Міністерство освіти і науки України
Харківський національний університет радіоелектроніки

Факультет *Інфокомунікацій*
(повна назва)
Кафедра *Інформаційно-мережної інженерії*
(повна назва)

КВАЛІФІКАЦІЙНА РОБОТА
Пояснювальна записка

рівень вищої освіти другий (магістерський)

Аналіз принципів управління контейнеризованими сервісами та додатками на базі системи Kubernetes
(тема)

Виконав:

студент *2* курсу, групи *ІМІм-22-3*

Приходько О.С.
(прізвище, ініціали)

Спеціальність *172 «Телекомунікації та радіотехніка»*
(код і повна назва спеціальності)

Тип програми *освітньо-наукова*
(освітньо-професійна або освітньо-наукова)

Освітня програма

«Інформаційно-мережна інженерія»
(повна назва освітньої програми)

Керівник *доц. Колтун Ю.М.*
(посада, прізвище, ініціали)

Допускається до захисту

Зав. кафедри _____ *Безрук В.М.*
(підпис) (прізвище, ініціали)

2024 р.

Не містить відомостей заборонених до відкритого публікування.

Студент / *Приходько О.С.* /

Керівник / *Колтун Ю.М.* /

Харківський національний університет радіоелектроніки

Факультет *Інфокомунікацій*
Кафедра *Інформаційно-мережної інженерії*
(повна назва)
Рівень вищої освіти другий (магістерський)
Спеціальність *172 «Телекомунікації та радіотехніка»*
(код і повна назва)
Тип програми *освітньо-наукова*
(освітньо-професійна або освітньо-наукова)
Освітня програма *«Інформаційно-мережна інженерія»*
(повна назва)

ЗАТВЕРДЖУЮ:
Зав. кафедри _____
(підпис)
«*18*» *березня* 2024 р.

ЗАВДАННЯ
НА КВАЛІФІКАЦІЙНУ РОБОТУ

студентові *Приходьку Олексію Сергійовичу*
(прізвище, ім'я, по батькові)
1. Тема роботи *Аналіз принципів управління контейнеризованими сервісами та додатками на базі системи Kubernetes*
затверджена наказом університету від «*18*» *березня* 2024 р. № 232 *Ст*
2. Термін подання студентом роботи до екзаменаційної комісії *17 червня 2024 р.*
3. Вихідні дані до роботи *Продукти, що рекомендуються для проведення досліджень: Kubernetes - ПЗ для автоматизації розгортання та управління контейнеризованими додатками; Docker – контейнеризатор додатків; Ansible – інструмент автоматизації; YAML – мова розмітки для створення конфігураційних файлів; та інші подібні. Тип додатків, що розгортаються ASP.NET, ADO.NET, додатки, що застосовуються для реалізації практик DevOps. Інструментарій Kubernetes: minikube, calico, kubeadm та ін. Розглянути архітектурні принципи організації Kubernetes та особливості розгортання та налаштування його кластера. Проаналізувати основні типи ресурсів та шаблонів, що забезпечують процеси управління в Kubernetes.*

4. Перелік питань, що потрібно опрацювати в роботі
Вступ
1. Загальні архітектурні принципи реалізації та особливості застосування системи Kubernetes для управління контейнеризованими сервісами та додатками.
2. Аналіз функціональних особливостей основних типів контролерів, як ресурсів що забезпечують процеси управління у системі Kubernetes.
3. Застосування патернів Kubernetes для управління взаємодією контейнерів у подах.
4. Реалізація стратегій розгортання мікросервісів в системі Kubernetes.
Висновки

5. Перелік графічного матеріалу із зазначенням креслеників, схем, плакатів, комп'ютерних ілюстрацій (слайдів) *Слайди у форматі Power Point (назва, мета і актуальність кваліфікаційної роботи, підходи щодо управління сервісною інфраструктурою, які застосовувалися до появи системи Kubernetes, архітектура системи Kubernetes, підготовчі роботи, щодо розгортання кластера Kubernetes, створення кластеру кластера Kubernetes, контролер реплікації та набір реплик, контролер DaemonSet, контролери Job і CronJob, контролер Deployment, патерни Kubernetes, стратегія Rolling update – поступовий деплоймент, стратегія Recreate (повторне створення стратегія Blue/Green – синьо-зелений деплоймент, висновки)*

КАЛЕНДАРНИЙ ПЛАН

№	Назва етапів роботи	Терміни виконання етапів роботи	Примітка
1	*Ознайомлення із завданням. Уточнення ТЗ.*	18.03 – 23.03.24	*виконано*
2	*Підбір літератури за темою роботи.*	24.03 – 07.04.24	*виконано*
3	*Виконання розділу 1*	08.04 – 21.04.24	*виконано*
4	*Виконання розділу 2*	22.04 – 06.05.24	*виконано*
5	*Виконання розділу 3*	07.05 – 16.05.24	*виконано*
6	*Виконання розділу 4*	17.05 – 31.05.24	*виконано*
7	*Оформлення пояснювальної записки*	01.06 – 05.06.24	*виконано*
8	*Оформлення презентаційного матеріалу та подання роботи до ЕК*	07.01 – 14.06.24	*виконано*
9	*Підготовка до захисту та захист у ЕК*	15.06 – 30.06.24	*виконано*

Дата видачі завдання *18 беоезня 2024 р.*

Студент _____
(підпис)

Керівник роботи _____ *(доц. Колтун Ю.М.)*
(підпис) (посада, прізвище, ініціали)

РЕФЕРАТ

Пояснювальна записка: 72 с., 21 рис., 1 табл., 25 джерел, 2 додатки.

KUBERNETES, УПРАВЛІННЯ, ОРКЕСТРАЦІЯ, КОНТЕЙНЕР, ПОД, ПАТЕРН, КОНТРОЛЕР, КЛАСТЕР, КОНТРОЛЕР РЕПЛІКАЦІЇ, НАБІР РЕПЛІК, REPLICASET, DAEMONSET, JOB, CRONJOB, DEPLOYMENT, SIDECAR, ADAPTER, AMBASSADOR, МІКРОСЕРВІСИ, СТРАТЕГІЇ РОЗГОРТАННЯ, ПОСТУПОВИЙ ДЕПЛОЙМЕНТ, ROLLING UPDATE, ПОВТОРНЕ СТВОРЕННЯ, RECREATE, СИНЬО-ЗЕЛЕНИЙ ДЕПЛОЙМЕНТ, BLUE/GREEN

Об'єкт дослідження – система Kubernetes.

Мета роботи – аналіз і дослідження можливостей застосування системи Kubernetes для автоматизації розгортання та управління (оркестрації) сервісів і додатків з використанням контейнерів.

Розглянуті архітектура та компоненти Kubernetes, а також особливості розгортання та налаштування кластера Kubernetes для здійснення процесів управління. Проведений аналіз функціональних особливостей основних типів контролерів, як ресурсів, що забезпечують процеси управління у системі Kubernetes. Досліджені різні варіанти розгортання додатків, патерни для їх правильного запуску. Запропоновані різні варіанти реалізації стратегій розгортання мікросервісів в системі Kubernetes.

THE ABSTRACT

Explanatory note 72 pages, 21 fig., 1 tab., 25 sources, 2 app.

KUBERNETES, CONTROL, ORCHESTRATION, CONTAINER, POD, PATTERN, CONTROLLER, API-SERVER, CLUSTER, REPLICATION CONTROLLER, REPLICASET, DAEMONSET, JOB, CRONJOB, DEPLOYMENT, SIDECAR, ADAPTER, AMBASSADOR, MICROSERVICES, DEPLOYMENT STRATEGIES, DEPLOYMENT ROLLING UPDATE, RECREATE, DEPLOYMENT BLUE/GREEN

Object of research – Kubernetes system.

The purpose of work – analysis and research of the possibilities of using the Kubernetes system to automate the deployment and management (orchestration) of services and applications using containers.

The architecture and components of Kubernetes, as well as the features of deploying and configuring a Kubernetes cluster for management processes are considered. The functional features of the main types a controllers as resources that provide management processes in the Kubernetes system are analyzed. Various options for deploying applications and patterns for their proper launch are investigated. Various implementations of microservice deployment strategies in the Kubernetes system are proposed.

ЗМІСТ

ПЕРЕЛІК СКОРОЧЕНЬ

API (Application Programming Interface) – прикладний програмний інтерфейс;

CI/CD (Continuous Integration / Continuous Delivery (Deployment)) – безперервна інтеграція, доставка та розгортання;

CNCF (Cloud Native Computing Foundation) – фонд території хмарних обчислень;

CMT (Configuration Management Tools) – інструменти управління конфігурацією;

CP (Control Plane) – площина управління;

CRI (Container Runtime Interface) – інтерфейс середовища виконання контейнерів;

EC (Endpoints Controller) – контролер кінцевих точок;

NC (Node Controller) – контролер вузла;

OCI (Open Container Initiative) – відкрита контейнерна ініціатива;

RC (Replication Controller, або ReplicaSet) – контролер реплікації або набір реплік;

RC (Route Controller) – контролер маршрутів;

SC (Service Controller) – контролер сервісів;

SSH (Secure Shell – «безпечна оболонка») – мережний протокол прикладного рівня, що дає змогу здійснювати віддалене управління операційною системою та тунелювання TCP-з'єднань (наприклад, для передачі файлів);

SSH-ключ – безпечний спосіб з'єднання із сервером. Підключення по SSH за допомогою ключа виключає ризик, що пов'язаний з підбором і зломом вашого пароля. Для автентифікації використовуються два ключі: приватний і публічний;

VC (Volume Controller) – контролер тому;

VDS (Virtual Dedicated Server) – віртуальний виділений сервер;

VM (Virtual Machine) – віртуальна машина;

ОЗП– оперативний запам’ятовуючий пристрій;

ОС – операційна система;

ЦП – центральний процесор;

ВСТУП

З появою мережі Internet перед усім у бізнесу з'явилася можливість використовувати онлайн-технології для надання інформаційних послуг і сервісів, а також почала зростати кількість різноманітних web-додатків, що дають змогу значно підвищити ефективність його роботи. Щоб максимально повно задіяти можливості сервісів і web-додатків, IT-фахівці того часу запускали їх безпосередньо з сервера. Однак, по мірі того, як збільшувалася кількість користувачів, у бізнес-компаній з'являлися вимоги до масштабованості додатків, вартості обслуговування серверного апаратного забезпечення і надійності [1].

Як варіант розв'язання проблеми було створена технологія віртуалізації, яка надавала можливості розділити ресурси сервера на частини, що функціонують незалежно, або віртуальні машини (Virtual Machine, VM). Кожна VM може мати свою операційну систему (ОС), використовує свою ділянку оперативної пам'яті і віртуальне сховище з власними встановленими додатками. VM ізольовані одна від одної, що дає змогу одночасно запустити кілька «віртуалок» на одному фізичному сервері (хості), водночас не важливо, яке апаратне забезпечення хосту використовується для запуску додатку. Важливо тільки, щоб збігався набір інструкцій процесора і вистачало апаратних ресурсів (оперативної пам'яті, об'єму жорсткого диска тощо), що зі свого боку свідчить про те, що віртуальною машиною зручно здійснювати управління [1].

Зазначимо, що звідси випливає поняття хостингу, під яким мається на увазі послуга з надання ресурсів для розміщення інформації на сервері, який постійно має доступ до мережі. Компанії-хостери почали надавати послугу віртуального виділеного сервера (Virtual Dedicated Server, VDS). Фактично ця послуга реалізувала платформу з надання клієнтам ресурсів виділеного сервера без необхідності придбання фізичного обладнання. Особливостями такого віртуального сервера є простота його перенесення з одного хоста на інший, а також він забезпечує високу продуктивність, безпеку та гнучкість, даючи змогу клієнтам масштабувати свої ресурси щоразу, коли це буде необхідно. Так, компанія-хостер може мати велику кількість серверів, а значить може надавати послугу VDS іншим компаніям відносно дешево (у порівнянні з витратами на розгортання фізичного сервера) [1, 2].

Застосування VM зіграло важливу роль у розвитку хмарних обчислень, які сьогодні є ключовими платформами створення та надання послуг, сервісів і додатків, що визначають сучасну інформаційну інфраструктуру як окремої бізнес-компанії або підприємства, так і суспільства в цілому. Хмарні обчислення трактуються як моделі забезпечення повсюдного мережного доступу на вимогу користувача до спільного пулу обчислювальних ресурсів (мереж передачі даних, серверів, пристроїв і систем зберігання даних, додатків, сервісів), які можуть бути оперативно надані з мінімумом фінансових витрат. У цьому аспекті можна виділити безліч переваг, які надають ці технології, такі як гнучкість, масштабованість, можливість оплати за фактично використані ресурси, високу надійність і відмовостійкість [3].

Однак використання віртуальних машин має і свої проблеми. Зокрема в кожній VM запускається ядро ОС (наприклад, Windows або Linux). Для окремого додатка запускати своє ядро ОС, з точки зору необхідних для цього ресурсів, досить надмірно. Цю проблему вдалося вирішити за допомогою механізму використання контейнерів. Сутність контейнера така ж сама, як і у VM, тобто запуск додатків здійснюється ізольовано один від одного, але під час застосування контейнерів не треба запускати ядро ОС у кожному з них. Ізоляція контейнерів забезпечується вбудованими інструментами вже наявної тієї чи іншої ОС, тому, коли не потрібні спеціальні можливості VM (наприклад, запуск Windows на Linux), замість них використовують саме контейнери [1].

Найпопулярніше рішення для роботи з контейнерами – це Docker, так званий контейнеризатор додатків, який дає змогу запакувати їх з усім оточенням і залежностями. Контейнери в Docker створюються на основі образів, які по суті є шаблонами. Образ – це набори файлів додатку та інсталяційних команд. Щоб запустити свій застосунок через Docker, формується Dockerfile, у якому міститься покроковий опис встановлення на комп'ютер: на основі цієї інформації створюється образ з додатком. Docker запускається як системна служба на сервері під управлінням ОС Windows або Linux. Ця служба відповідає за запуск контейнера, його зупинку та інші дії щодо управління [1].

У деякому сенсі контейнер можна розглядати як своєрідне ізольоване середовище, у якому можна запускати додатки з обмеженими правами доступу до ресурсів системи. У контейнері міститься ізольоване оточення – файлова система, бібліотеки, залежності і навіть ОС (як, наприклад, у випадку контейнера Docker). При цьому контейнери задіюють загальне ядро ОС хоста, тому вони є більш

легкими порівняно з VM, де кожна має власне ядро. Контейнери також дають змогу упакувати додаток і його компоненти (код, бібліотеки, залежності, налаштування, тощо) в один самодостатній та ізольований від довкілля контейнерний образ, що забезпечує цілісність даних і переносимість між різними середовищами, адже всі необхідні компоненти перебувають усередині контейнера і не залежать від зовнішніх факторів [1].

Таким чином, контейнери являють собою компактну і переносиму одиницю розгортання сервісів і додатків, але при цьому потребують ефективних рішень задач, що пов'язані з їх автоматизацією, управлінням, маршрутизацією, забезпеченням можливостей з виявлення та усунення проблем, і т.ін. Так для забезпечення гнучкої та ефективної взаємодії і управління інфраструктурними засобами, що пропонуються хмарними платформами, використовується методології взаємодії на основі концепції DevOps. Вищезгадана програмна платформа Docker також надає інструменти для автоматизації розгортання та управління додатками в середовищах із підтримкою контейнеризації. Але в комплексних додатках часто використовується безліч контейнерів, які якось взаємодіють між собою. Кожен сервіс і додаток висуває свої унікальні вимоги до ресурсів і масштабування, а також потребує індивідуального моніторингу, тобто потрібно організувати загальну єдину систему управління інфраструктурою. Зазначимо, що поняття управління інфраструктурою, яка складається з безлічі контейнерів, називається оркестрацією [1, 4].

