузлами і образами контейнерів, включаючи кількість екземплярів. Крім того, експерименти зі сценаріями можуть запускатися в різних кластерах, що дозволяє реалізовувати складніші сценарії (наприклад, міграцію користувачів з одного кластера до іншого).

Після цього користувачі запускають фреймворк і входять у середу виконання. Фреймворк встановлює балансувальники навантаження, створює додатки, налаштовує клієнтів і запускає оболонку galileo для запису всіх даних телеметрії та трасування. Нарешті, на етапі експерименту користувачі можуть аналізувати результати за допомогою galileo-jupyter. Відбувається класифікація даних аналітики за трьома основними категоріями: метадані, трасування та використання ресурсів. Метадані складаються з ідентифікатора експерименту, автора, початку та кінця. Крім того, фреймворк також зберігає інформацію про всі вузли. Сюди входить інформація, детально описана в репозиторії telemd і мітках Kubernetes кожного вузла та їхніх розподілених ресурсів.

На рисунку 3.6 наведено детальний опис однієї траси експерименту.

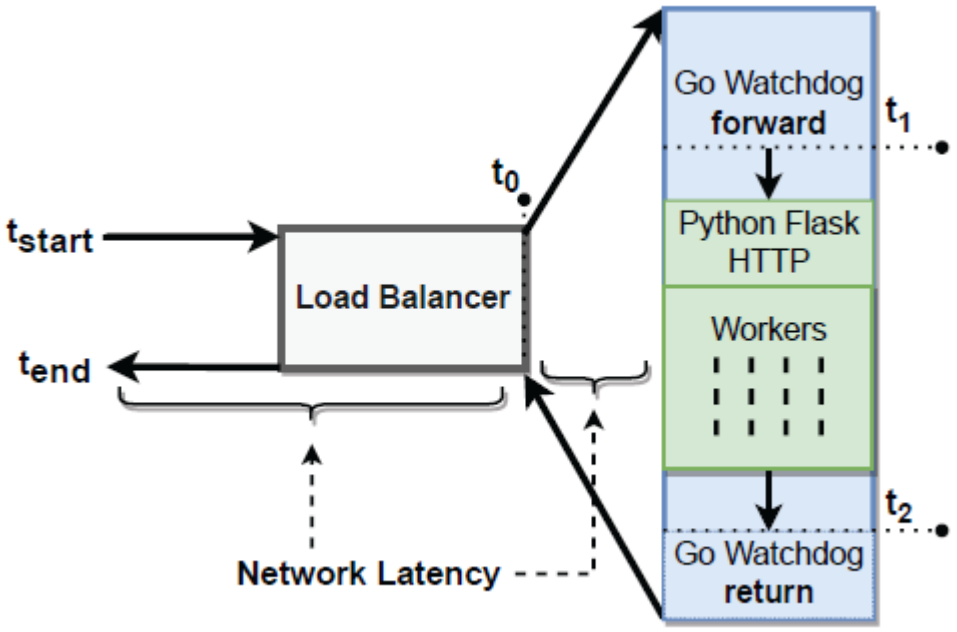


Рисунок 3.6 – Еволюція потоків в експерименті

Насамперед записується: час створення трасування (RTT, round-trip-time), її відправлення клієнтом (t_{start}), її пересилання від балансувальника навантаження (t_0), її отримання функцією t_1, повернення t_2 і повернення до клієнта. На основі цих часових позначок ми можемо розрахувати та оцінити подальші вимірювання: час зворотного зв'язку ($t_{end} - t_{start}$), мережеву затримку між клієнтом і балансувальником навантаження ($t_0 - t_1$), мережеву затримку між балансувальником навантаження та екземпляром програми ($t_1 - t_0$) і час виконання ($t_2 - t_1$). Час виконання включає будь-які постановки в чергу, спричинені статичною кількістю робочих потоків, які сервер Python Flask використовує для обробки запитів. Функції можуть повертати час виконання функції в тілі відповіді. Телеметричні дані складаються з використання ресурсів на основі вузла і кожного контейнера. Вичерпний список моніторингових ресурсів можна знайти у [28], там де *telemd*.

У сценарних експериментах користувачі повинні розгорнути власні методи керування кластером. Вони можуть створювати та демонтувати Pods через API

Kubernetes. Далі *telemd-kubernetes-adapter* публікує ці події, а також *telemd* відстежуватиме новостворені Pods.

3.3.2 Імплементація

В проектах Galileo в основному використовують Python. Лише балансир навантаження, адаптер Kubernetes і агент моніторингу написані на Go. Усі проекти є відкритими та доступні в проекті Edge Run [28]. Далі проведено загальний огляд різних репозиторіїв та їх функціональності.

Galileo робить дві речі: запускає клієнтів і записує дані експерименту. Він пропонує основні робочі інструменти. Користувачі повинні розгорнути налаштування експерименту вручну.

Galileo Experiments містить код, який викликає Galileo для запуску та запису експериментів. Він також містить файли розгортання для необхідних компонентів і детально описує, які інші служби потрібні для початку експерименту.

Galileo Experiments Extensions надає клієнтські реалізації наших функцій і служить точкою входу для виконання експерименту.

Galileo Experiments Functions містить функції на основі OpenFaaS, які можна розгортати та перевіряти за допомогою клієнтів, реалізованих у проекті Extensions.

Galileo Jupyter містить шлюзи, які дозволяють користувачам легко отримувати доступ до даних, записаних під час експериментів. Зокрема, K3sGateway реалізовано для аналізу експериментів Galileo.

Проект galileo-jupyter внутрішньо використовує galileo-db для доступу до сховищ даних. Обидва проекти були розширені для підтримки InfluxDB.

Проект Request Generator надає функції для створення профілів робочого навантаження, які можна використовувати в експериментах.

Go Load Balancer — це зважений циклічний балансувальник навантаження, який стежить за змінами ваги тощо. Він встановлює заголовки HTTP, щоб пізніше

проаналізувати шлях, який пройшов кожен запит, щоб нарешті досягти екземпляра програми.

Telemd використовується як легкий агент детального моніторингу на основі push-розсилки та підтримує моніторинг ресурсів на рівні вузла та програми. Користувачі можуть встановити інтервал для кожного ресурсу окремо.

Фреймворк автоматично розгортає адаптер Telemd Kubernetes, який спостерігає за Kubernetes Pods за допомогою офіційного клієнта Go Kubernetes. Будь-яка подія життєвого циклу Pod повідомляється через Redis і записується під час експериментів. Це дозволяє користувачам аналізувати масштабні події та пов’язувати телеметрію з модулями.