Серед рішень, створених для оркестрації сервісів і додатків, можна виділити систему Kubernetes, що являє собою відкрите ПЗ для автоматизації розгортання, масштабування та управління контейнеризованими додатками. Проект із відкритим вихідним кодом розміщено на серверах так званого Фонду території хмарних обчислень (Cloud Native Computing Foundation, CNCF), який за своєю суттю є програмним підходом до створення, розгортання та управління сучасними додатками в середовищах хмарних обчислень. Основою системи Kubernetes є контейнерна технологія, яку складають docker-контейнери, які є базовими для створення сервісів. Тут контейнер фактично є міні ОС із необхідним функціоналом лише для виконання певної дуже простої задачі. Такі контейнери займають мінімум дискового простору, а їх запуск триває невеликий час. Іншими словами, контейнер фактично реалізує один мікросервіс. Сукупність мікросервісів формує мікросервісну платформу або, по іншому, котейнерну технологію, що

лежить в основі Kubernetes, де кожен сервіс характеризується незалежністю від інших [4, 5, 6].

Відповідно з цим метою цієї магістерської кваліфікаційної роботи є аналіз і дослідження можливостей застосування системи Kubernetes для автоматизації розгортання та управління (оркестрації) сервісів і додатків з використанням контейнерів. У процесі аналізу такої складної системи як Kubernetes треба відстежувати сервіси в контейнерах, збирати та систематизувати метрики, налаштовувати журналювання та безпеку, а також враховувати аспекти мережного управління трафіком у процесі взаємодії мікросервісів, що свідчить про актуальність досліджень та аналізу, що будуть проведені.

1 ЗАГАЛЬНІ АРХІТЕКТУРНІ ПРИНЦИПИ РЕАЛІЗАЦІЇ ТА ОСОБЛИВОСТІ ЗАСТОСУВАННЯ СИСТЕМИ KUBERNETES ДЛЯ УПРАВЛІННЯ КОНТЕЙНЕРИЗОВАНИМИ СЕРВІСАМИ ТА ДОДАТКАМИ

1.1 Роль і місце системи Kubernetes серед існуючих підходів щодо управління сервісами та додатками

Традиційно сервіси та додатки розгортали безпосередньо на фізичних серверах із вихідних кодів або інсталяційних пакетів (рис. 1.1). У цьому випадку досить складно було визначити межі ресурсів для додатків на фізичному сервері, що спричинило проблеми з розподілом ресурсів. Наприклад, якщо кілька додатків виконуються на фізичному сервері, то якийсь один додаток із високою ймовірністю займатиме більшу частину ресурсів, а інші додатки в результаті цього працюватимуть гірше. Рішенням цього могла стати можливість інсталяції та запуску кожного додатку на окремому своєму фізичному сервері. Але масштабування такої інфраструктури було малоефективним, тому що ресурси фізичного сервера використовувалися далеко не повністю, вона була громіздкою і, як наслідок, підтримка такої великої кількості фізичних серверів була досить дороговартісною, як із погляду первісного розгортання, так і подальшого технічного обслуговування та адміністрування. З появою наборів програмних інструментів управління конфігурацією (Configuration Management Tools, CMT), таких як `puppet`, `ansible`, `chef`, – процес інсталяції додатків і управління ними на одному фізичному сервері став більш автоматизованим. Системи CMT дали змогу відійти від схеми «один додаток – один сервер», що дало змогу знизити вартість інфраструктури, але все ще була потрібна участь адміністратора в її плануванні та автоматизації супутніх процесів (написанні конфігураційних файлів, скриптів прогону тощо) для виконання рутинних завдань. Головним недоліком такого підходу вважається низька продуктивність [7].

Як зазначалося у вступі, більш ефективним підходом до управління конфігурацією сервісної інфраструктури стала віртуалізація та поява віртуальних машин, які стали одним з інструментів розгортання сервісів і додатків у хмарних обчисленнях. Віртуалізація дозволила запускати кілька VM на одному фізичному сервері (рис. 1.1). Вона ізолює додатки між VM і забезпечує певний рівень

безпеки, оскільки інформація із одного додатку не може бути вільно доступною для іншого додатка, дає змогу краще використовувати ресурси на фізичному сервері та забезпечує кращу масштабованість, оскільки додаток можна легко додати чи оновити, крім цього знижуються витрати на обладнання та багато іншого. Також у вступі було зазначено, що за допомогою віртуалізації можна перетворити набір фізичних ресурсів на кластер одноразових віртуальних машин. Перевага цього підходу полягає в отриманні повністю незалежної віртуальної машини, обмеженої тільки залежностями ресурсів гіпервізора [7].

Ще більш просунутим підходом до управління конфігурацією сервісної інфраструктури стало застосування контейнерних технологій, де основними елементами були контейнери. Як зазначалося вище, вони схожі на VM, але в них є властивості ізоляції для спільного використання ОС між додатками. Тому контейнери вважаються легкими. Подібно до VM, контейнер має свою власну файлову систему, процесор, пам'ять, простір процесу і багато іншого (рис. 1.1). Оскільки контейнери не зв'язані з базовою інфраструктурою, їх можна переносити між хмарами та дистрибутивами ОС [7].

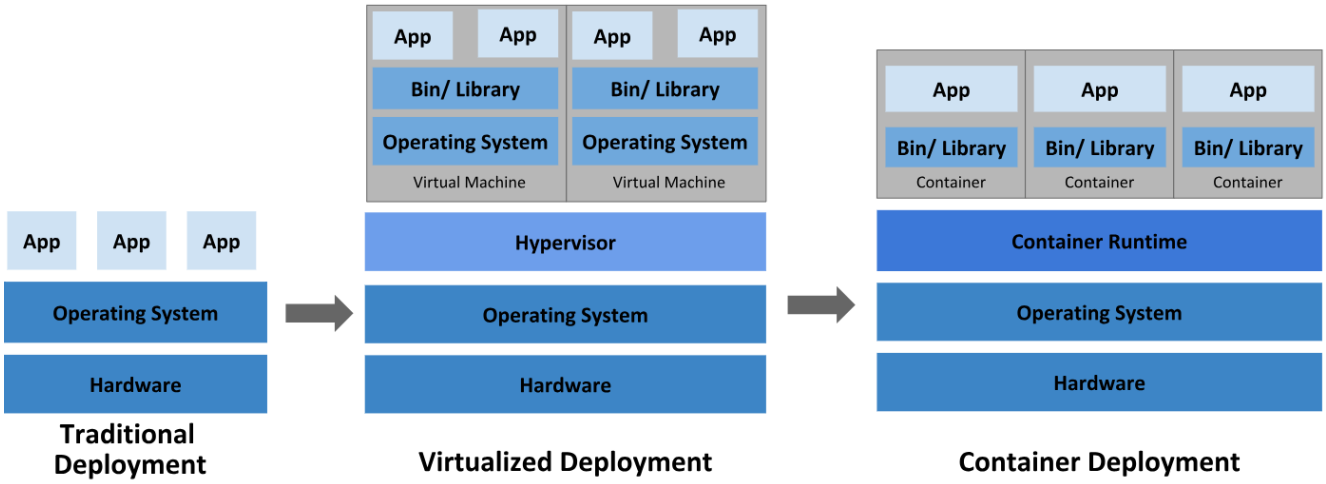

Рисунок 1.1 – Підходи щодо управління сервісною інфраструктурою, які застосовувалися до появи системи Kubernetes

На сьогоднішній день серед рішень, що були створені для реалізації більш гнучких і масштабованих підходів щодо управління сервісами та додатками на рівні контейнерів, як зазначалося вище, широкого поширення отримала система Kubernetes. Вона розроблена на основі системи управління контейнерами під

назвою Borg, що була створена розробниками компанії Google. Kubernetes написаний мовою Go, має відкритий код і надає можливості для автоматизації процесів розгортання, масштабування та управління контейнерами, їх діагностики та самокерування, а також балансування навантаження між ними, – забезпечуючи тим самим більш легке та ефективне управління складними сервісами та додатками [1, 4].

З появою мікросервісних архітектур на основі контейнерних технологій, коли контейнер додаток розбивається на невеликі сервіси, які розгортаються незалежно, Kubernetes став просто необхідним інструментом для управління цими складними системами. У той же час він не обмежується лише оркестрацією контейнерів і мікросервісів. Kubernetes здатний ефективно здійснювати управління монолітними додатками, додатками, що мають певні стани, інфраструктурними сервісами і навіть складними додатками, які працюють у гібридних і багатокластерних оточеннях. При цьому слід зазначити, що сам Kubernetes не є монолітним рішенням, тому зазначені вище можливості за замовчуванням у більшості випадків є додатковими та доступними для подальшого підключення за необхідністю [1, 7].

У Kubernetes є компоненти для організації платформи розробника, але ця система також надає можливість вибору і користувачеві, а також надає гнучкість там, де це важливо. Іншими словами, Kubernetes є, по суті, фреймворком для гнучкої роботи розподілених систем, який реалізує масштабування та обробку помилок у додатку, надає шаблони розгортання та багато іншого. Зокрема Kubernetes надає [7]:

- моніторинг сервісів і розподіл навантаження. Kubernetes може виявити контейнер, використовуючи DNS-ім'я або власну IP-адресу. Якщо трафік у контейнері високий, то Kubernetes може збалансувати навантаження і розподілити мережний трафік так, щоб розгортання було максимально стабільним;

- оркестрацію сховища. Kubernetes дає змогу автоматично створити систему зберігання за персональним вибором користувача, таку, наприклад, як локальне сховище у провайдера загальнодоступної хмари, тощо;

- автоматичне розгортання і відкати. Використовуючи Kubernetes, можна розгортати контейнери в необхідних станах і формах відповідно до необхідних умов, а потім, у процесі виникнення необхідності, змінити ці фактичні стани під нові умови. Наприклад, можна автоматизувати Kubernetes, з одного боку, на створення і розгортання нових контейнерів, а, з іншого боку, на видалення існуючих контейнерів з розподілом всіх їх ресурсів у новий контейнер;

- автоматичний розподіл навантаження. Користувач має можливість надати Kubernetes кластер вузлів, який він може використати для запуску контейнерних задач, при цьому користувач вказує Kubernetes, скільки продуктивності центрального процесора (ЦП) і пам'яті (ОЗП) потрібно кожному контейнеру. Kubernetes може розмістити контейнери на вузлах хмарної інфраструктури так, щоб найбільш ефективно використовувати ресурси;

- самоконтроль. Kubernetes може перезапускати контейнери, що вийшли з ладу, замінювати і завершувати роботу контейнерів, які не проходять визначену користувачем перевірку працездатності, а також не показує їх клієнтам, поки контейнери не будуть готові до обслуговування;

- управління конфіденційною інформацією та конфігурацією. Kubernetes може зберігати та управляти конфіденційною інформацією (паролі користувачів, SSH-ключі, що гарантують безпечний спосіб з'єднання з сервером, тощо). Також можна розгортати і оновлювати конфіденційну інформацію та конфігурацію застосунку без змін образів контейнерів і не розкриваючи конфіденційної інформації в конфігурації стека.

Також під час використання Kubernetes важливо пам'ятати, чим ця система не є, і для чого не призначена [7]:

- не обмежує типи додатків, що підтримуються. Kubernetes прагне підтримувати широкий спектр робочих навантажень, включно з тими, що мають або не мають стан, а також ті, що пов'язані з обробкою даних. Якщо додаток може працювати в контейнері, він має чудово працювати і в Kubernetes;

- не розгортає вихідний код і не збирає додаток. Так одна з практик методології DevOps, у якій робочі процеси, що реалізуються за допомогою безперервної інтеграції, доставки та розгортання (Continuous Integration / Continuous Delivery (Deployment), CI/CD), визначаються культурою, набором принципів і практик, а також технічними вимогами, що дають змогу розробникам частіше та надійніше розгортати зміни додатків завдяки інструментам автоматизації;

- не надає деякі сервіси для додатку, такі як проміжне програмне забезпечення (наприклад, черги повідомлень), платформи обробки даних (наприклад, Spark), бази даних (наприклад, MySQL), кеші або кластерні системи зберігання (наприклад, Ceph), у вигляді вбудованих сервісів. Такі компоненти можуть працювати в Kubernetes та/або можуть бути доступні для додатків, що працюють у Kubernetes, через переносні механізми, такі як Open Service Broker;

- не включає рішення для ведення журналу, моніторингу або оповіщення. Kubernetes реалізує деякі механізми для збору та експорту метрик;

- не вказує і не вимагає налаштування мови/системи (наприклад, Jsonnet). Kubernetes надає декларативний прикладний програмний інтерфейс (Application Programming Interface, API), який може бути використаний для довільних форм декларативних специфікацій;

- не надає і не приймає ніяких комплексних систем конфігурації, технічного обслуговування, управління або самовідновлення.

Таким чином, ґрунтуючись на вищевикладеному, можна дійти висновку, що Kubernetes – це не просто система управління чи оркестрації, бо фактично необхідність у цьому мінімізується. Технічне визначення традиційної оркестрації – це послідовне виконання певного циклу роботи: спочатку виконуємо процес «А», потім «В», потім «С». Використання інструментів системи Kubernetes передбачає виконання набору компонованих, але незалежних процесів управління, які безперервно підлаштовують поточний стан до очікуваного стану. Тобто немає необхідності чітко виконувати послідовність робочого циклу, у разі виконання процесів від «А» до «С». І при цьому також не потрібна наявність постійного централізованого контролю. Усе це робить систему простішою у використанні, потужнішою, надійнішою, стійкішою і більш розширюваною [7].

1.2 Архітектура та компоненти Kubernetes

Архітектуру системи Kubernetes розроблена з урахуванням складних вимог щодо здійснення управління контейнерами та мікросервісами. Її практична реалізація являє собою кластер, в основі розгортання якого закладені властивості модульності, розширюваності та горизонтальної масштабованості. Кластер Kubernetes (Cluster) складається з набору машин, – так званих вузлів, які запускають контейнеризовані додатки. Загалом, він повинен мати щонайменше один робочий вузол. У робочих вузлах розміщуються поди (Pods), які є компонентами додатка (рис. 1.2) [1, 8].

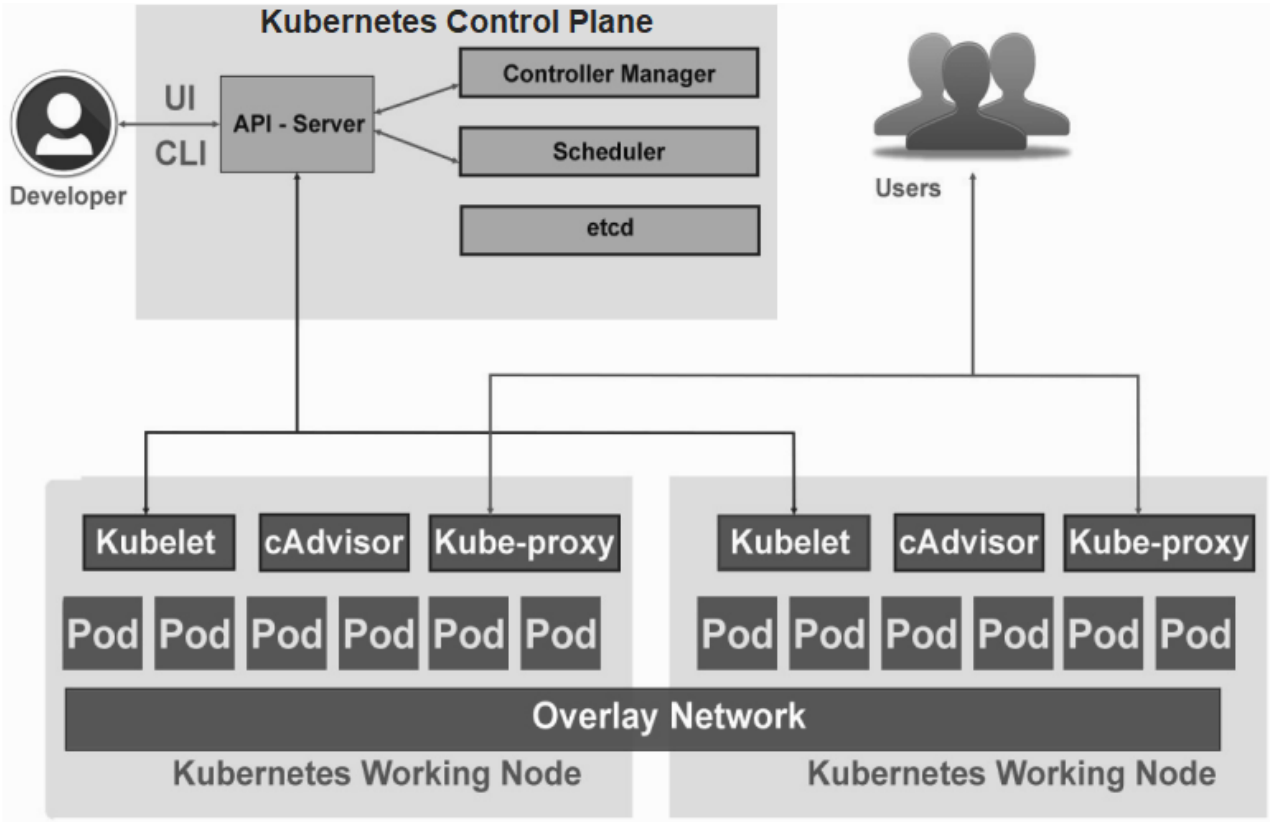

Рисунок 1.2 – Архітектура системи Kubernetes

Поди є основними структурними блоками Kubernetes. Це найменші одиниці розгортання, які являють собою додатки у кластері, що виконуються. Один под – це один екземпляр додатку в Kubernetes. Він може складатися або з одного контейнера, або з невеликої кількості контейнерів, які спільно використовують ресурси. Функціонально под інкапсулює контейнер додатку (або, в деяких випадках, кілька контейнерів), ресурси зберігання, унікальний мережний IP і параметри, які визначають, як має працювати контейнер. Крім контейнерів додатків, под може містити контейнери ініціалізації, які запускаються під час запуску пода. У вступі зазначалося, що найпоширенішим рішенням для контейнеризації в Kubernetes є Docker, але роботу з подами підтримують також і інші контейнерні системи [1, 8].