3.3.3 Демонстрація та результати

Експерименти профілювання можуть бути використані для збору показників продуктивності та використання ресурсів, щоб допомогти розробникам краще зрозуміти реалізацію додатків і оцінити, наскільки добре може працювати розгортання. Особливого значення це набуває під час роботи з пристроями з обмеженими ресурсами та сучасними апаратними прискорювачами, де важливо розуміти сутність апаратного забезпечення та програм. Іншим випадком використання експерименту з профілюванням є вимірювання перешкод у продуктивності через багатокористування. Експерименти сценаріїв відрізняються можливостями конфігурації та дозволяють користувачам визначати діапазон початкових екземплярів програми та їх розташування. Платформа не пропонує жодної підтримки методів керування кластером, і користувачі повинні запускати планувальники або програми масштабування, які взаємодіють із кластером. Тим не менш, під час експерименту будь-які зміни в кластерах реєструються, що означає, що користувачі можуть виконувати дії з керування кластером під час виконання та використовувати побудовану структуру для аналізу експериментів. Наприклад, техніка аналітики полягає в тому, щоб

відобразити, коли та де екземпляри програми були створені та закриті.

Для ілюстрації додаток розгортається на основі функцій OpenFaaS [30], яка обслуговує нейронну мережу Mobilenet [31] з використанням TFLite в режимі CPU. TFLite — це версія Tensorflow, адаптована для пристроїв з обмеженими ресурсами, наприклад, на граничній хмарі. Функція виконує класифікацію об'єктів на зображеннях, закодованих в base64. Ми також включаємо оцінку використання ресурсів telemd, як агента моніторингу, єдиного компонента експерименту, що працює на конечних пристроях (за рахунок kubelet).

Для моделювання використовуються параметри пристроїв Nvidia Jetson [32]. Компанія Nvidia пропонує сімейство вбудованих обчислювальних модулів у формфакторі SoM (System On Module), які орієнтовано на створення компактних і енергоефективних систем машинного навчання. Модулі Nvidia Jetson – це компактні плати, що містять на борту всі компоненти повноцінного комп'ютера: процесор, відеоядра, оперативну пам'ять, USB-контролери і тощо. Вони призначені для вбудовування в інші плати (carrier board), розроблені під конкретні завдання. Відомо, що використання SoM значно спрощує розробку систем, так як виробнику специфічного рішення потрібно розробити тільки плату з обв'язкою (carrier board) для периферії і вставити готовий обчислювальний модуль. Це дозволяє знизити витрати на розробку складних материнських плат та сфокусуватися на якості збирання та додаткових опціях. Також це простіше для розробників, оскільки вони можуть використовувати той самий модуль SoM у вигляді Evaluation Kit, поки фінальний пристрій ще не готовий. У результаті розробник ПО отримує передбачуване апаратне оточення і може бути впевнений, що при перенесенні програм на фінальний пристрій він отримає таку саму продуктивність. Це особливо важливо при розробці систем машинного навчання, коли результат залежить від характеристик заліза. Переваги використання таких пристроїв на граничній хмарі полягають в тому, що вся попередня обробка даних відбувається на локальному периферійному пристрої. В результаті пересилання даних на сервер дорогим інтернет-каналом зводиться тільки до відправки

короткого повідомлення після передобробки. Це дозволяє скоротити витрати на мобільний трафік у рази.

Для моделювання вибрано три пристрої з лінійки продуктів Nvidia Jetson, назви та параметри яких наведено у таблиці 3.1. Кластери організовані на основі IoT Box, принципи моделювання яких можна знайти у [32].

Таблиця 3.1 – Параметри пристроїв Nvidia Jetson

Пристрій	Архітектура	CPU	Пам'ять
NX	Arm64v8	4x Cortex @ 2 Ghz	8 GB
TX2	Arm64v8	4x Cortex @ 2 Ghz	8 GB
Nano	Arm64v8	4x Cortex@1.4 Ghz	4 GB

Нагадаємо, що архітектура Arm — це сімейство архітектур із набором спрощених команд (Reduced Instruction Set Architecture, RISC) із режимами простої адресації. Цифра 64 означає, що тут підтримуються 64-бітові регістри. A, R і M – це три архітектурні профілі. «A» в Armv8 позначає "Application Profile". Також є профіль «R» для додатків із вимогами системи реального часу та профілі «M», які найчастіше використовують у мікроконтролерах, де відсутні блоки керування пам'яттю.

Метою експерименту з профілювання є оцінка продуктивності та використання ресурсів на різних пристроях. Один клієнт надсилає 100 запитів з інтервалом між надходженнями за одну секунду. Ми проводимо експеримент на трьох пристроях: NX, TX2 та Nano.

На рисунку 3.7 показані ресурси CPU, використовувані контейнером, RTT в мікросекундах і використання пам'яті в мегабайтах для кожного пристрою.

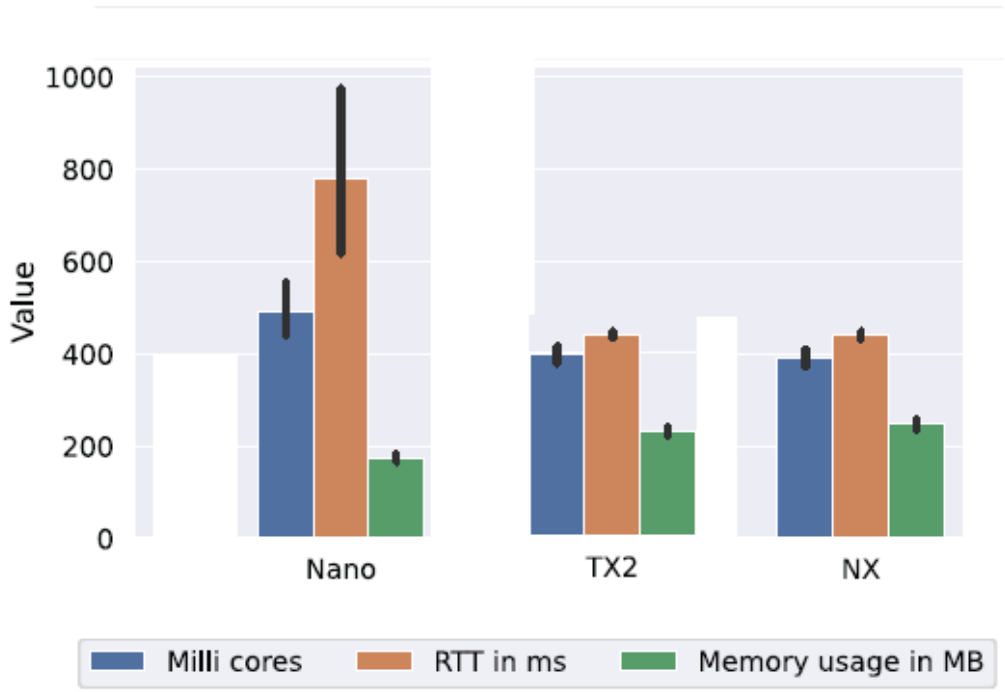


Рисунок 3.7 – Показники пристроїв Nvidia Jetson на мережі Mobilenet

Нагадаємо, що ресурси CPU вимірюються у CPU одиницях. Так, один CPU в Kubernetes відповідає одному значенню віртуального ядра (vCore) на Amazon Web Services (AWS), Google Cloud Platform (GCP) і Microsoft Azure.

Дробові значення можливі. Контейнер, що просить 0.5 CPU, отримає наполовину менше ресурсів, ніж контейнер, що запитує 1 CPU. Можна використовувати закінчення m для позначення тисячної частки. Точність вище 1m не підтримується. Планки похибок вказують на 95-й середній довірчий інтервал. В той час, як NX і TX2 мають порівнянну продуктивність, у вузла Nano найвища RTT. Таким чином, Nano працює гірше, а викиди сягають 6 секунд. Значення 1000 CPU відповідає одному повністю завантаженому ядру ЦП. Nano знову працює гірше та має найвище використання ЦП. Цікаво, що пристрої NX і TX2 мають найвище використання пам'яті, у той час як Nano найнижче. Проект galileo-jupyter пропонує безліч зручних функцій (наприклад preprecessed_traces, preprocessed_telemetry), які дозволяють розробникам будувати цю діаграму на основі отриманих результатів (наприклад, за допомогою Jupyter Notebooks).

Тепер перейдемо до сценарного експерименту. Зокрема, ми запускаємо трьох клієнтів, які породжують 60 запитів з інтервалом між надходженнями за 1 секунду. Два клієнти знаходяться в кластері IoT-Box, а один – у кластері Cloudlet. Ми також запускаємо програму Python, яка випадково масштабує кластер вгору і вниз з інтервалом в 5 секунд. На відміну від горизонтального масштабування, коли додаються нові елементи, додається додаткова функціональність без зміни числа елементів. Із ймовірністю 10% нічого не відбувається, з ймовірністю 40% запускається один новий екземпляр функції, а з ймовірністю 50% відключається один екземпляр. Всі балансувальники навантаження розподіляли запити по колу з Cloudlet до Cloud. На рисунку 3.8 показано, скільки реплік функцій було запущено у кластері.

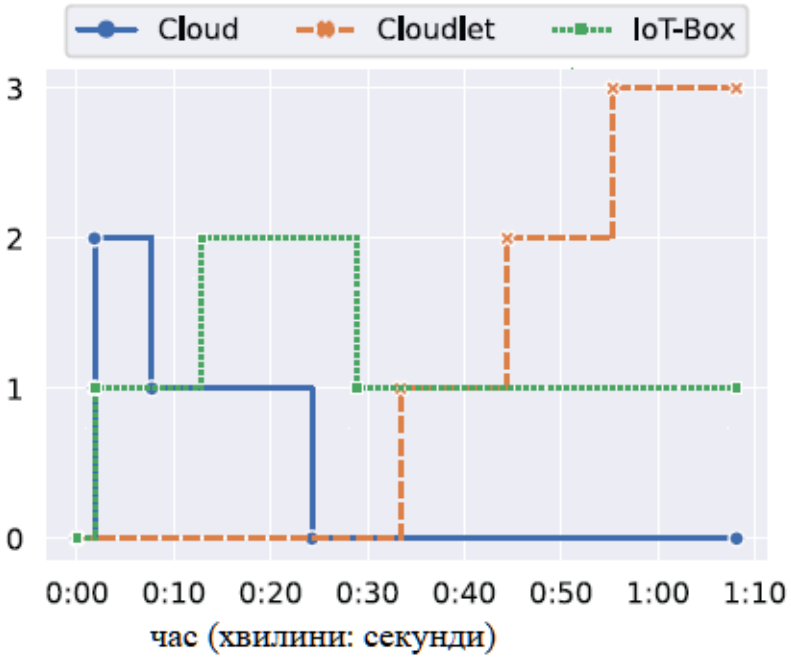


Рисунок 3.8 – Кількість реплік функцій у кластерах

Цей графік ілюструє поведінку методів управління кластером. Крім того, ця структура дозволяє користувачам аналізувати, де були генеровані запити та де вони були оброблені. Таким чином, ця структура може виконувати експерименти, які дозволяють обробляти запити в континуумі периферія-хмара, та пропонує функціональні можливості для їх аналізу. Як і раніше, galileo-jupyter надає зручні

функції для підтримки користувачів у розробці та тестуванні методів керування кластером.

Далі ми показуємо використання ресурсів агентом моніторингу, який кожну секунду видає метрики ресурсів. На рисунку 3.9 показано результати, записані під час 100-секундного експерименту.

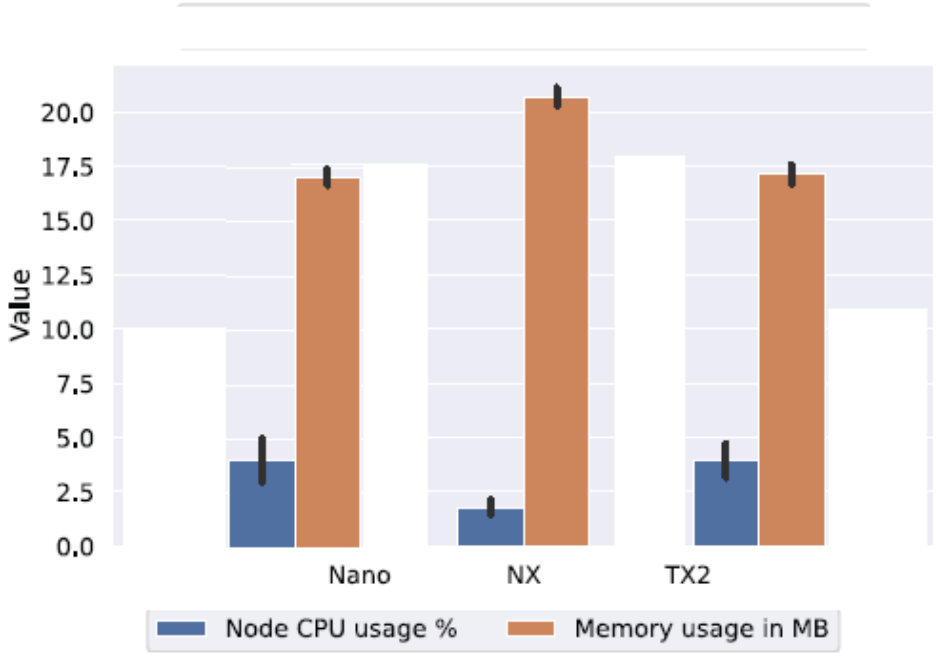


Рисунок 3.9 – Показники пристроїв Nvidia Jetson на мережі Mobilenet

На рисунку показано відносне використання CPU контейнера telemd. 100% використання CPU означає, що контейнер (тобто telemd) повністю використав усі ядра пристрою. На рисунку 3.9 також показано використання пам'яті у мегабайтах. Обидва результати показують, що використання ресурсів є відносно низьким на всіх протестованих пристроях. Користувачі можуть встановлювати різний інтервал моніторингу кожного ресурсу, щоб ще більше скоротити використання ресурсів.