Головним контролюючим вузлом кластера є площина управління Kubernetes (Control Plane, CP), компоненти якої відповідають за прийняття рішень щодо стану кластера. На CP виконується планування і розподіл задач, а також відбувається управління ресурсами. Компоненти CP можуть бути запущені на будь-якій машині в кластері. Однак для простоти сценарії налаштувань зазвичай

запускають усі компоненти CP на одному комп'ютері та водночас не дозволяють запускати контейнери користувача на цьому комп'ютері [1, 8].

Для роботи з об'єктами Kubernetes (подами, службами, реплікаціями тощо), їх створення, змінювання або видалення необхідно забезпечити взаємодію з Kubernetes API-сервером (API-Server), який є клієнтською частиною CP. API-Server призначений для горизонтального масштабування, тобто таких серверів можна запустити кілька екземплярів і збалансувати трафік між ними. API-Server надає `RESTful`-інтерфейс, який дозволяє здійснювати взаємодію з об'єктами через HTTP-запити. Для деяких мов програмування існують офіційні клієнтські бібліотеки, які спрощують роботу з API, а також надають засоби для авторизації та автентифікації [1].

З API-Server можна реалізувати взаємодію різними способами [1]:

- за допомогою утиліти командного рядка `kubectl`, яка надає зручний спосіб взаємодії з API-Server. Використовуючи команди `kubectl` можна створювати, змінювати і видаляти об'єкти, а також отримувати інформацію про стан кластера;

- програмно, тобто можна використовувати клієнтські бібліотеки, що надаються Kubernetes, щоб взаємодіяти з API-Server через програмний код. Ці бібліотеки надають зручні методи для створення, читання, оновлення та видалення об'єктів Kubernetes;

- безпосередньо, тобто можна також взаємодіяти з Kubernetes API-Server безпосередньо, відправляючи HTTP-запити на відповідні його кінцеві точки. Таку взаємодію застосовують, коли необхідно реалізувати більш специфічну поведінку або управління.

Для зберігання всієї конфігурації, даних і стану кластера в Kubernetes використовується еталонне сховище (`etcd`) у форматі «ключ-значення», що забезпечує надійність і узгодженість даних. Слід зазначити, що в цьому випадку має обов'язково бути налаштоване резервне копіювання даних [8].

За розміщення подів (контейнерів) на фізичних або віртуальних вузлах з урахуванням безлічі факторів відповідає планувальник (Scheduler). До цих факторів, наприклад, належать [1, 8]:

- вимоги до ресурсів;

- обмеження, що пов'язані з апаратними та/або програмними політиками;

- приналежності (affinity) і неприналежності (anti-affinity) вузлів (node affinity) / подів (pod affinity);

- місцезнаходження даних;

- граничні терміни.

Зазначимо, що `affinity` дають змогу реалізувати управління розподілом подів у кластері з урахуванням вимог до ресурсів, близькості до інших подів або до інших факторів [1].

Компонент Controller Manager, що також знаходиться на площині управління Kubernetes (рис. 1.2), забезпечує запуск у кластері процесів (циклів) управління, які відстежують загальний стан кластера через API-Server і вносять відповідні зміни, які спрямовані на те, щоб привести наявний поточний стан у відповідність до бажаного стану. У Kubernetes кожен процес управління є окремим контролером. Вони забезпечують автоматичне масштабування і управління відмовостійкістю, а також забезпечують підтримку правильної кількості екземплярів додатків і сервісів відповідно до заданих параметрів. Для спрощення такі контролери скомпільовані в один двійковий файл і виконуються в одному процесі. Наприклад, у загальному процесі можуть виконуватися контролери таких типів [1, 8]:

- контролер вузла (Node Controller, NC) – повідомляє і реагує на збої вузла;

- контролер реплікації (Replication Controller, RC) або набір реплік (ReplicaSet) – підтримує бажану кількість подів (подів з однаковим образом) для кожного об'єкта контролера реплікації в системі. Якщо кількість подів знижується або збільшується, RC автоматично коригує кількість, щоб вона відповідала заданій;

- контролер кінцевих точок (Endpoints Controller, EC) – заповнює об'єкт кінцевих точок (Endpoints), тобто зв'язує сервіси та поди;

- контролери облікових записів і токенів (Account & Token Controllers) – створюють стандартні облікові записи і токени доступу API для нових просторів імен;

- контролер Deployment – відповідає за управління репліками подів. Фактично це абстракція поверх ReplicaSet, яка дає змогу користувачеві здійснювати управління оновленнями додатків. Він надає можливість плавного

оновлення версії додатку, а також плавного відкату до попередніх версій у разі виникнення проблем;

- контролер StatefulSet – призначений для управління подами з унікальними ідентифікаторами та стабільними мережними іменами;

- контролер DaemonSet – забезпечує на кожному вузлі в кластері роботу тільки одного екземпляра пода. Найчастіше його застосовують для додатків, які мають бути запущені на кожному вузлі, наприклад, мережні плагіни або моніторингові агенти;

- контролери Job і CronJob. Контролер Job відповідає за виконання певної задачі (пода) і за успішне її (його) завершення. Коли задача успішно завершена, то Job вважається виконаним. Якщо задача завершується з помилкою, Kubernetes може автоматично перезапустити її для забезпечення успішного виконання. Контролер CronJob – це планувальник, що дає змогу запускати задачі (поди) періодично, подібно до планувальника задач в ОС.

Також на площині управління Kubernetes, починаючи з версії 1.6, використовується компонент Cloud Controller Manager, який запускає контролери, що взаємодіють з основними хмарними провайдерами. По суті, ці контролери надають додаткову функціональність до набору контролерів, що реалізується на базі звичайного компонента Controller Manager. Цей компонент запускає тільки цикли контролера, які належать до хмарного провайдера. Їх можна відключити, встановивши прапор `cloud-provider` зі значенням `external` під час запуску звичайного компонента Controller Manager. За допомогою Cloud Controller Manager код, як хмарних провайдерів, так і самого Kubernetes може розроблятися незалежно один від одного [8].

Приклади контролерів, що залежать від хмарних провайдерів [8]:

- контролер NC – перевіряє хмарний провайдер, щоб визначити, чи був видалений вузол у хмарі після того, як він перестав працювати;

- контролер маршрутів (Route Controller, RC) – налаштовує маршрути в основній інфраструктурі хмари;

- контролер сервісів (Service Controller, SC) – створює, оновлює і видаляє балансувальники навантаження хмарного провайдера;

- контролер тому (Volume Controller, VC) – створює, приєднує і монтує томи, а також взаємодіє з хмарним провайдером для оркестрації томів.

Робочі машини в кластері, на яких запускаються контейнери, називаються вузлами (Nodes). Вони виконують фактичні задачі – здійснюють запуск і управління контейнерами, а також передають стан назад у площину управління Kubernetes. Кожен вузол у кластері повинен мати контейнерний движок, що визначає середовище виконання контейнерів (Container Runtime, CR). Під час запуску пода в Kubernetes, він (pod) створює контейнери і управляє ними, використовуючи CR на конкретному вузлі. Нижче наведено приклади найпоширеніших CR, які можуть використовуватися в Kubernetes [1]:

- docker – одне з найпопулярніших і найбільш використовуваних CR. Він забезпечує середовище для запуску контейнерів і управління ними;

- containerd – більш низькорівневе CR, яке було спочатку розроблене як частина проекту docker. Забезпечує базові функції для запуску та управління контейнерами;

- CRI-O – це сумісна з відкритою контейнерною ініціативою (Open Container Initiative, OCI) полегшена реалізація стандарту, що підтримує інтерфейс середовища виконання контейнерів (Container Runtime Interface, CRI). CRI-O є окремим контейнерним движком, що визначає різні CR. Він високопродуктивний і легкий, оптимізований під мінімальні вимоги у разі роботи з контейнерами [9].

CRI визначає API, який Kubernetes буде використовувати для управління різними середовищами виконання контейнерів (Container Runtime), що створюють і управляють контейнерами. CRI спрощує для Kubernetes використання різних Container Runtime. Замість того, щоб включати в Kubernetes підтримку кожної з них, використовується стандарт CRI. При цьому задача управління контейнерами повністю лягає на Container Runtime [9].

OCI – це група компаній, які підтримують специфікацію формату образу контейнера і методу запуску контейнерів. Ідея OCI полягає в тому, що можна обирати між різними CR, які відповідають цій специфікації. При цьому кожне Container Runtime може мати різні реалізації нижнього рівня. Наприклад, можна мати одне OCI-сумісне CR для хостів Linux і одне для хостів Windows [9].

Таким чином, CRI-O – це сумісна з OCI полегшена реалізація інтерфейсу CRI. За допомогою CRI-O можна здійснювати запуск Kubernetes подів і витягувати необхідні образи [10].

Також на кожному вузлі у кластері є агент `kubelet`. Він підтримує зв'язок між Kubernetes Control Plane і nodes, слідкує за запуском подів і здійснює управління ними, а також їх контейнерами, образами, розділами тощо [1, 8].

У парі з агентом `kubelet` і Container Runtime працює компонент `Kube-proxy`, який конфігурує правила мережі на вузлах. Він забезпечує взаємодію між подами та зовнішніми мережами, управляє мережним проксуванням і балансуванням навантаження. Також він дозволяє подам використовувати стабільні DNS-імена для звернення один до одного, навіть якщо їх IP-адреси змінюються. `Kube-proxy` працює на кожному вузлі в кластері, і реалізує частину концепції сервісів або служб (Services) [1, 8].

У всіх кластерів Kubernetes має бути кластерний DNS, оскільки багато прикладів передбачають його наявність. Кластерний DNS – це такий самий DNS-сервер поряд з іншими DNS-серверами в мережному оточенні, який здійснює оновлення DNS-записів для сервісів Kubernetes. Контейнери, що запущені за допомогою Kubernetes, автоматично включають цей DNS-сервер у свої групи DNS [8].

Концепція Services реалізує методи надання доступу до мережного додатка, що працює як один або кілька подів. Інакше кажучи, це спосіб зробити контейнери (поди) доступними для інших додатків. Наприклад, коли є кілька контейнерів, що працюють в одному кластері, кожен із них може мати свою IP-адресу і DNS-ім'я, яке не завжди стабільне, оскільки контейнери можуть перезапускатися або масштабуватися. Служби вирішують цю проблему, надаючи стабільні DNS-імена та адреси для групи контейнерів [1].

У разі створення служби, Kubernetes призначає їй унікальне ім'я та віртуальну IP-адресу. Цю віртуальну адресу використовуватимуть, щоб звертатися до служби, замість того щоб знати конкретні IP-адреси контейнерів. Служби також автоматично маршрутизують запити до контейнерів (навіть якщо вони змінюють свої IP-адреси або перезапускаються). Тому немає необхідності турбуватися про точні IP-адреси контейнерів, оскільки можна звертатися до служб за їх іменами [1].

1.3 Особливості розгортання та налаштування кластера Kubernetes для здійснення процесів управління

На сьогоднішній день існує багато утиліт, що застосовуються для створення кластера Kubernetes, наприклад, утиліта `Kind`, яка дає змогу розгорнути кластер у контейнері Docker. Варто зазначити, що саме за допомогою цієї утиліти розробники Kubernetes тестують нові версії цієї системи. Для локальної розробки додатків або

тестування системи Kubernetes найкращим буде інструмент minikube, який дозволяє налаштувати одновузловий кластер і на цей час є найпростішим і найшвидшим шляхом щодо створення повністю функціонуючого кластера Kubernetes. Для безпосередньо побудови кластера зазвичай використовують утиліту kubeadm, або Kubespray, яка є певною надбудовою над kubeadm, і дає змогу розгорнути кластер одразу на багатьох серверах, адже в її роботі використовують Ansible, що, у свою чергу, дає змогу автоматизувати монотонні процеси на різних машинах. У цій кваліфікаційній роботі будемо використовувати утиліту kubeadm, яка дає змогу досить швидко розгорнути готовий до використання кластер [11, 12].

Для того, щоб можна було почати процес створення кластера, потрібен сервер з інстальованою Linux-подібною операційною системою, наприклад, Ubuntu Далі по крокам покажемо особливості розгортання кластера Kubernetes [11, 12]:

1) Відключаємо розділ підкачки, використовуючи команду `swappoff –a.`

2) Додаємо ключ Google apt із репозиторію gpg командою [12]:

```
Curl –s https:/0/packages.cloud.google,com/apt/doc/
apt-key.gpg | sudo apt-key add -.
```

3) Також потрібно додати Kubernetes apt репозиторій в локальний лист репозиторієв, використовуючи команду [12]:

```
sudo bash -c 'cat <<EOF>/etc/apt/sources.list.d/
kubernetes.list
deb https://apt.kubernetes.io/ kubernetes-xenial main EOF'
```

4) Далі потрібно зробити оновлення додатків, використовуючи команду [11, 12]:

```
Sudo apt-get update.
```

Також можна побачити, які версії додатків доступні для розгортання служб kubelet і docker. Для цього використовуємо команди [11, 12]:

```
apt-cache policy kubelet | head -n 20
apt-cache policy docker.io | head -n 20
```

5) Наступним кроком розгортаємо потрібні додатки для роботи кластера [12]:

```
sudo apt-get install -y docker.io
kubelet kubeadm kubectl
```

6) Наступним кроком ці додатки потрібно позначити для того, щоб вони не могли оновлюватися як всі інші додатки командою `apt-get update` [12]:

```
sudo apt-mark hold docker.io kubelet kubeadm kubectl
```

Цей крок є важливим, тому що версії додатків мають залишатися тими, які були задані під час їх розгортання.

7) Далі треба перевірити, що служби `kubelet` і `docker` мають статус «увімкнений». Це перевірка проводиться виконанням команди [11, 12]:

```
sudo systemctl status kubelet.service
sudo systemctl status docker.service
```

Результат виконання команд наведений на рис. 1.3 та 1.4.

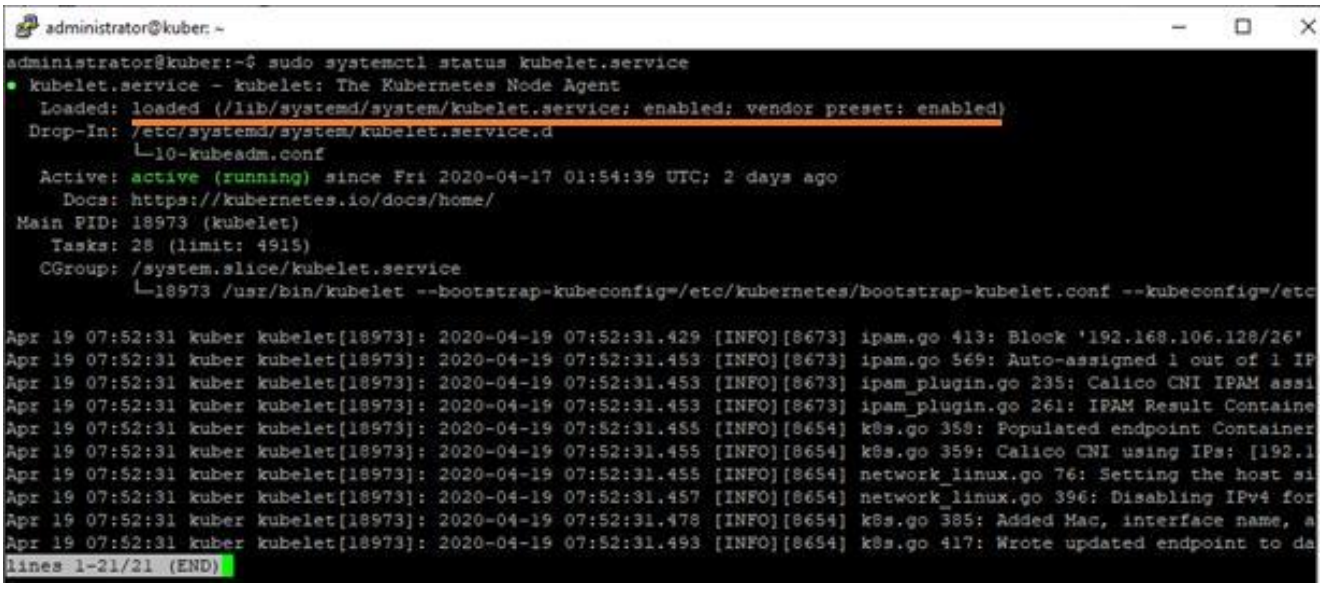

Рисунок 1.3 – Перегляд встановленого статусу служби `kubelet`

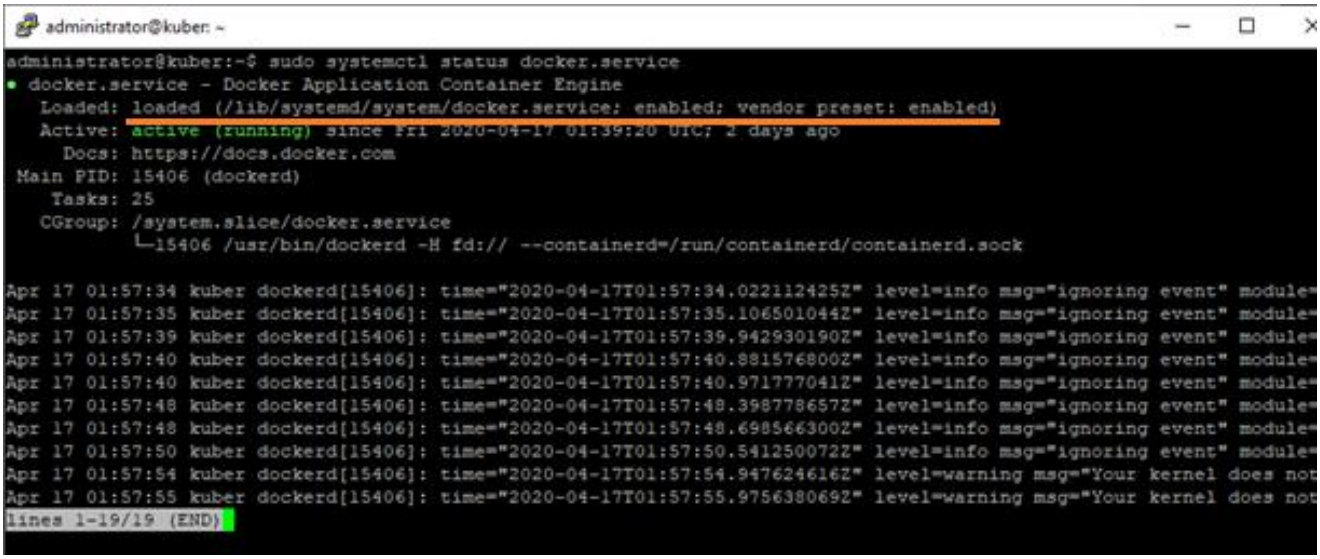

Рисунок 1.4 – Перегляд встановленого статусу служби `docker`

У тому разі якщо вказаний статус буде «`disabled`», то потрібно запустити служби, скориставшись командами [12]:

```
sudo systemctl enable kubelet.service
sudo systemctl enable docker.service
```

8) Розгортаємо `Docker daemon`, як це наведено на рис. 1.5 [12, 13].

Після розгортання цієї служби треба одразу ж перезавантажити Docker, використовуючи команди [12]:

```
sudo systemctl daemon-reload
sudo systemctl restart docker
```

```
sudo bash -c 'cat > /etc/docker/daemon.json <<EOF
{
  "exec-opts": ["native.cgroupdriver=systemd"],
  "log-driver": "json-file",
  "log-opts": {
    "max-size": "100m"
  },
  "storage-driver": "overlay2"
}
EOF'
```

Рисунок 1.5 – Розгортання служби `Docker daemon`

Кроки, що були вище розглянуті, відносяться до підготовчих робіт. Після їх виконання можна починати безпосередньо створювати кластер.

Перш за все зазначимо, що для того, щоб налаштувати мережу в кластері, існує готове рішення під назвою Calico. Потрібно завантажити відповідний за назвою YAML-файл з метою застосування налаштувань до кластера. Файл можна завантажите використовуючи команду і посилання, що показані нижче [12, 13]:

```
wget https://docs.projectcalico.org/manifests/calico.yaml
```

Необхідно відкрити завантажений файл `calico.yaml` і знайти там поле `CALICO-IPV4POOL-CIDR`, – саме воно буде потрібне для створення кластера [13].

Далі потрібно провести ініціалізацію кластера використовуючи команду [12]:

```
sudo kubeadm init —pod-network-cidr=192.168.9.0/16,
```

де 192.168.0.0/16 – значення із поля `CALICO_IPV4POOL_CIDR` файлу `calico.yaml`.

Після ініціалізації виводиться повідомлення про успішність виконання процесу: «Your Kubernetes master has initialized successfully» [11].

Наступним кроком потрібно створити користувача для системи Kubernetes, що досягається виконанням команд [12]:

```
mkdir -p $HOME/.kube
sudo cp -i /etc/kubernetes/admin.conf $HOME/.kube/config
sudo chown $(id -u):$(id -g) $HOME/.kube/config
```

Як можна бачити, в результаті буде створений користувач `admin` для роботи в кластері Kubernetes.

Останнім кроком потрібно застосувати файл `calico.yaml` для встановлення конфігурації мережі в кластері, застосувавши команду [12]:

```
kubectl apply -f calico.yaml.
```

Якщо все зроблено правильно, то кластер буде функціонувати і далі в ньому можна розгортати потрібні сервіси і додатки.

Таким чином, платформа Kubernetes дає можливість організаціям легше здійснювати управління складними мікросервісними додатками, забезпечуючи при цьому високу доступність, масштабованість і надійність. Вона дає змогу визначити бажаний стан необхідного додатку у вигляді коду (YAML-файлів), а потім автоматично створює і запускає контейнери на основі цих визначень. Крім того, Kubernetes стежить за збоями та автоматично відновлює контейнери і вузли. Також ця система розподіляє навантаження між екземплярами програми, забезпечуючи рівномірне використання ресурсів. Опис додатку відбувається у вигляді декларативних файлів, що спрощує управління і полегшує масштабування. Також варто відзначити підтримку найпоширеніших хмарних платформ і можливість розгортання цієї системи на власних серверах. Потенціал Kubernetes можна легко розширювати за допомогою доповнень, що дає змогу адаптувати його під специфічні потреби [1].

При цьому необхідно мати на увазі і наявність недоліків у Kubernetes. Зокрема, по мірі збільшення складності мікросервісних додатків, відладка може стати важкою задачею. Помилка в одному з компонентів може вплинути на всю систему. Крім того, виконання налаштувань і здійснення управління кластером також може стати досить складною задачею для новачків [1].

2 АНАЛІЗ ФУНКЦІОНАЛЬНИХ ОСОБЛИВОСТЕЙ ОСНОВНИХ ТИПІВ КОНТРОЛЕРІВ, ЯК РЕСУРСІВ, ЩО ЗАБЕЗПЕЧУЮТЬ ПРОЦЕСИ УПРАВЛІННЯ У СИСТЕМІ KUBERNETES

Як було вище зазначено, основним компонентом платформи Kubernetes є Pod – найменша і найпростіша одиниця, що являє собою запит на запуск одного або більше контейнерів на одному вузлі. Це єдиний об'єкт в Kubernetes, який дозволяє запускати контейнери, причому контейнер ніяк не може існувати без поду. Тобто Pod – це мінімальний блок, з яким може працювати Kubernetes, що виражає виконання його процесів управління. Pod, як правило, містить у собі від 1-го до 5-ти контейнерів, що взаємодіють між собою різними способами . Зокрема для налаштування взаємодії контейнерів усередині поду використовуються спеціалізовані патерни, що будуть розглянуті у наступному розділі [14].

Далі детальніше проаналізуємо контролери Kubernetes, в аспекті його функціональних ресурсів, що надають можливості різними способами здійснювати управління набором подів залежно від вимог тієї чи іншої поставленої задачі.

2.1 Контролер реплікації та набір реплік

Контролер RC – це ресурс Kubernetes, який займається забезпеченням постійної дієздатності його модулів (подів). Якщо под зникає, наприклад, у разі відмови вузла кластера або його було виключено із вузла, то RC помічає відсутній под і створює под на заміну. На рис. 2.1 продемонстровано приклад роботи RC у випадку, коли вузол падає і водночас втрачається два поди. Припустимо, що под A був створений без посередника, і тому ним не можна здійснювати управління, тоді як под B управляється RC-контролером. Після аварійного завершення роботи вузла, RC створює новий под (B2) для заміни того, що був втрачений (B). При цьому под A втрачається повністю, бо його неможливо відтворити. Контролер RC на рис. 2.1 здійснює управління тільки одним подом, але такі контролери загалом призначені для створення і забезпечення управління кількома копіями (репліками) поду [13].

RC-контролер постійно спостерігає за тим, щоб фактична кількість подів визначеного набору завжди збігалася з необхідним їх числом. Якщо запущено занадто мало подів цього набору (тобто тих, що відповідають певному селектору міток), то із шаблону поду створюються його репліки. І навпаки, якщо запущено

занадто багато таких подів, то зайві репліки мають бути видалені RC. Задача контролера полягає в тому, щоб забезпечити точну відповідність кількості подів з його селектором міток. Якщо немає відповідності, то RC виконує необхідні дії для приведення у відповідність фактичної кількості подів із необхідною. Алгоритм роботи RC-контролера наведений на рис. 2.2 [12].

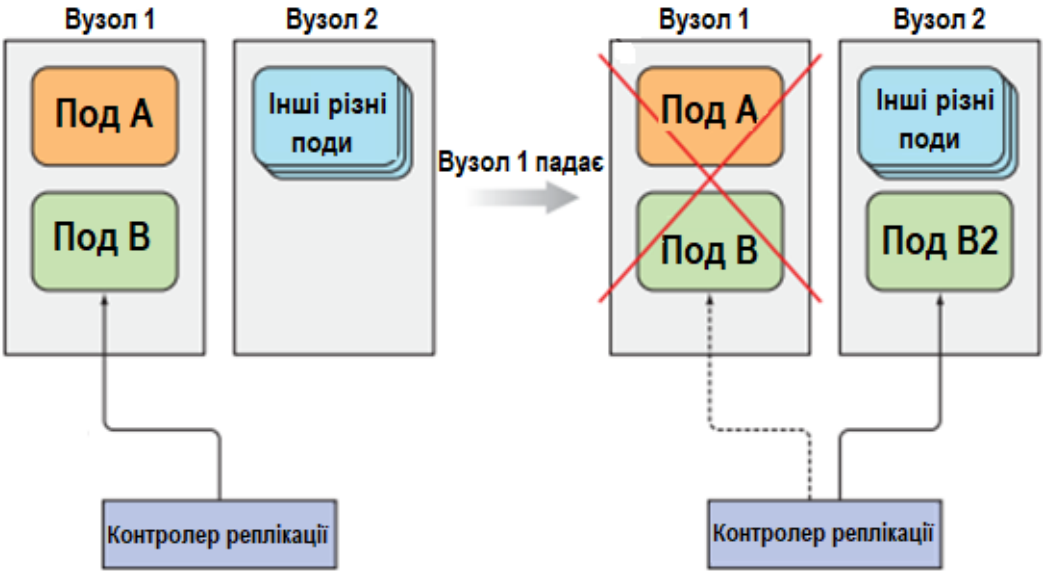

Рисунок 2.1 – Приклад роботи контролера реплікації

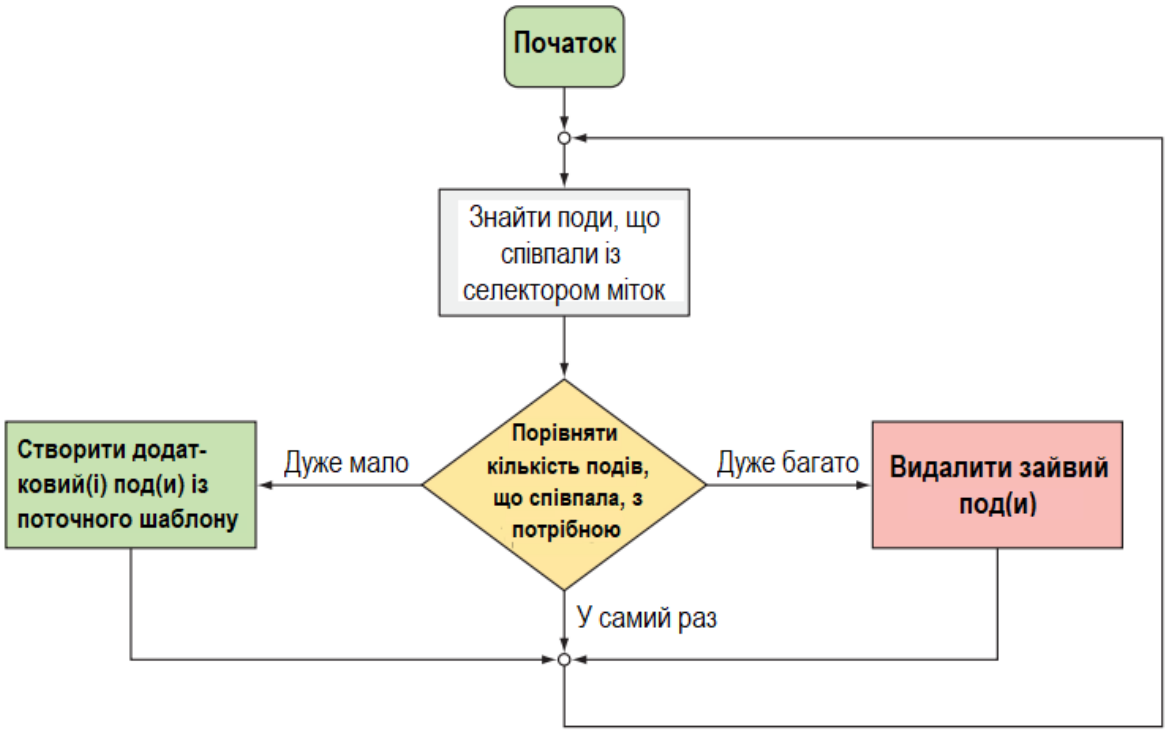

Рисунок 2.2 – Алгоритм роботи контролера реплікації

Подібно до модулів та інших ресурсів Kubernetes, контролер реплікації створюється шляхом пересилання дескриптору у вигляді файлу YAML на API-Server (див. рис. 1.2). Тобто для визначення RC-контролера створюється конфігураційний файл YAML: `example-app-rc.yaml`, – приклад якого наведено на рис 2.3 [13, 15].