3.3.4 Висновки з результатів експериментів

Оцінка систем периферійних обчислень є складним завданням, враховуючи

як зародження області, і неоднорідність інфраструктури, де працюють такі системи. Тести та випробувальні стенди, що використовуються для оцінки методів управління кластером, часто адаптовані до конкретної техніки, що ускладнює їхнє відтворення і, отже, порівняння підходів. Ми використовували Galileo, систему для розподіленого тесту навантаження, яка дозволяє користувачам визначати і запускати відтворювані тести. У Galileo відсутні інструменти для операціоналізації експериментів, користувачі повинні вручну розгортати інші інструменти для масштабування робочих процесів, інструментування випробувального стенду і збору даних. У цій роботі розширені можливості Galileo, щоб використовувати Kubernetes для розгортання компонентів середовища виконання, а також використані інструменти для надання експериментальної та аналітичної структури для кластерів периферійної хмари. Майбутня робота включає підхід на основі коду для визначення випробувальних стендів і динамічного створення різних налаштувань кластера з різною затримкою мережі.

ВИСНОВКИ

У роботі проведено аналіз шляхів підвищення ефективності побудови сучасних алгоритмів оркестрації контейнерів у системах хмарних сервісів. На основі огляду різних напрямків досліджень за контейнерною технологією, які актуальні для хмарних обчислень, проведено порівняльний аналіз контейнерних технологій, показано переваги Docker як основної платформи для розробки, доставки та експлуатації додатків. Показано основні можливості для оркестрації контейнерів, які можуть бути обрані користувачами у кожному конкретному випадку. З цього погляду кожен проект оркестровки має свої переваги та недоліки, і важливо вибрати кращий. Рекомендовано вибрати Docker Compose або Swarm для невеликого розгортання додатків або для розробників, які проводять їх тестування. Для більш широкого розгортання слід вибрати Kubernetes, який набагато складніший, але використовується для всієї інфраструктури та завдяки модульності може бути оснащений іншими розширеними опціями. На основі порівняльного аналізу різних існуючих механізмів для оркестрації контейнерів у хмарних додатках ми рекомендуємо використовувати Kubernetes.

Серед фреймворків оркестровки ресурсів, які існують на ринку, виділимо фреймворк MiCADO. Нагадаємо, що MiCADO – це мультихмарна оркестровка та фреймворк автоматичного масштабування для кластерів додатків контейнерів Docker, що працюють на Kubernetes. Запропоновано два нових підкомпоненти в еталонну архітектуру NIST хмарної федерації. Ці нові компоненти призначені для ідентифікації ресурсів та управління оркеструванням контейнерів.. Інформація, що дається для ідентифікатора ресурсів, розглядається в рамках двох напрямків: інформація по ресурсах та інформація користувачів. Оскільки у механізмі аукціону постачальник ресурсу переслідує максимізацію суспільного добробуту, оцінку кожного користувача можна вважати частиною соціального добробуту. Існує два види алгоритмів машинного навчання для підгонки оптимального рішення розподілу. Це алгоритм прогнозування розподілу ресурсів на основі

лінійної регресії (Linear-ALLOC) і алгоритм прогнозування розподілу ресурсів на основі логістичної регресії (Logistic-ALLOC). В обох випадках користувач подає різні вимоги до ресурсів і відповідні порогові значення ціни, які можна враховувати у лінійній функції гіпотези. Для моделювання використано чотири тестові набори, які включали 1000, 2000, 3000 або 5000 вимог користувачів і відповідні значення ставок. Проведено порівняння суспільного добробуту на основі цих двох алгоритмів. Показано, що із зростанням кількості вимог користувачів ефективність вкладу у параметр добробуту пов'язаний із застосуванням алгоритму лінійної регресії (Linear-ALLOC). На основі цього запропоновано використання техніки лінійної регресії компонента ідентифікатора ресурсу. Аналізуючи блок-схеми алгоритму оркестрації контейнерів, виявлено основні особливості контейнерної організації у хмарній федерації. З урахуванням доданих нових компонентів у хмарну федерацію проведено моделювання розподілу навантаження методами машинного навчання.

Розглянуто основні риси процесів кластеризації та оркестрації контейнерів у граничній хмарі. Розгортання розподілених програм через контейнери підтримується за допомогою віртуального масштабованого сервісного вузла (кластера) з високою внутрішньою складністю (підтримкою масштабування, балансування навантаження, відновлення після відмови) і зниженою зовнішньою складністю. У зв'язку з цим було проведено моделювання у кластерній інфраструктурі граничної хмари з використанням платформи проекту Edge Run. Для ілюстрації додаток розгортається на основі функцій OpenFaaS, яка обслуговує нейронну мережу Mobilenet з використанням TFLite в режимі CPU. Для моделювання вибрано три пристрої з лінійки продуктів Nvidia Jetson: NX, TX2 та Nano. Отримано показники пристроїв Nvidia Jetson на мережі Mobilenet. В той час, як NX і TX2 мають порівнянну продуктивність, у вузла Nano найвища RTT. Таким чином, Nano працює гірше, а викиди сягають 6 секунд. Значення 1000 CPU відповідає одному повністю завантаженому ядру CPU. Nano знову працює гірше та має найвище використання CPU. Цікаво, що пристрої NX і TX2 мають найвище

використання пам'яті, у той час як Nano найнижче. Таким чином, Nano як найвибагливіший до ресурсу пристрій з лінійки Nvidia Jetson демонструє найкращі тестові показники. Що призводить до виводу про ефективність більш дрібнішого (за вимогами та параметрами) розбиття пристроїв на периферії.

На основі цього запропоновано шляхи покращення ефективності механізмів організації контейнерів у хмарній федерації та граничній хмарі. В обох випадках на перший план виходить методика моделювання процесів у віртуальному оточенні.

Ми припускаємо, що для хмарної федерації актуальним для майбутніх досліджень є збільшення кількості кластерів, які складаються з додатків з масштабуванням. Іншими словами, кластери ростуть вшир, а їх зміст зростає у напрямку зростання функціональності. Для цього до архітектури треба додавати нові компоненти, пов'язані з такого роду кластерними рішеннями. У будь-якому випадку треба набирати статистику, яка пов'язана зі збільшенням кількості вимог користувачів. Для граничних обчислень актуальним для майбутніх експериментів є збільшення кількості пристроїв, навіть на шкоду їх якісним показникам. При цьому можна проводити експерименти на невибагливих віртуальних машинах. А їх організація пов'язана з роєм, коли кожен окремий член хмари дає малий внесок у величезний результат.