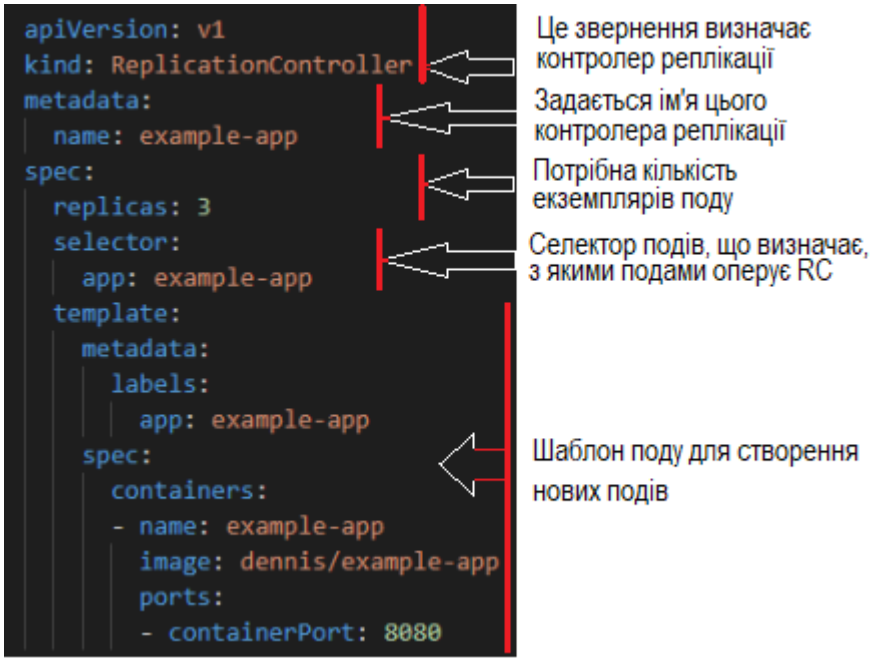

Рисунок 2.3 – Приклад визначення RC-контролера у вигляді файлу YAML: `example-app-rc.yaml`

Можна бачити, що у разі відправлення файлу на API сервер, Kubernetes створює новий контролер реплікації з ім'ям `example-app`, який гарантує, що три екземпляри подів завжди співпадають із селектором міток `app=example-app`. Коли не вистачає подів, нові поди будуть створені із наданого шаблону поду. При цьому у полі `spec:selector:` – вказується селектор, за яким контролер буде знаходити поди, а в полі `template:metadata:labels:` – вказуються мітки самих подів [13, 15].

Для того, щоб створити RC-контролер, використовується команда `kubectl create` [13]:

```
$ kubectl create -f example-app-rc.yaml
replicationcontroller "example-app" created
```

Первісно RC були єдиним компонентом Kubernetes для реплікації подів і зміни їхньої приписки в разі аварійного збою вузлів. Пізніше було введено альтернативний ресурс під назвою набір реплік (ReplicaSet), який на цей час є новим поколінням RC. Набір реплік поводиться так само, як і RC, але має більш виразні селектори поду. У той час як селектор міток в RC дає змогу вибирати тільки ті поди, які містять певну мітку, у селектор набору реплік було введено додаткові можливості [13]:

- `matchLabels` – перевіряє точну відповідність ключа і значення міток пода (фактично так само, як і в RC);

- `matchExpressions` – у селектор можна додавати додаткові вирази. Кожен вираз має містити ключ, оператор і, можливо (в залежності від оператора), список значень. Описи кожного оператора зведено до таблиці 2.1 [13].

Таблиця 2.1 – Оператори можливості matchExpressions

Оператор	**Функція, що виконується**
`In`	значення мітки **повинно** співпадати з одним із зазначених значень `values`
`NotIn`	значення мітки **не повинно** співпадати з будь-яким із зазначених значень `values`
`Exists`	под **повинен** містити мітку із зазначеним ключем (при цьому значення не важливе). Під час використання цього оператора не слід вказувати поле `values`
`DoesNotExist`	под **не повинен** містити мітку із зазначеним ключем (значення не важливе). Під час використання цього оператора не слід вказувати поле `values`

Якщо задано кілька виразів, то, для того, щоб селектор співпав із подом, усі ці вирази мають бути істинними. Якщо вказані і `matchLabels`, і `matchExpressions`, то, щоб под ототожнився з селектором, усі мітки мають збігатися, а всі вирази мають виявлятися істинними [13].

Приклад використання оператора `In` у селекторі `matchExpressions`, як фрагмент конфігураційного файлу `example-app -replicaset-matchexpressions.yaml`, наведено на рис. 2.4 [13].

Зазвичай у разі роботи з кластером ресурс ReplicaSet не створюється вручну, – цей набір використовується контролером більш вищого рівня – Deployment. RC і ReplicaSet у разі запуску подів не гарантують знаходження останніх на фіксованих вузлах, тому що планувальник (Scheduler) розподіляє

поди в залежності від завантаженості вузлів, і запитів на пам'ять та ЦП у подів. Але в Kubernetes існує ще один ресурс для управління набором подів, що має назву «набір демонів» (DaemonSet) [13].

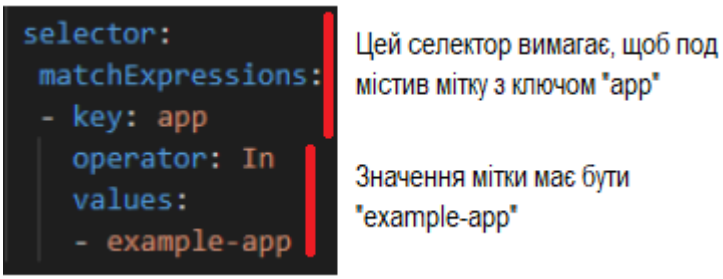

Рисунок 2.4 – Приклад застосування оператора `In` у селекторі `matchExpressions`

Зазвичай у разі роботи з кластером ресурс ReplicaSet не створюється вручну, – цей набір використовується контролером більш вищого рівня – Deployment. RC і ReplicaSet у разі запуску подів не гарантують знаходження останніх на фіксованих вузлах, тому що планувальник (Scheduler) розподіляє поди в залежності від завантаженості вузлів, і запитів на пам'ять та ЦП у подів. Але в Kubernetes існує ще один ресурс для управління набором подів, що має назву «набір демонів» (DaemonSet) [13].

2.2 Контролер DaemonSet

І RC, і набори ReplicaSet використовуються для запуску певної кількості подів, які розгорнуті в будь-якому місці кластера Kubernetes. Але існують деякі випадки, коли потрібно, щоб под працював на кожному вузлі в кластері. Ці випадки включають поди, що мають відношення до забезпечення роботи інфраструктури, виконують операції системного рівня. Наприклад, потрібно на кожному вузлі запустити збирача логів і монітор ресурсів. Контролер DaemonSet якраз і є таким ресурсом Kubernetes, який дає змогу запускати поди по одному на визначених вузлах. Ще, для прикладу, можна навести власний процес `kube-proxy` системи Kubernetes, який повинен працювати на всіх вузлах, щоб примушувати служби працювати. Таким чином, набори демонів запускають

тільки одну репліку поду на кожному вузлі, тоді як, розгляінуті до цього ReplicaSet – розкидають їх по всьому кластеру випадковим чином (рис. 2.5). Інакше кажучи, DaemonSet гарантовано створює стільки подів, скільки є вузлів, і розгортає кожен под на своєму вузлі [13].

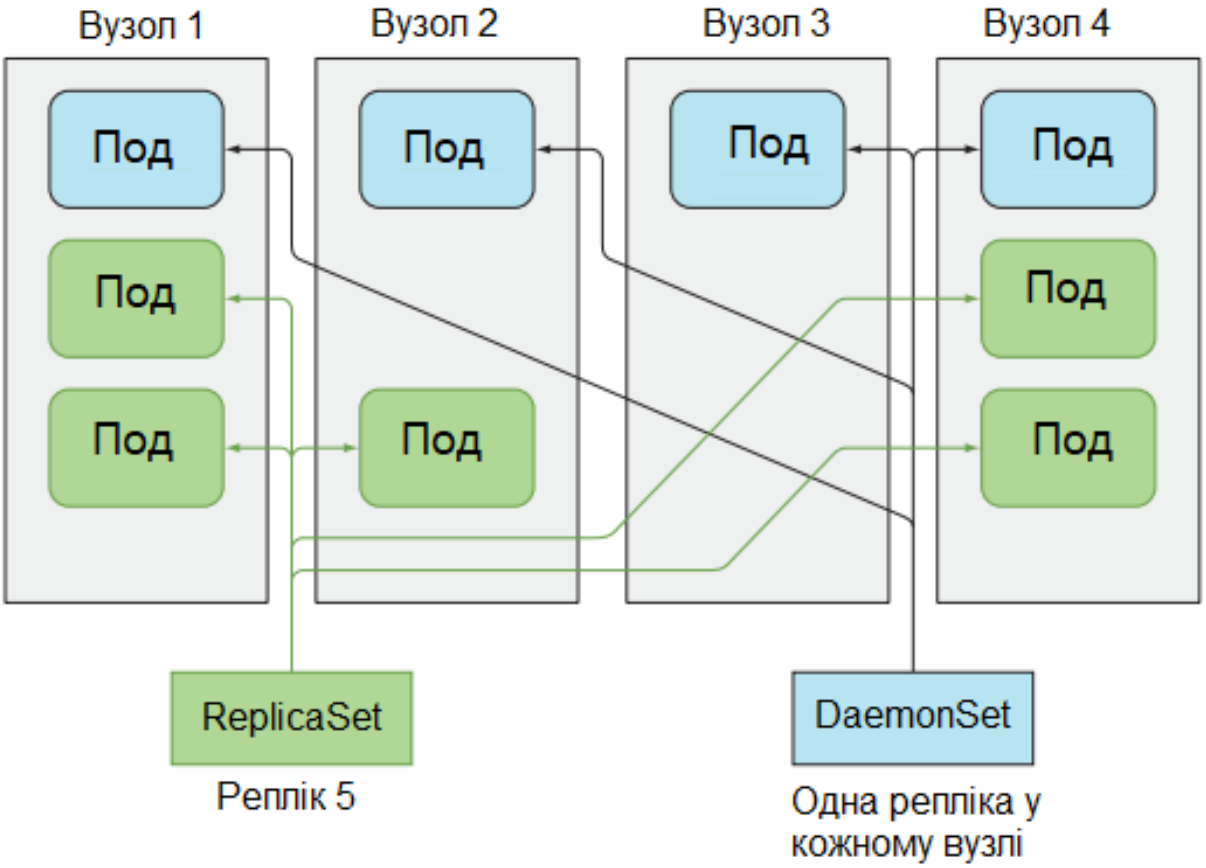

Рисунок 2.5 – Запуск подів контролерами DaemonSet та ReplicaSet

Якщо вузол падає, то контролер DaemonSet не буде створювати под в іншому місці. Але коли новий вузол додається в кластер Kubernetes, то контролер негайно розгортає в ньому нову репліку поду. Він також це робить у разі, якщо один із подів був випадково видалений. Як і ReplicaSet, DaemonSet створює модуль із сконфігурованого в ньому шаблону поду [13].

За замовчуванням, якщо ніяк не обмежувати под, то його запускатимуть на всіх вузлах, і навіть на тих, які позначені як `unschedulable` (не призначаються), тому що DaemonSet не контролюється планувальником щодо розміщення подів. Припустимо, що на всіх вузлах, що містять, наприклад, твердотільний накопичувач (SSD), працює демон на ім'я `ssd-metrics-monitor`. Створюється контролер DaemonSet, який запускає цей набір демонів на

всіх вузлах, які позначено як такі, що мають SSD. Адміністратори кластера додали мітку `disk:ssd` в усі такі вузли, тому контролер DaemonSet створюється за допомогою селектора вузлів, який обирає тільки вузли, що мають цю мітку. Для обмеження кількості вузлів, на яких потрібно запускати поди, потрібно встановити властивість `nodeSelector` у шаблоні поду, який описується в конфігураційному YAML-файлі контролера DaemonSet. На рисунку 2.6. наводиться приклад опису DaemonSet у конфігураційному файлі `ssd-metrics daemonset.yaml` із селектором вузлів, на яких встановлений `ssd disk` [13].

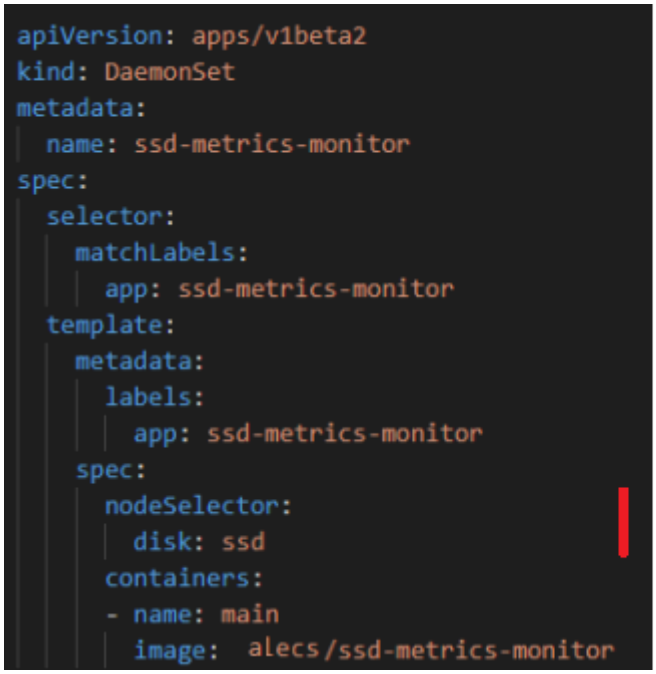

Рисунок 2.6 – Приклад конфігураційного файлу для опису контролера DaemonSet із селектором вузлів, на яких встановлено диск SSD

DaemonSet запускатиме под з одним контейнером на основі образу контейнера `alecs/ssd-metrics-monitor`. Репліку цього поду буде створено для кожного вузла, що має мітку `disk:ssd` [13].

2.3 Контролер StatefulSet

Контролер StatefulSet – є ресурсом в системі Kubernetes, що надає змогу запускати додатки із збереженням стану. Цей контролер, на відміну від

розглянутих вище RC і ReplicaSet, працює з подами дещо по-іншому. Він дозволяє здійснювати запуск додатків із забезпечення їм стабільного сховища та стабільного мережного імені [16].

У тому разі, коли додаток не відповідає, або ж вузол, на якому запущено цей под, перестав функціонувати, – Kubernetes створює інший под, але з такою ж самою IP-адресою, і інтегрує його в те ж саме сховище. Це дає змогу розгортати будь-які додатки, навіть кластери баз даних у кластері Kubernetes [16].

2.4 Контролери Job і CronJob

У функціонуванні будь-якої системи існують спеціальні типи додатків, які запускаються задля досягнення певної мети. Часто потрібно запускати консольний додаток для будь-якої синхронізації, якщо сталася, наприклад, інтеграція із зовнішнім ресурсом. Такі додатки необхідно запускати тільки для того, щоб виконати свою роботу, і після цього додаток має бути зупинений [13].

У Kubernetes для таких цілей передбачено окремий вид контролерів під назвою Job (Завдання). Наприклад, у разі аварійного завершення роботи вузла, поди на цьому вузлі, управління якими здійснюється контролером Job, будуть перепризначені на інші вузли (так само, якби подом керував контролер ReplicaSet). У разі збою самого процесу (тобто коли процес повертає код виходу з помилкою), Job може бути налаштований або на здійснення перезапуску контейнера, або ні [17].

Приклад визначення ресурсу контролера Job у конфігураційному YAML-файлі показаний на рис. 2.7 [13, 17].

У секції специфікації поду (`spec:`) можна задати, що слід робити Kubernetes після завершення процесів, запущених у контейнері. Це здійснюється за допомогою властивості `restartPolicy:` в секції `spec:` поду, яка за замовчуванням повинна мати значення `Always`. Поди, які управляються Job, не можуть використовувати цю політику, що прийнята за замовчуванням, оскільки вони не призначені для необмеженого застосування. Отже, необхідно явно задати значення перезапуску виду `OnFailure` або `Never`. Таке налаштування запобігає перезапуску контейнера після закінчення його роботи [13, 17].

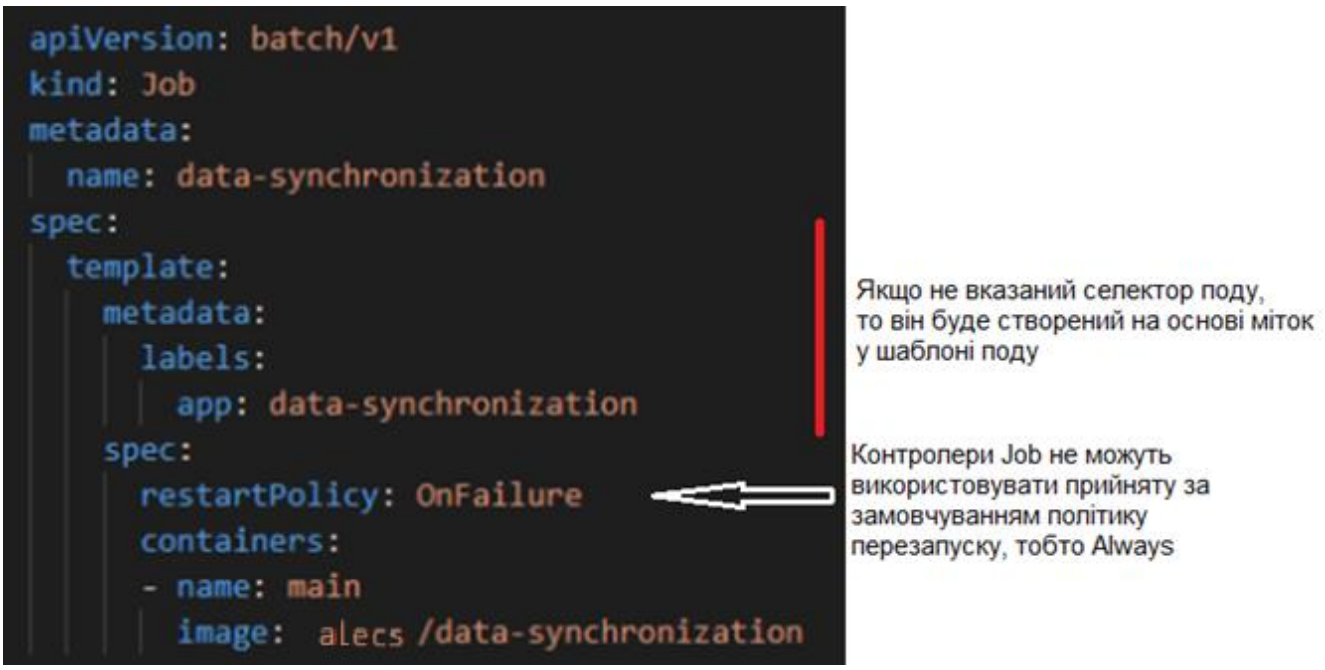

Рисунок 2.7 – Приклад визначення ресурсу контролера Job у конфігураційному YAML-файлі

В спецификации Job-а можно указывать поле completions: со значением, которое будет равным некоторому числу. Это поле указывает Kubernetes, что данный под нужно запустить последовательно указанное количество раз. Сначала создается один под, и когда он успешно отработает, то при его уничтожении создается новый, и так будет продолжаться, пока все задания не будут завершены успешно. Также, если разрешено запускать несколько подов Job-а параллельно, то нужно указать свойство spec:parallelism с требуемым числом возможных параллельных заданий (рис. 2.8) [13].

```
apiVersion: batch/v1
kind: Job
metadata:
  name: multi-completion-batch-job
spec:
  completions: 5
  parallelism: 2
  template:
          ...
```

Рисунок 2.8 – Приклад запуску подів Job декілька разів і можливості їх паралельного виконання

Специфікація Job, що наведена на рис. 2.8, буде запускати п'ять подів один за одним. Спочатку Job створює один под, а коли контейнер поду закінчується, він створює другий под і так доти, доки п'ять подів не будуть завершені успішно. Крім того, встановивши `parallelism` рівним 2, Job створює два поди і запускає їх паралельно [13].

У системі Kubernetes є ще один тип контролера, який потрібний для запуску Job-а за розкладом, і він називається CronJob. Тобто контролер CronJob – це ресурс Kubernetes, у якому в секції `spec:schedule` вказується час запуску Job-а в спеціальному форматі. Формат поля `schedule:` має такий вигляд: * * * *. Тут «*» має сенс «у кожний» (тобто «кожну хвилину», «кожну годину» тощо) [18].

Припустимо, що Job має виконуватися кожні 15 хв. Приклад конфігураційного YAML-файлу в цьому випадку матиме вигляд, що демонструє рис. 2.9 [13].

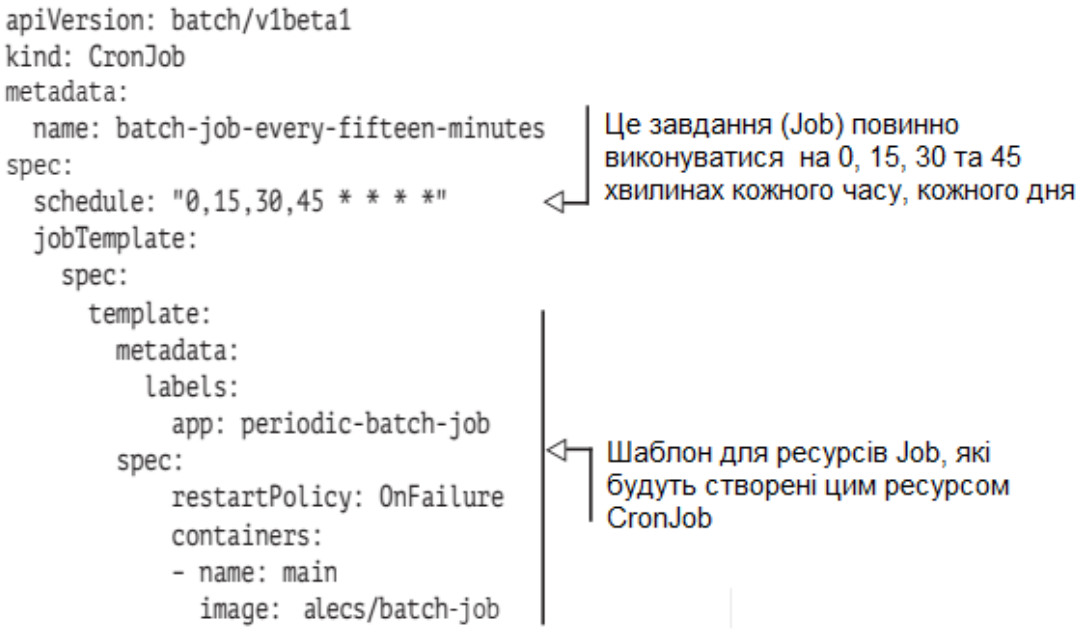

Рисунок 2.9 – Приклад конфігураційного YAML-файлу для контролера CronJob

В приведенном примере требуется запускать задание каждые 15 минут, поэтому расписание должно быть «0,15,30,45 * * * *», что означает на отметке в 0, 15, 30 и 45 минут каждого часа (первая звездочка), каждого дня месяца (вторая звездочка), каждого месяца (третья звездочка) и каждого дня недели (четвертая звездочка) [13].

2.5 Контролер Deployment

Як тільки був запущений кластер Kubernetes (див. підрозділ 1.3), то на ньому можна розгорнути різні контейнеризовані додатки та сервіси (тобто поди, що їх утворюють). Для запуску таких подів у кластері можна, як було вище показано, використовувати ресурси контролерів RC або ReplicaSet. Але для того, щоб запустити, у разі необхідності, оновлені варіанти подів, потрібно забезпечити певне управління трафіком, який формується ними або надходить до них. Треба спочатку запустити ReplicaSet з контейнерами нової версії подів, далі переспрямувати трафік на цю версію, і тільки після цього виконати видалення старої версії набору реплік. Для спрощення та автоматизації описаних операцій реалізації цього процесу в системі Kubernetes було реалізовано спеціальний контролер деплоймент (Deployment), який визначає, як створювати й оновлювати (тобто розгортати) екземпляри подів нових додатків. Після створення деплоймента Control Plane у Kubernetes (див. рис. 1.2) запланує запуск екземплярів подів додатка на окремих вузлах у кластері [19].

Після того, як нові поди додатка були створені, контролер Deployment буде безперервно відстежувати їх. Якщо вузол, на якому розміщений под, наприклад, вийшов з ладу або був видалений, то Deployment замінить цей под реплікою на іншому вузлі в кластері. Цей процес є механізмом самовідновлення, що забезпечує роботу кластера Kubernetes у разі виникнення апаратних несправностей, або технічних робіт, або оновлення контейнеризованих додатків і сервісів системи. Таким чином, можна зазначити, що контролер Deployment надає ресурси вищого рівня, ніж контролери RC або ReplicaSet. Він дає змогу проводити розгортання та оновлення додатку за допомогою декларативного підходу. Створювати та здійснювати управління деплойментами можна через консольний інструмент Kubernetes під назвою `kubectl`, який використовує Kubernetes API для роботи з кластером. У разі створення процесу такого розгортання потрібно вказати образ контейнера додатка та кількість запущених реплік поду. Згодом ці параметри можна змінити. Наприклад, щоб розгорнути застосунок із 3 репліками потрібно створити YAML-конфігурацію, яка показана на рис. 2.10 [13, 19].