ПЕРЕЛІК ПОСИЛАНЬ

1. Jamsa, Kris. *Cloud computing*. Jones & Bartlett Learning, 2022.
2. V. Parida, Digitalization, 2018, p. 23–38, Editors Johan Frishammar Åsa Ericson URL: https://urn.kb.se/resolve?urn=urn%3Anbn%3Ase%3Altu%3Adiva-68008 (дата звернення: 4.10. 2024).
3. VMware Topics/ Bare Metal Hypervisor. What is a bare metal hypervisor? URL: https://www.vmware.com/topics/glossary/content/ (дата звернення: 5.10. 2024).
4. Hwang J. et al. "A component-based performance comparison of four hypervisors." *2013 IFIP/IEEE International Symposium on Integrated Network Management (IM 2013)*.IEEE, 2013.
5. Hyper-v architecture. (Accessed: 5 May 2022). URL: https://docs.microsoft.com/en-us/virtualization/hyper-v-on-windows/reference/hyper-v-architecture (дата звернення: 8.10. 2024).
6. Fayyad-Kazan, Hasan, Luc Perneel, and Martin Timmerman. "Benchmarking the performance of Microsoft Hyper-V server, VMWare ESXi and Xen hypervisors." *Journal of Emerging Trends in Computing and Information Sciences* 4.12 (2013): 922-933.
7. Jiang, Congfeng, et al. "Energy efficiency comparison of hypervisors." Sustainable Computing: Informatics and Systems 22 (2019): 311-321.
8. Scheepers, Mathijs Jeroen. "Virtualization and containerization of application infrastructure: A comparison." *21st twente student conference on IT*. Vol. 21. 2014.
9. Asvija, B., Rajagopal Eswari, and M. B. Bijoy. "Security in hardware assisted virtualization for cloud computing—State of the art issues and challenges." *Computer Networks* 151 (2019): 68-92.
10. Hardware-assisted virtualization By Stephen Y. Bigelov, Senior Tenology Editor URL:https://www.techtarget.com/searchitoperations/definition/hardware-assisted-virtualization. (дата звернення: 10.10. 2024).

11. Open source container-based virtualization for Linux. URL: https://openvz.org/
12. Solaris Containers/ From Wikipedia, the free encyclopedia. URL: http://en.wikipedia.org/wiki/Solaris_Containers (дата звернення: 12.10. 2024).
13. What's LXC? URL: https://linuxcontainers.org/lxc/introduction/ (дата звернення: 25.10. 2024).
14. Nickoloff, Jeffrey, and Stephen Kuenzli. *Docker in action*. Simon and Schuster, 2019.
15. Docker: Accelerated Container Application Development. URL: https://www.docker.com (дата звернення: 10.10. 2024).
16. Casalicchio, Emiliano. "Container orchestration: A survey." *Systems Modeling: Methodologies and Tools* (2019): 221-235.
17. Al Jawarneh, Isam Mashhour, et al. "Container orchestration engines: A thorough functional and performance comparison." *ICC 2019-2019 IEEE International Conference on Communications (ICC)*. IEEE, 2019.
18. Rouse M. “What is Docker Swarm?”, TechTarget, August 2016. URL: http://searchitoperations.techtarget.com/definition/Docker-Swarm. (дата звернення: 1.11. 2024).
19. Ellingwood J. “An Introduction to Kubernetes”, Digital Ocean, 14 October 2016. URL: https://www.digitalocean.com/community/tutorials/anintroduction-to-kubernetes. (дата звернення: 7.10. 2024).
20. Tomarchio O., Calcaterra D., Modica G. D. Cloud resource orchestration in the multi-cloud landscape: a systematic review of existing frameworks //Journal of Cloud Computing. – 2020. – Т. 9. – №. 1. – С. 49.
21. Gebrealif Y. et al. Architecture for orchestrating containers in cloud federations //International Conference on the Economics of Grids, Clouds, Systems, and Services. – Cham : Springer International Publishing, 2021. – С. 66-75.
22. Lee, C.A., Bohn, R.B., Michel, M.: The NIST Cloud Federation Reference Architecture(2020).URL:https://nvlpubs.nist.gov/nistpubs/SpecialPublications/NIST.SP.500-332.pdf

23. Tsakalidou V. N., Mitsou P., Papakostas G. A. Machine Learning for Cloud Resources Management—An Overview //Computer Networks and Inventive Communication Technologies: Proceedings of Fifth ICCNCT 2022. – 2022. – P. 903-915.
24. Zhang J. et al. Machine Learning Based Resource Allocation of Cloud Computing in Auction //Computers, Materials & Continua. 2018. Vol. 56, No. 1. P.123-135
25. Chandra A., Weissman J., Heintz B. Decentralized edge clouds //IEEE Internet Computing. 2013. Vol. 17. No. 5. P. 70-73.
26. Khan W. Z., Ahmed E., Hakak S., Yaqoob I., & Ahmed A. Edge computing: A survey //Future Generation Computer Systems. 2019. Vol. 97. P. 219-235.
27. Alwasel K. et al. IoTSim-Osmosis: A framework for modeling and simulating IoT applications over an edge-cloud continuum //Journal of Systems Architecture. 2021. Vol. 116. P. 101956.
28. Edge Run - Edge Computing & Analysis Platform URL: https://github.com/edgerun (дата звернення: 10.10. 2024).
29. Rausch T., Lachne C., Frangoudis P. A., Raith P., & Dustda S.Synthesizing plausible infrastructure configurations for evaluating edge computing systems //3rd USENIX Workshop on Hot Topics in Edge Computing (HotEdge 20). – 2020.
30. Serverless Functions, Made Simple. OpenFaaS® makes it simple to deploy both functions and existing code to Kubernetes. URL: https://www.openfaas.com/ (дата звернення: 12.11. 2024).
31. Howard A. G. Mobilenets: Efficient convolutional neural networks for mobile vision applications //arXiv preprint arXiv:1704.04861. – 2017.
32. NVIDIA Jetson for Next-Generation Robotics URL: https://www.nvidia.com/en-gb/autonomous-machines/embedded-systems/ (дата звернення: 17.11. 2024).

ДЕМОНСТРАЦІЙНІ МАТЕРІАЛИ

МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ

ДЕРЖАВНИЙ УНІВЕРСИТЕТ ІНФОРМАЦІЙНО-КОМУНІКАЦІЙНИХ ТЕХНОЛОГІЙ

Навчально–науковий інститут інформаційних технологій

Кафедра комп'ютерних наук

Система віртуалізації і контейнеризації для хмарних сервісів

КВАЛІФІКАЦІЙНА РОБОТА

на здобуття ступеня магістр

зі спеціальності 122 Комп'ютерні науки

Здобувача групи КНДМ-61 Семененко В.О.

Керівник д.ф., доцент, Березовська Ю.В.

Київ – 2024

ВСТУП

Об'єктом дослідження є процеси та технології організації контейнерів у системах хмарних обчислень

Предмет дослідження – методи та способи створення сучасних систем оркестрації контейнерів у хмарних сервісах

Мета роботи – розроблення нових підходів щодо покращення ефективності процесів оркестрації контейнерів у системах хмарних сервісів, від хмарної федерації до граничних обчислень

Завдання дослідження. Провести порівняльний аналіз контейнерних технологій, інструментів для оркестрації контейнерів. Дослідити архітектуру для ідентифікації ресурсів та управління оркеструванням контейнерів у хмарних федераціях. Провести моделювання та обробку даних по розподілу навантаження в структурах хмарної федерації методами машинного навчання. Розглянути особливості процесів кластеризації та оркестрації контейнерів у граничній хмарі.