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: asp-net-app
spec:
  replicas: 3
  template:
    metadata:
      name: asp-net-app
      labels:
        app: asp-net-app
    spec:
      containers:
      - image:  alecs/asp-net-app:v1
        name: asp-net-app
```

Рисунок 2.10 – Приклад файлу конфігурації деплоймента, що створює додаток із 3 репліками

3 ЗАСТОСУВАННЯ ПАТЕРНІВ KUBERNETES ДЛЯ УПРАВЛІННЯ ВЗАЄМОДІЄЮ КОНТЕЙНЕРІВ У ПОДАХ

3.1 Загальні поняття та особливості патернів

Первісне значення терміна «патерн» (pattern – шаблон) у контексті проектування будь-якої загальної архітектури було визначено в такому трактуванні: «Кожен патерн дає опис тієї чи іншої проблеми, яка регулярно виникає в навколишньому просторі, що нас оточує, слідом за яким надається суть вирішення цієї проблеми, яка викладається таким чином, щоб ви могли багаторазово використовувати це рішення, але ніколи не копіювати його». Щодо сфери IT і проектування програмного забезпечення, то тут патерн (або шаблон проектування) визначають як архітектурну конструкцію, що повторюється, та яка пропонує розв'язання проблеми проектування в межах певного контексту, що часто виникає. Тобто більш простими словами термін патерн описує розв'язання задачі, що повторюється [16].

Особливістю патернів є те, що вони не просто надають рішення, але й формують компактну мову, в основі якої лежать іменники, внаслідок чого кожен патерн має унікальну назву. Загалом можна зазначити, що коли люди згадують ці назви в розмовах між собою, вони автоматично викликають у них схожі ментальні уявлення. Наприклад, коли в контексті хмарних обчислень чи об'єктно-орієнтованого програмування розробник оперує терміном «фабрика» (Fabric), то одразу ж розуміється об'єкт, який продукує інші об'єкти. Ще важливою особливістю патернів є те, що вони дотримуються жорсткого формату, але водночас у кожній галузі, у кожній системі або технології (технологічному процесі) визначають свій формат надання рішень, тобто, інакше кажучи, немає єдиного стандарту, який визначав би, як мають подаватися патерни [16].

Патерни взаємопов'язані між собою і можуть перекриватися, тому разом охоплюють більшу частину простору завдань. Крім того, патерни мають різні рівні деталізації та сфери дії: більш загальні патерни охоплюють ширший спектр задач і пропонують досить приблизні рекомендації щодо їх вирішення, а спеціалізовані патерни дають цілком конкретне рішення, але застосовуються не так широко [16].

Стосовно системи Kubernetes, в аспекті аналізу принципів управління контейнеризованими додатками та сервісами, існує кілька патернів для управління взаємодією контейнерів в одному поді, серед яких основними є: `Sidecar`, `Adapter` і `Ambassador`. Проаналізуємо їх більш детально.

3.2 Патерн `Sidecar`

Як попередньо було зазначено, Kubernetes - це система управління контейнерами з відкритим вихідним кодом для автоматичного розгортання, масштабування та управління контейнеризованими додатками та сервісами. Базовим поняттям і компонентом під час проектування додатків у Kubernetes є Pod. Kubernetes оперує подами, а не контейнерами, але при цьому поди містять у собі контейнери. Под може містити в собі описи одного або декількох контейнерів, розділів, що монтуються, IP-адрес і налаштувань того, як контейнери мають працювати всередині поду. Под, що містить один контейнер, – належить до одноконтейнерних подів (як зазначалося, це найпоширеніший варіант їх використання в системі Kubernetes), а под, що містить кілька зв'язаних контейнерів, – належить до мультиконтейнерних подів. Патерн `sidecar` використовується для управління мультиконтейнерними подами [20].

Патерн `sidecar` – це `Sidecar`-контейнер, який має бути запущений поруч з основним контейнером усередині поду. Цей патерн потрібен для розширення і поліпшення функціональності основного додатка без внесення в нього змін. Він належить до базових патернів контейнерів і дозволяє створювати вузькоспеціалізовані контейнери, які тісно взаємодіють один з одним [16, 20].

Загалом, контейнер інкапсулює в собі роботу однієї команди, і додаток, що знаходиться всередині, розгортається самостійно. Тому найкращим варіантом додавання будь-яких додаткових функцій буде винесення цих функцій до іншого контейнера, запуск нового контейнера в поді поряд з основним, і надання можливості взаємодії між основним і додатковим контейнером. Тобто можна виокремлювати однотипні функції в додаткові контейнери і компонувати свої додатки за допомогою кількох необхідних. Патерн `Sidecar` якраз надає таку можливість у розробці різних систем [20].

Взаємодія контейнерів у поді у разі використання патерна `sidecar` показана на рис. 3.1. На схемі можна бачити такі елементи [16]:

- основний контейнер додатку, що обслуговує файли через сервер HTTP;

- допоміжний контейнер (`sidecar`-контейнер), що діє паралельно з основним і отримує дані із сервера Git;

- обмін даними між основним і допоміжним `Sidecar`-контейнером здійснюється у загальному для них поді.

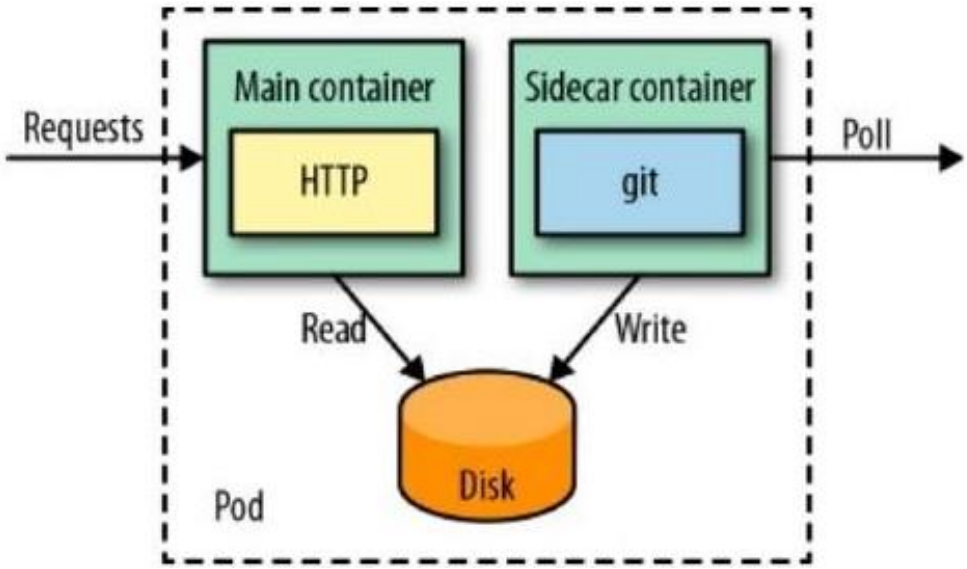

Рисунок 3.1 – Подання патерна `Sidecar`

Тут контейнер синхронізації з Git додає контент для обслуговування сервером HTTP і підтримує його в активному стані. Зауважимо, що обидва контейнери діють у тісній взаємодії і є однаково важливими, але патерн `Sidecar` розширює їх колективну поведінку. Як правило, головний контейнер завжди ініціалізується першим і є контейнером за замовчуванням (його запускають командою `kubectl exec`). Патерн `Sidecar`, як зазначалося, забезпечує тісний взаємозв'язок контейнерів при виконанні процесів управління і водночас дає змогу розділити задачі, що можуть належати окремим колективам розробників, які використовують, наприклад, різні мови програмування з різними версіями, циклами оновлень, тощо. Він також сприяє взаємозамінності та багаторазовому використанню контейнерів, наприклад, у тому розумінні, що HTTP-сервер і механізм синхронізації Git можна повторно використовувати в інших додатках та з іншими налаштуваннями, як самостійно, так і спільно з іншими контейнерами [16].

У разі розгортання в Kubernetes поду з кількома контейнерами всередині, вони мають у своєму розпорядженні загальні ресурси у вигляді загальних томів (на рис. 3.1 жорсткий диск (Disc)). Також запити можуть оброблятися через локальну мережу, що дає змогу задати фіксовану адресу для відправлення та приймання запитів через `localhost:<номер порту>`. Прикладом використання патерну `Sidecar`, на який треба звернути увагу, є проект Istio. У ньому реалізовано безліч функцій, зокрема авторизацію, автентифікацію, трасування і повторні виклики саме за допомогою цього патерну [20].

3.3 Патерн `Adapter`

Патерн `Adapter` дає змогу привести гетерогенну контейнерну систему у відповідність до єдиного уніфікованого інтерфейсу, що має стандартизований і нормалізований формат, який може використовуватися зовнішнім оточенням. Патерн `Adapter` – це спеціалізований варіант патерну `Sidecar` (усі свої характеристики цей патерн перейняв від патерну `Sidecar`), а його єдиною метою є надання адаптованого доступу до додатку. [16, 21].

Узагалі, контейнери дають змогу уніфікувати упакування та запуск додатків, що написані різними мовами, з використанням різних бібліотек і різних технологій, тобто фактично створюються розподілені системи, що складаються з різнорідних (гетерогенних) компонентів. Ця гетерогенність може створювати труднощі, якщо всі компоненти є потреба обробляти іншими системами в однаковий спосіб. І тут патерн `Adapter` пропонує рішення, яке допомагає приховати цю складність системи та надати уніфікований інтерфейс для доступу до неї [16].

Розглянемо цей патерн на такому прикладі. Основною умовою успішної експлуатації розподілених систем є повний всебічний моніторинг і можливість надсилання сповіщень. Крім того, якщо розподілена система складається з декількох служб, для їх моніторингу можна використовувати зовнішній інструментарій, що видобуває і зберігає метрики кожної служби. Однак служби можуть бути написані різними мовами та можуть мати різні можливості і надавати метрики в різних форматах, що будуть відрізнятися від тих, які очікуються інструментарієм, який здійснює моніторинг. Така різноманітність

створює проблему для моніторингу різнорідних застосунків із використанням єдиного рішення, яке очікує підтримки єдиного формату всією системою. Патерн `Adapter` дає змогу організувати уніфікований інтерфейс моніторингу шляхом експорту метрик із різних контейнерів додатків в один стандартний формат і протокол [21].

На рис. 3.2 наведено подання `Adapter`-контейнеру, який перетворює інформацію з метриками, що зберігається локально, у зовнішній формат, що розуміє сервер моніторингу [16, 21].

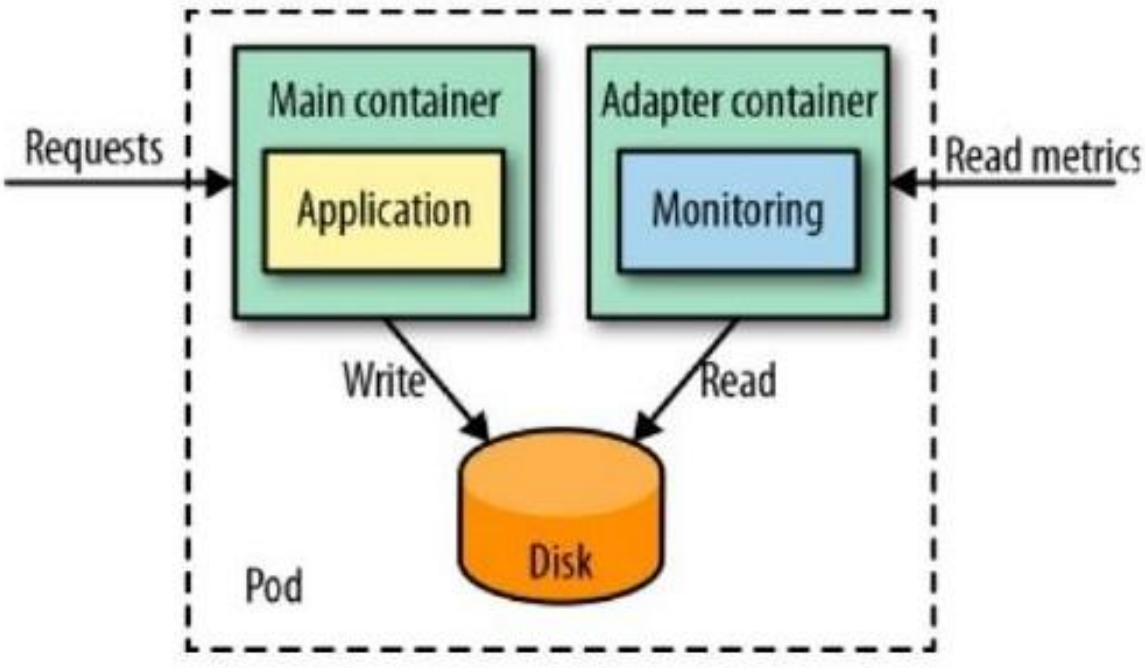

Рисунок 3.2 – Подання патерна `Adapter`

У разі такого підходу кожна служба, що представлена окремим подом, буде крім основного контейнера додатка, містити ще один контейнер, який «знає», як прочитати метрики додатка та перетворити їх у формат, що буде зрозумілим для інструментарію, який здійснює моніторинг. Наприклад, можна мати один `Adapter`-контейнер, який «знає», як експортувати метрики Java через HTTP, та інший `Adapter`-контейнер, в іншому поді, який «знає», як експортувати метрики Python через HTTP. Інструментарію, який здійснює моніторинг, усі метрики будуть доступні через сервер HTTP у загальному нормалізованому форматі [16, 21].

3.4 Патерн Ambassador

Патерн Ambassador – це спеціалізований варіант патерну Sidecar, що відповідає за приховування складності та надає уніфікований інтерфейс для доступу до служб поза межами поду. Патерн Ambassador може виступати як інтелектуальний проксі-сервер та захищати поди від прямого доступу до залежностей зовнішнього середовища [16, 22].

Підхід щодо реалізації та управління мікросервісними архітектурами досить складний, оскільки навіть один сервіс не може існувати без залежностей від інших служб. Оскільки мікросервіси створюються таким чином, щоб виконувати лише одну певну закінчену функціональність, то буде неправильно реалізовувати відповідну логіку роботи із зовнішніми службами в цьому ж мікросервісі. У хмарних технологіях розробки такої логіки заведено виносити в окремий контейнер, щоб не перевантажувати основний, і мати можливість повторно використовувати цей функціонал. Крім того, буває корисним під час розробки, коли сервіс запущений для тестування функцій системи, підміняти сервіси на інші, відмінні від промислового оточення. Саме цим цілям і служить патерн Ambassador. Тобто іншими словами можна сказати, що абстрагування та ізоляція логіки доступу до інших служб у зовнішньому оточенні якраз і є метою використання патерну Ambassador [22].

Використання патерну стає зрозумілим на наступних прикладах. Так можна навести приклад, де для доступу до локального кешу в оточенні для розроблення може бути достатньо визначити деякі налаштування в конфігурації, але в промисловому оточенні може знадобитися налаштувати клієнт, що підключається до різних сегментів кешу. Наступним прикладом є пошук служби в реєстрі для її виявлення на стороні клієнта. Інший приклад –це взаємодія із зовнішньою службою через ненадійний протокол, такий як HTTP, через що для захисту додатку доводиться використовувати логіку обробки обривів зв'язку, налаштовувати тайм-аути, виконувати повторні спроби зв'язку і багато іншого. У всіх цих випадках можна використовувати Ambassador контейнер-посередник, що приховує складність доступу до зовнішніх служб і забезпечує спрощене їх подання для основного контейнера додатку через локальне мережне з'єднання [16].

На рис. 3.3 показано, як Ambassador контейнер-посередник допомагає відокремити прикладну логіку від логіки доступу до сховища пар

«ключ/значення», здійснюючи прийом запитів через локальний порт. На цьому рисунку видно, як можна делегувати доступ до даних, що зберігаються у віддаленому розподіленому сховищі, такому як `etcd`.

Під час використання такий `Ambassador` контейнер-посередник легко замінити локальним сховищем пар «ключ/значення», таким як, наприклад, `memcached` замість `etcd client`. У цьому випадку віддалений розподілений доступ до сховища `etcd` організовувати не потрібно [16].

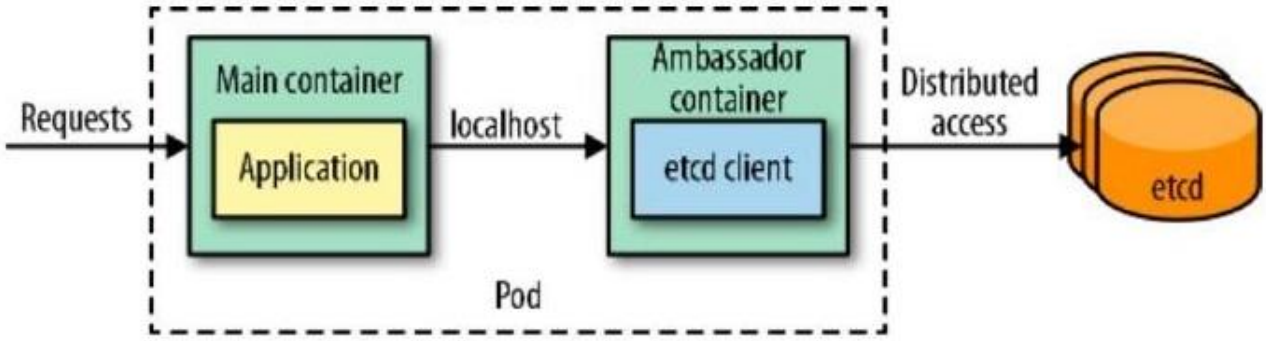

Рисунок 3.3 – Подання патерна `Ambassador`

Патерн `Ambassador` характеризується такими ж самими перевагами, що й патерн `Sidecar` – обидва сприяють вузькій спеціалізації контейнерів і придатності їх до багаторазового використання. У разі використання цього патерну застосунок може зосередитися на розв'язанні бізнес-завдань і делегувати відповідальність за використання зовнішньої служби іншому спеціалізованому контейнеру. Це також дає змогу створювати спеціалізовані та багаторазово використовувані `Ambassador` контейнери-посередники, які можна об'єднувати з іншими контейнерами додатків [16, 22].

4 РЕАЛІЗАЦІЯ СТРАТЕГІЙ РОЗГОРТАННЯ МІКРОСЕРВІСІВ В СИСТЕМІ KUBERNETES

Існує багато методів розгортання додатків та управління ними у середовищі розробника, тому вибір правильної стратегії є важливим рішенням, що дозволяє зважити варіанти з точки зору впливу змін на систему та на кінцевих користувачів. Однак найчастіше вибір стратегії обмежується або виділеним часом на реалізацію деплойменту, або бюджетом на інфраструктуру, що відноситься до найчастіших проблем, які обмежують можливості прискорення процесу розгортання системи. У Kubernetes існує кілька основних типів стратегій розгортання систем. До них можна віднести: `rolling update`, `recreate`; `blue/green` [23, 24].

Проаналізуємо та обґрунтуємо кожну з них детальніше.

4.1 Стратегія `Rolling update` – поступовий деплоймент

У Kubernetes – це стандартна стратегія розгортання. Її особливістю є поступова одна за одною заміна подів зі старою версією на поди з новою версією, причому без простою кластера. Kubernetes чекає готовності нових подів до роботи, перш ніж приступити до згортання старих (для визначення готовності проводяться так звані тести `readiness`). Kubernetes чекає, коли щойно запущений под буде кілька разів стабільно повертати успішне виконання перевірки готовності, і тільки після цього починається знищення подів із старою версією [23, 25].

Якщо виникає якась проблема, подібне поступове оновлення можна перервати, не зупиняючи всього кластера. Нижче на рис. 4.1 показаний приклад YAML-файлу з описом специфікації для поступової стратегії деплоймента [25].

У разі поступового розгортання, оновлення подів відбувається за принципом оновлення, що накочується. Створюється вторинний набір реплік подів (ReplicaSet) з новою версією застосунку, потім кількість реплік старої версії зменшується, а нової - збільшується, доки не буде досягнуто потрібної кількості реплік. Реалізацію стратегії `Rolling update` у вигляді функціональних діаграм показано на рис. 4.2 [23].

```
spec:
  replicas: 3
  strategy:
    type: RollingUpdate
    rollingUpdate:
       maxSurge: 25%
       maxUnavailable: 25%
  template:
```

Рисунок 4.1 – Приклад специфікації поступової стратегії `Rolling update`

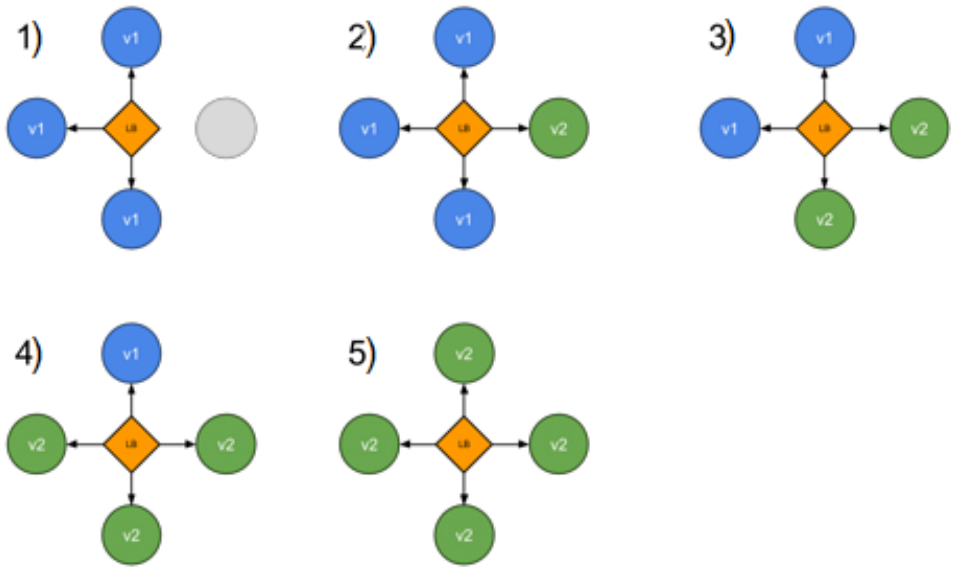

Рисунок 4.2 – Діаграми реалізація стратегії деплойменту `Rolling update`

Аналізуючи YAML-файл на рис. 4.1 необхідно звернути увагу на 2 важливі параметри [23]:

- `maxSurge` – цей параметр визначає скільком екземплярам поду можна існувати понад норму кількості реплік, яку встановлено в деплойменті. За замовчуванням використовується значення 25%. При перетворенні в абсолютне число, це значення заокруглюється вгору. Також можна встановлювати відразу абсолютні значення [23];

- `maxUnavailable` – цей параметр визначає скільки екземплярів поду може бути недоступними, щодо зазначеної кількості реплік під час оновлення на

нову версію програми. За замовчуванням також дорівнює 25%. Але при перетворенні в число, це значення буде заокруглено вниз [23].

Варіюючи значеннями цих двох параметрів можна керувати процесом розгортання залежно від потужностей кластера та необхідної швидкості розгортання [23].

4.2 Стратегія `Recreate` (повторне створення)

У цій стратегії деплойменту старі поди одночасно знищуються і замінюються на поди з новою версією додатку. Недоліком такого деплойменту є наявність часу простою, доки нові застосунки запустяться, пройдуть перевірки готовності, і почнуть приймати трафік від користувачів. Нижче на рис. 4.3 показано приклад YAML-файлу з описом специфікації для стратегії повторного створення подів [25].