Гіпервізори - базова технологія віртуалізації

Віртуалізація - створення віртуальних уявлень пристроїв, емуляція та імітація роботи самої комп'ютерної системи

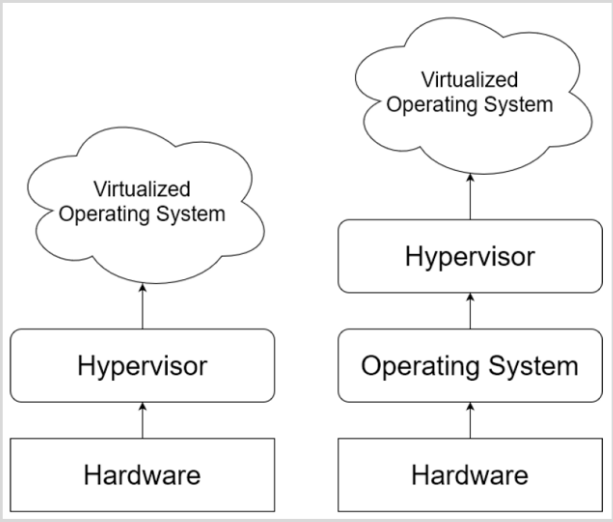


Рисунок 1.

Гіпервізор типу 1 (зліва на рис.1) розміщує свою віртуальну машину поверх апаратного забезпечення, а не над головною операційною системою, як гіпервізор типу 2. Це дає гіпервізорам типу 1 можливість використовувати обладнання більш безпосередньо та ефективніше, ніж гіпервізори типу 2, які використовують апаратне забезпечення, змодельоване операційною системою хосту.

Види віртуалізації

Повна віртуалізація, апаратна віртуалізація та паравіртуалізація

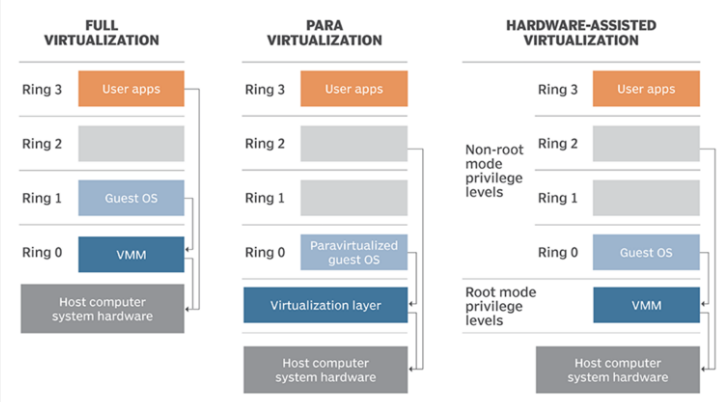


Рисунок 2 - Кільцева архітектура типів віртуалізаціїІ [10]

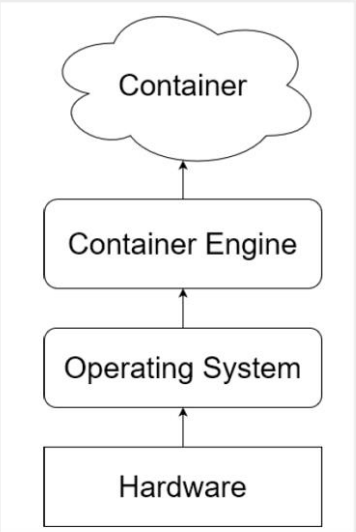
Повна віртуалізація - це метод віртуалізації, який дозволяє повністю моделювати базове обладнання VM.

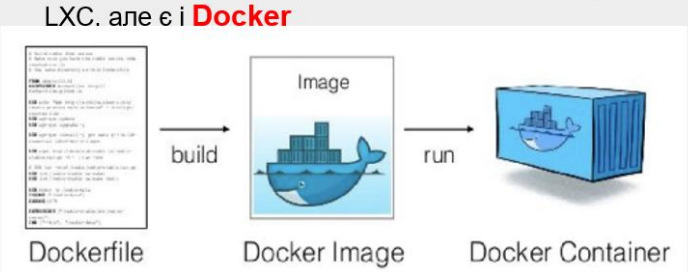
Паравіртуалізація - це тип віртуалізації, коли інструкції програмного забезпечення від гостьової операційної системи (всередині VM) використовують «гіпервиклики», які спілкуються безпосередньо з гіпервізором.

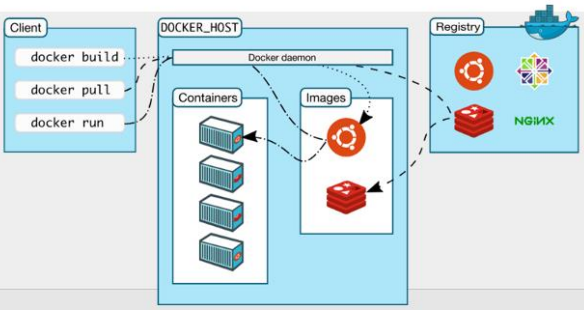
Віртуалізація з апаратним забезпеченням - це технологія, яка дозволяє зв'язуватися з набором інструкцій CPU, у якому VMM (диспетчер віртуальних машин) працює в режимі кореневого рівня нижче рівня ядра OS.

Технологія контейнеризації

Контейнеризація — це технологія, яка використовує контрольні групи та простори імен для створення ізольованих легких середовищ для запуску певних програм із лише необхідними бібліотеками та залежностями.Є OpenVZ, Solaris Zones,і контейнери Linux LXC. але є і **Docker**







Docker Swarm,
Kubernetes,
Mesos та інші

Оркестрація контейнерів

Вищий рівень контейнеризації - оркестровка контейнерів.
Керування контейнерною інфраструктурою.

Рівень керування ресурсами (**Resource Managemen**): пам'ять, CPU/GPU, дисковий простір…Рівень планування (**Sheduling**) спрямовано на ефективне використання ресурсів кластера. Рівень управління службами (**Service**) надає функціональні можливості для створення та розгортання додатків

Архітектура хмарної федерації

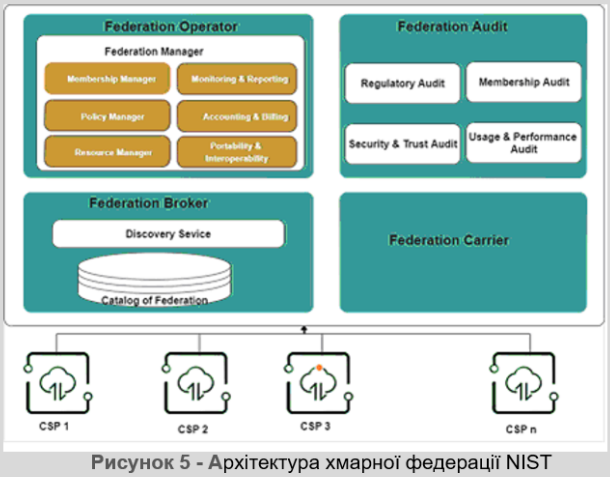


Рисунок 5 - Архітектура хмарної федерації NIST

Національний інститут стандартів і технологій (NIST) визначив чотири компоненти в еталонній архітектурі хмарної федерації: брокер федерації (Federation Broker), оператор федерації (Federation Operator), аудитор федерації (Federation Audit) і перевізник федерації (Federation Carrier).

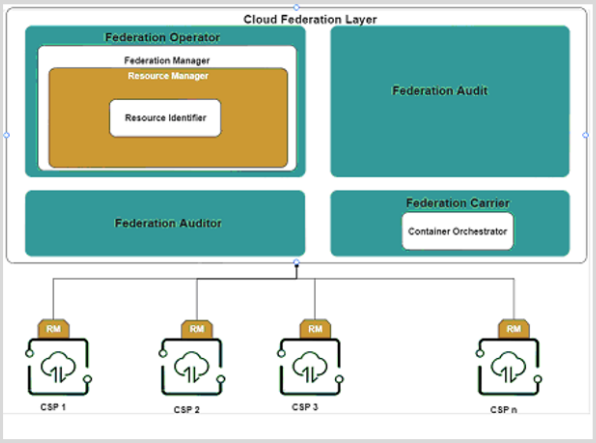


Рисунок 6 - Нові компоненти:

1. ідентифікатор ресурсів (RI, Resource Identifier);
2. оркестратор контейнерів.

Ідентифікатор ресурсів

Інформація про ресурси

Ємність (CPU, пам'ять і сховище)

$C = (c_1, c_2, ..., c_n)$

Вартість одиниці кожного ресурсу

$P = (p_1, p_2, ..., p_n)$

Інформація користувачів

m користувачів подають вимоги

$\mathbf{k}^i = (k_1^i, k_2^i, ..., k_n^i)$

ціновий поріг t^i

Загальна інформація

$\mathbf{R}^i = (\mathbf{k}^i, t^i)$

4 ядра CPU,16 ГБ RAM,150 ГБ

$\mathbf{R}^1 = (4,16,150,9)$ 9 -ціна

Загальний добробут

$$V = \max\left(\sum_{i \in U} (t^i - h_\beta(\mathbf{k}_i)) \mathbb{x}^i \right) \qquad x^i = \{0,1\}$$

суспільний добробут дає (оптимізація) $x^i = \{0,1\}$

Функція гіпотези

$$h_\beta(k^i) = \beta_0 + \beta_1 k_1^i + \beta_2 k_2^i + ... + \beta_n k_n^i$$

$\beta_0, \beta_1, ..., \beta_n$ ціна

Функція витрат

$$J(\theta) = \frac{1}{2m} \sum_{i=1}^{m} x^i \left(t^i - h_\theta(k^i) \right)^2$$

Моделювання

два види алгоритмів машинного навчання для підгонки оптимального рішення розподілу. Це алгоритм прогнозування розподілу ресурсів на основі лінійної регресії (Linear-ALLOC) і алгоритм прогнозування розподілу ресурсів на основі логістичної регресії (Logistic-ALLOC).

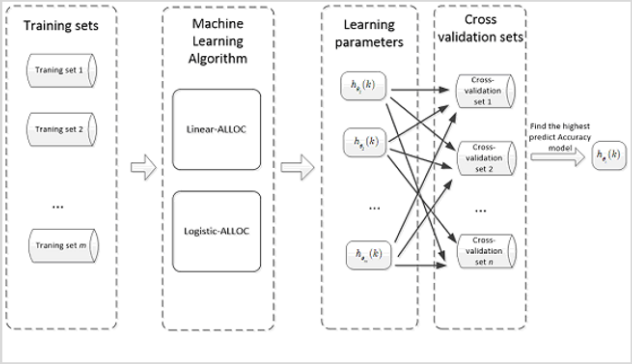


Рисунок 8 - Алгоритм машинного навчання (RI)

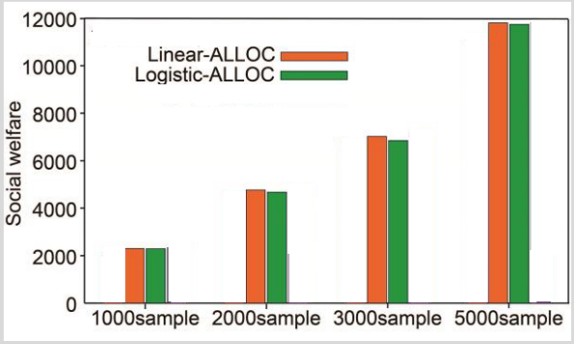


Рисунок 9 – Порівняння суспільного добробуту

Оркестратор контейнерів у хмарній федерації

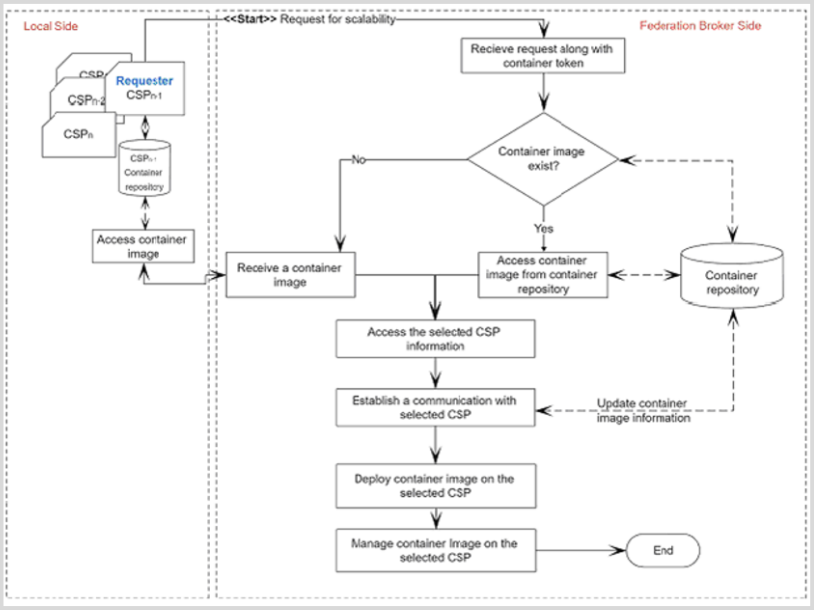
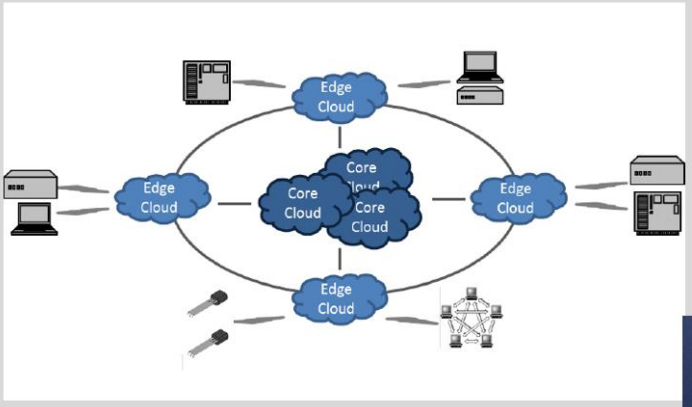


Рисунок 10 Блок-схема алгоритму оркестратора

Оркестратор контейнера ініціалізується запитом від CSP разом із маркером контейнера. Маркер використовується для унікальної ідентифікації кожного образу контейнера. Після отримання запиту цей компонент перевіряє, чи існує образ в сховищі. Якщо так, оркестратор контейнера отримує копію образу з сховища контейнерів. Якщо ні, він отримує доступ до образу контейнера шляхом запиту до CSP. Після отримання образу контейнера від CSP або пов'язаного з CSP репозиторію, який вибрано ідентифікатором ресурсу для розгортання контейнера