```
spec:
  replicas: 5
  strategy:
    type: Recreate
  template:
```

Рисунок 4.3 – Приклад специфікації стратегії деплойменту `Recreate`

Перевагами цієї стратегії деплойменту є простота налаштування і те, що стан застосунку повністю оновлений. До недоліків можна віднести сильний вплив на користувача, а також те, що згаданий вище час простою залежить від тривалості вимкнення та завантаження додатку [25].

4.3 Стратегія `Blue/Green` – синьо-зелений деплоймент

Стратегія синьо-зеленого деплойменту передбачає одночасне розгортання старої (зеленої) і нової (синьої) версій додатку. Після розміщення обох версій

звичайні користувачі отримують доступ до зеленої, тоді як синя версія доступна для QA-команди для автоматизації тестів. Реалізацію стратегії `Blue/Green` у вигляді функціональних діаграм приведена на рис. 4.4 [23].

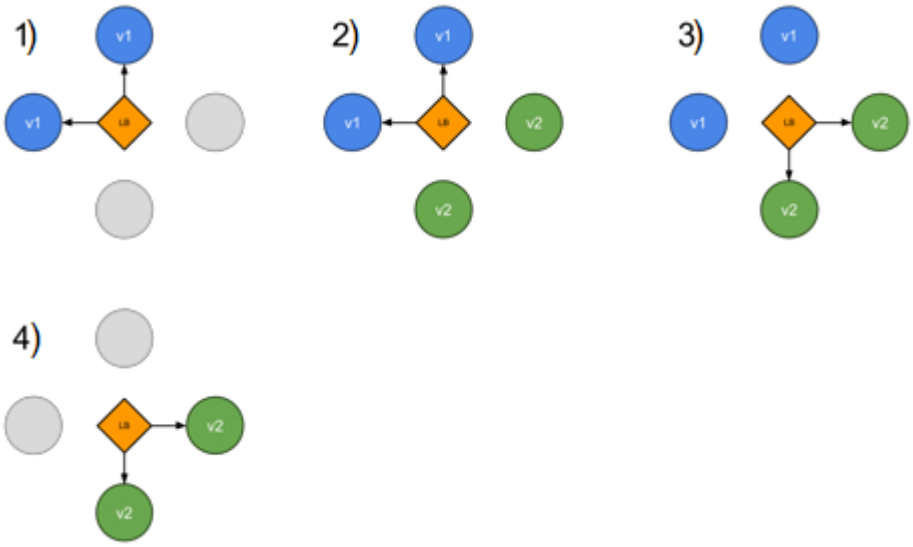

Рисунок 4.4 – Діаграми реалізації стратегії деплойменту `Blue/Green`

Перемикання на нову версію відбувається за допомогою зміни селектора міток, за якими спостерігає сервіс. Тобто у подів з новою версією буде вказано номер версії, і у сервісу також зміниться тільки цей селектор. Для кожного конкретного випадку обираються різні стратегії оновлення подів додатку [23].

Перевагами цієї стратегії деплойменту є можливість миттєвого відкочування і відсутність необхідності перевірки версій (весь стан додатку змінюється за один раз). До недоліків можна віднести високу вартість (ця стратегія вимагає подвоєння ресурсів), а також перед її розгортанням необхідно провести належне тестування всієї платформи. Крім того може бути складною робота з додатками із зміненим станом [25].

Ця стратегія буде корисною в разі, коли потрібно виконати реліз без простою з мінімальними зусиллями на написання логіки процесу розгортання, і немає жорстких вимог до скорочення/підтримки бюджету [25].

ВИСНОВКИ

В цій кваліфікаційній роботі проведений аналіз можливостей застосування системи Kubernetes для управління сервісами та додатками на рівні контейнерів. Ця система по суті є відкритим програмним забезпеченням, що застосовується для автоматизації розгортання, масштабування та управління контейнеризованими додатками. Серед існуючих рішень Kubernetes виділяється більш гнучкими та масштабованими підходами щодо здійснення такого управління і на цей час ресурси цієї системи широко застосовуються в хмарних інфраструктурах найвідоміших вендорів (AWS, Windows Azure, Google App) та DevOps практиках, що говорить про актуальність проведених досліджень.

У роботі показано, що спочатку сервіси розгортали безпосередньо на вузлах із вихідних кодів або інсталяційних пакетів. Подальша розробка і впровадження інструментарію управління конфігурацією (Configuration Management Tools), таких як puppet, ansible, chef, yaml, дозволило цей процес зробити більш автоматизованим, але зберігалася потреба у залученні фахівця для здійснення коректування і контролю за процесами планування інфраструктури сервісної платформи і написанні конфігураційних файлів для автоматизації одноманітних процесів і їх виконанням. Головним недоліком такого підходу є низька продуктивність. З появою мікросервісних архітектур на основі контейнерних технологій, коли контейнеризований додаток розбивається на невеликі сервіси, які розгортаються незалежно, найнеобхіднішим інструментом для управління цими складними системами став Kubernetes. Як і традиційні системи розгортання контейнерів (наприклад, Docker), Kubernetes дозволяє організувати платформи розробника, але також є багато можливостей і для вибору користувачам з наданням гнучкості там, де це важливо. Також у роботі було акцентовано увагу на те, що Kubernetes не обмежується лише оркестрацією контейнерів і мікросервісів. Система також здатна ефективно забезпечити управління монолітними додатками, додатками, що мають певні стани, інфраструктурними сервісами і навіть складними додатками, які працюють у гібридних і багатокластерних оточеннях [7].

У кваліфікаційній роботі проведений детальний аналіз архітектури системи Kubernetes, її компонентів, служб і ресурсів (рис. 1.2). Показано, що основним компонентом платформи Kubernetes є Pod, який у свою чергу становить найменшу та найпростішу одиницю, що являє собою запит на запуск одного або

більше контейнерів на одному вузлі. Це єдиний об'єкт в Kubernetes, який дозволяє запускати контейнери, причому контейнер ніяк не може існувати без поду. Тобто Pod – це мінімальний блок, з яким може працювати Kubernetes, що виражає виконання його процесів управління [1].

Головним контролюючим вузлом кластера є площина управління Kubernetes (CP), компоненти якої відповідають за прийняття рішень щодо стану кластера. На ній виконується планування і розподіл задач, а також відбувається управління ресурсами [1]. У роботі дається визначення і розкривається подання і інших елементів архітектури, що наведена на рис. 1.2 [1].

Показано, що практичною реалізацією архітектури Kubernetes є кластер, в основі розгортання якого закладені властивості модульності, розширюваності та горизонтальної масштабованості. Cluster Kubernetes складається з набору вузлів, які запускають контейнеризовані сервіси та додатки. У роботі надана покрокова методика і показані особливості розгортання кластера Kubernetes на базі інструментарію `minikube` і утиліти `kubeadm`, які дозволяють налаштувати одновузловий кластер [12].

У процесі виконання кваліфікаційної роботи було зроблено акцент на те, що у системі Kubernetes кожен процес управління є окремим контролером. Вони забезпечують автоматичне масштабування і управління відмовостійкістю, а також забезпечують підтримку правильної кількості екземплярів додатків і сервісів відповідно до заданих параметрів. Тому у другому розділі роботи детально проаналізовано контролери Kubernetes, в аспекті його функціональних ресурсів, що надають можливості різними способами здійснювати управління набором подів залежно від вимог тієї чи іншої поставленої задачі. Зокрема аналізувалися контролери реплікації (RC), набір реплік (ReplicaSet), DaemonSet, Job і CronJob, Deployment [1, 8].

Також у третьому розділі кваліфікаційної роботі звертається увага на те, що для налаштування взаємодії контейнерів усередині поду використовуються спеціалізовані патерни. В аспекті аналізу принципів управління контейнеризованими додатками та сервісами в Kubernetes існує кілька патернів для управління взаємодією контейнерів в одному поді, зокрема [16]:

- патерн `sidecar`, контейнер якого запускається поруч з основним контейнером усередині поду. Цей патерн потрібен для розширення і поліпшення функціональності основного додатка без внесення в нього змін;

- патерн Adapter, який дає змогу привести гетерогенну контейнерну систему у відповідність до єдиного уніфікованого інтерфейсу, що має стандартизований і нормалізований формат, який може використовуватися зовнішнім оточенням;

- патерн Ambassador, що відповідає за приховування складності та надає уніфікований інтерфейс для доступу до служб поза межами поду.

Для підвищення ефективності процесів управління контейнеризованими сервісами і додатками у кваліфікаційній роботі наведена методика реалізації основних стратегій розгортання мікросервісів в системі Kubernetes. Зазначено, що вибір правильної стратегії є важливим рішенням, яке дозволить оцінити процеси розгортання контейнеризованих додатків з точки зору впливу змін, як на саму систему оркестрації Kubernetes, так і на кінцевих користувачів. Наголошується, що найчастіше вибір стратегії обмежується або виділеним часом на реалізацію деплойменту, або бюджетом на інфраструктуру, що відноситься до найчастіших проблем, які обмежують можливості прискорення процесу розгортання системи. В цьому аспекті були розглянуті кілька основних типів стратегій розгортання систем, зокрема: rolling update (поступовий деплоймент), recreate (повторне створення), blue/green (синьо-зелений деплоймент) [23, 25].

ПЕРЕЛІК ДЖЕРЕЛ ПОСИЛАННЯ

1. Бондаренко С. Серверный дирижер: что такое Kubernetes и как он работает [Електронний ресурс] / Сергей Бондаренко. – Доступ здійснено 18.03.2024. – Режим доступу до ресурсу: https://robotdreams.cc/blog/397-serverniy-dirigent-shcho-take-kubernetes-i-yak-vin-pracyuye.

2. Савельев Алексей Решения Microsoft для виртуализации ИТ-инфраструктуры предприятий [Електронний ресурс] / Алексей Савельев // Учебный Internet-курс. – 2012. – Режим доступу до ресурсу: https://intuit.ru/studies/courses/2324/624/info.

3. Лобанов А.А. Облачные вычисления как развитие информационного сервиса [Електронний ресурс] / А.А Лобанов // Перспективы науки и образования. – №2. – 2014. – С. 40 – 45. http://cyberleninka.ru/ article/n/oblachnye-vychisleniya-kak-razvitie-informatsionnogo-servisa.

4. Приходько О.С. Аналіз принципів контейнерного управління на базі системи Kubernetes / О.С. Приходько // 28-й Міжнародний молодіжний форум «Радіоелектроніка та молодь у XXI столітті». Зб. матеріалів форуму. Т.4. Харків: ХНУРЕ. 2024. – С. 148 - 149.

5. What is Cloud Native? [Електронний ресурс] // Amazon Web Services, Inc. – Доступ здійснено 18.03.2024. – Режим доступу до ресурсу: https://aws.amazon.com/ru/what-is/cloud-native/.

6. Kubernetes Documentation: Overview of Kubernetes [Електронний ресурс] // KubeCon + CloudNativeCon Europe 2024. – Режим доступу до ресурсу: https://kubernetes.io/docs/concepts/overview/.

7. Що таке Kubernetes? [Електронний ресурс] // The Linux Foundation. – 2023. – Режим доступу до ресурсу: https://kubernetes.io/uk/docs/concepts/overview/what-is-kubernetes/.

8. Kubernetes Components [Електронний ресурс] // The Linux Foundation. – 30.01.2024. – Режим доступу до ресурсу: https://kubernetes.io/docs/concepts/overview/ components/.

9. Различия между Docker, containerd, CRI-O и runc [Електронний ресурс] // Habr: блог компании Домклик, DevOps, Kubernetes. – 2021. – Режим доступу до

10. Docker, containerd и CRI-O: углубленное сравнение [Електронний ресурс] // $ information Security Squad. – 2022. – Режим доступу: https://itsecforu.ru/2022/03/16/🐋-docker-containerd-и-cri-o-углубленное-сравнение/

11. Створення кластера [Електронний ресурс] / The Linux Foundation. – 2020. Режим доступу до ресурсу: https://kubernetes.io/uk/docs/tutorials/kubernetes-basics/create-cluster/.

12. Rinor Maloku Руководство по Kubernetes, часть 2: создание кластера и работа с ним [Електронний ресурс] / Maloku Rinor // Habr: Блог компании RUVDS.com. – 2019. – Режим доступу до ресурсу: https://habr.com/ru/companies/ruvds/articles/438984/.

13. Марко Лукша Kubernetes в действии /Лукша Марко. – М.: ДМК Пресс, 2019. – 672 с.

14. Так что же такое pod в Kubernetes? [Електронний ресурс] / Andrew Chen, Dominik Tornow // Habr: блог компании Флант, DevOps, Kubernetes. – 2018. – Режим доступу до ресурсу: https://habr.com/ru/companies/flant/articles/427819/

15. Cetin Erhan Part 5 – Replication Controller in Kubernetes [Електронний ресурс] / Erhan Cetin. – 2020. – Режим доступу до ресурсу: https://berhancetin.medium.com/part5-replication-controller-in-kubernetes-78ffeb9133a3.

16. Билджин Ибрам, Роланд Хасс Паттерны Kubernetes: Шаблоны разработки собственных облачных приложений. – СПб.: Питер, 2020. – 391 с.

17. Jobs [Електронний ресурс] // The Linux Foundation. – 2024. – Режим доступу до ресурсу: https://kubernetes.io/docs/concepts/workloads/controllers/job/.

18. CronJob [Електронний ресурс] // The Linux Foundation. – 2024. – Режим доступу до ресурсу: https://kubernetes.io/ru/docs/concepts/workloads/controllers/cron-jobs/.

19. Using kubectl to Create a Deployment [Електронний ресурс] // The Linux Foundation. – 2024. – Режим доступу до ресурсу: https://kubernetes.io/docs/tutorials/kubernetes-basics/deploy-app/deploy-intro/.

20. Sidecar [Електронний ресурс] // k8spatterns / examples. – 2019. – Режим доступу до ресурсу: https://github.com/k8spatterns/examples/tree/main/ structural/Sidecar.

21. Bachina Bhargav Kubernetes – Learn Adapter Container Pattern [Електронний ресурс] / Bhargav Bachina // Medium. – Режим доступу до ресурсу: https://medium.com/bb-tutorials-and-thoughts/kubernetes-learn-adaptor-container-pattern-97674285983c.

22. Bachina Bhargav Kubernetes – Learn Ambassador Container Pattern [Електронний ресурс] / Bhargav Bachina // Medium. – Режим доступу до ресурсу: https://medium.com/bb-tutorials-and-thoughts/kubernetes-learn-ambassador-container-pattern-bc2e1331bd3a.

23. Tremel Etienne Kubernetes deployment strategies [Електронний ресурс] / Etienne Tremel // Container Solutions. – 2017. – Режим доступу до ресурсу: https://blog.container-solutions.com/kubernetes-deployment-strategies.

24. Гаманенко Д. DevOps: Реализуем итеративный подход к внедрению смешанной стратегии непрерывного развертывания [Електронний ресурс] / Дмитрий Гаманенко // DOU.ua. – 2021. – Режим доступу до ресурсу: https://dou.ua/forums/topic/35565/.

25. Buehrle A. Стратегии деплоя в Kubernetes: rolling, recreate, blue/green, canary, dark (A/B-тестирование) [Електронний ресурс]/ Anita Buehrle // Habr: блог компании Флант, DevOps, Kubernetes. – 2019. – Режим доступу до ресурсу: https://habr.com/ru/companies/flant/articles/471620/.