Периферійні хмари

Мета полягає в тому, щоб дозволити службам аналітики та генерації знань розміститись там, де джерело



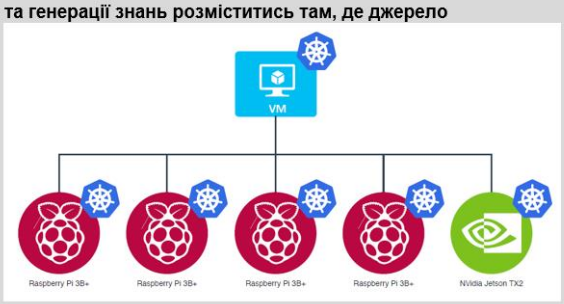
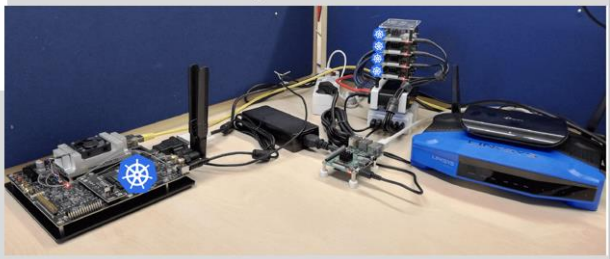


Рисунок 11 Стенд



Відтворювані експерименти для кластерів периферійних хмар допомагають оцінити методи керування кластерами та включають наступні етапи: (1) налаштування випробувального стенда, (2) створення шаблонів робочого навантаження, (3) керування розгортанням додатків, (4) оркестрування експериментів, (5) прилади та моніторинг, (6) аналіз даних експерименту

Моделювання

Використовувались образи пристроїв Nvidia Jetson. Для обробки Galileo проекту EdgeRun

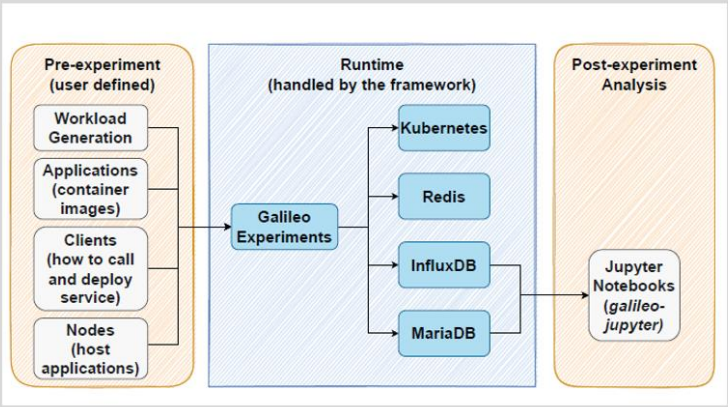


Рисунок 12 - Робочий процес експерименту

Galileo Experiments містить код, який викликає Galileo для запуску та запису експериментів.

Galileo Experiments Extensions надає клієнтські реалізації наших функцій

Galileo Experiments Functions містить функції на основі OpenFaaS

Galileo Jupyter містить шлюзи, які дозволяють користувачам легко отримувати доступ до даних, записаних під час експериментів.

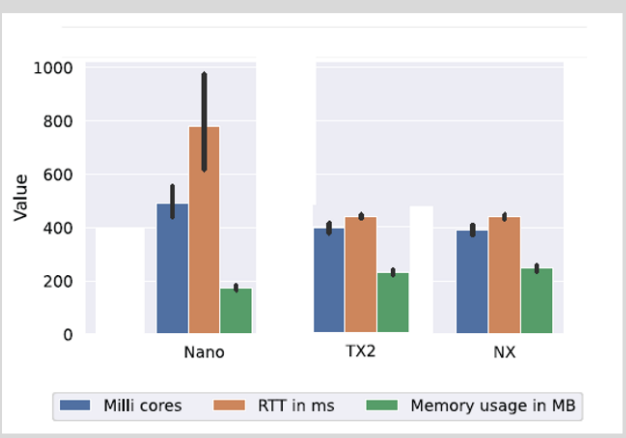
Проект galileo-jupyter внутрішньо використовує galileo-db для доступу до сховищ даних..

Telemd використовується як легкий агент детального моніторингу на основі push-розсилки

Фреймворк автоматично розгортає адаптер Telemd Kubernetes, який спостерігає за Kubernetes Pods за допомогою офіційного клієнта Go Kubernetes.

Результати моделювання

Для ілюстрації додаток розгортається на основі функцій OpenFaaS, яка обслуговує нейронну мережу Mobilenet з використанням TFLite в режимі CPU. Проект EdgeRun



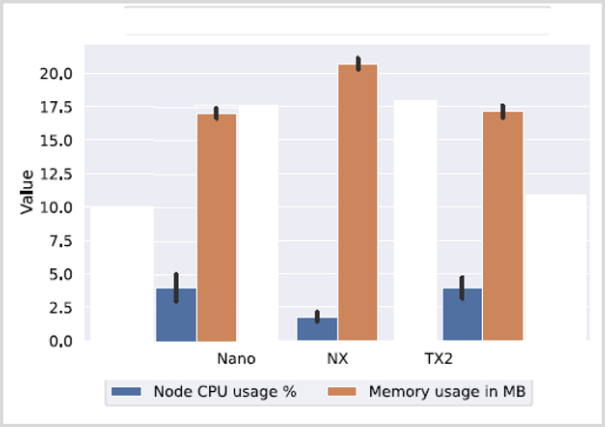


Рисунок 13 Показники пристроїв Nvidia Jetson на мережі Mobilenet

Висновки

1. У роботі проведено аналіз шляхів підвищення ефективності побудови сучасних алгоритмів оркестрації контейнерів у системах хмарних сервісів.
2. Наведено огляд різних напрямків досліджень за контейнерною технологією, які актуальні для хмарних обчислень. Проведено порівняльний аналіз контейнерних технологій, показано переваги Docker як основної платформи для розробки, доставки та експлуатації додатків. Проведено порівняння інструментів для оркестрації контейнерів.
3. Проаналізовано та досліджено архітектуру для ідентифікації ресурсів та управління оркеструванням контейнерів у хмарних федераціях. Запропоновано два нових підкомпоненти в еталонну архітектуру NIST. Запропоновано використання техніки лінійної регресії компонента ідентифікатора ресурсу. На основі аналізу блок-схеми алгоритму оркестрації контейнерів виявлено основні особливості контейнерної організації у хмарній федерації.
4. Розглянуто основні риси процесів кластеризації та оркестрації контейнерів у граничній хмарі. У зв'язку з цим проведено моделювання у кластерній інфраструктурі граничної хмари з використанням платформи проекту Edge Run. На основі цього запропоновано шляхи покращення ефективності механізмів організації контейнерів у хмарній федерації та граничній хмарі.