Міністерство освіти і науки України
Харківський національний університет радіоелектроніки

Факультет комп'ютерних наук
(повна назва)

Кафедра програмної інженерії
(повна назва)

КВАЛІФІКАЦІЙНА РОБОТА
Пояснювальна записка

рівень вищої освіти другий (магістерський)

Дослідження методів оркестрування та масштабування контейнеризованих систем у хмарних середовищах
(тема)

Виконав:
студент (ка) 2 курсу, групи ІПЗм-22-5

Павленко В.О.
(прізвище, ініціали)

Спеціальність 121 – Інженерія програмного забезпечення
(код і повна назва спеціальності)
Тип програми освітньо-наукова

Керівник доц. Хацько Н.Є.
(посада, прізвище, ініціали)

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

2024 р.

Харківський національний університет радіоелектроніки

Факультет комп’ютерних наук
Кафедра програмної інженерії
Рівень вищої освіти другий (магістерський)
Спеціальність 121 – Інженерія програмного забезпечення
Тип програми освітньо-наукова програма
Освітня програма Інженерія програмного забезпечення
(шифр і назва)

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

ЗАВДАННЯ
НА КВАЛІФІКАЦІЙНУ РОБОТУ

студентові Павленку Владиславу Олександровичу
(прізвище, ім’я, по батькові)

1. Тема роботи «Дослідження методів оркестрування та масштабування розподілених контейнеризованих систем у хмарних середовищах»
Затверджена наказом по університету від 16.01. 2024р. № 35 Ст
2. Термін подання студентом роботи до екзаменаційної комісії 18.06.2022
3. Вихідні дані до роботи дослідження методів та застосунків оркестрації розподілених систем, методи контейнеризації застосунків, процес розгортання контейнеризованих застосунків, горизонтальне масштабування контейнеризованих додатків на основі базових та сервісних метрик, технологія шаблонізації ресурсів Helm
4. Перелік питань, що потрібно опрацювати в роботі
аналіз процесів масштабування окремих додатків в оркестрованій розподіленій системі, побудова оркестрованої розподіленої системи, масштабованої на базі ресурсних та сервісних метрик, створення універсального шаблону розгортання та компоненту автоматичного масштабування

КАЛЕНДАРНИЙ ПЛАН

№	Назва етапів роботи	Термін виконання етапів роботи	Примітка
1	Аналіз предметної галузі та постановка задачі	16.01 – 01.02.24	*виконано*
2	Аналіз та моделювання предметної області	02.02 – 08.02.24	*виконано*
3	Аналіз технологій оркестрування та контейнеризації додатків	09.02 – 16.02.24	*виконано*
4	Дослідження та порівняння існуючих методів	17.02 – 28.02.24	*виконано*
5	Програмна реалізація компоненту автоматичного масштабування	29.02 – 03.04.24	*виконано*
6	Експериментальні дослідження	04.04 – 08.04.24	*виконано*
7	Аналіз результатів експериментальних досліджень	09.04 – 10.04.24	*виконано*
8	Написання та оформлення тез доповіді	11.04 – 13.04.24	*виконано*
9	Підготовка пояснювальної записки	14.04 – 05.06.24	*виконано*
10	Підготовка презентації та доповіді	06.06 – 07.06.24	*виконано*
11	Нормоконтроль	08.06 – 10.06.24	*виконано*
12	Рецензування	11.06 – 14.06.24	*виконано*
13	Занесення диплома в електронний архів	15.06.2024	*виконано*
14	Попередній захист	15.06.2024	*виконано*
15	Допуск до захисту у зав. кафедри	18.06.2024	*виконано*

Дата видачі завдання 22 січня 2024р.

Студент (ка) _____ Павленко В.О.
(підпис)

Керівник роботи _____ доц. Хацько Н.Є.
(підпис) (посада, прізвище, ініціали)

РЕФЕРАТ / ABSTRACT

Пояснювальна записка містить: 104 с., 37 рис., 1 табл., 78 джерел.

КОНТЕЙНЕР, КОНТЕЙНЕРИЗАЦІЯ, МАСШТАБУВАННЯ, МЕТРИКИ, ОРКЕСТРАЦІЯ, DOCKER, HPA, INFRASTRUCTURE-AS-A-SERVICE, KUBERNETES, PROMETHEUS, PROMETHEUS ADAPTER.

Об’єкт дослідження – методи оркестрування та масштабування розподілених контейнеризованих систем у хмарних середовищах.

Мета роботи – дослідження існуючих технологій та моделей оркестрування та масштабування розподілених хмарних інформаційних систем, аналіз методів оркестрування та налаштування автоматичного масштабування комплексних мікросервісних систем.

В результаті будуть знайдені варіанти застосування існуючих технологій для побудови оптимальної масштабованої архітектури, реалізовано компонент автоматичного масштабування та універсальний шаблон розгортання додатків.

CONTAINER, CONTAINERIZATION, ORCHESTRATION, SCALING, METRICS, DOCKER, KUBERNETES, PROMETHEUS, PROMETHEUS ADAPTER, HPA, INFRASTRUCTURE-AS-A-SERVICE.

The object of work is methods of orchestration and scaling of distributed contrainerised systems in cloud environments.

The purpose of the work is to study existing technologies and models for orchestration and scaling of distributed cloud information systems, analyse methods of orchestration and set up automatic scaling of complex microservice systems.

As a result, options for applying existing technologies to build an optimal scalable architecture will be found, an automatic scaling component and a universal application deployment template will be implemented.

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

Я, Павленко Владислав Олександрович, студент(ка) гр. ІПЗм-22-5, здобувач вищої освіти на другому (магістерському) рівні кафедри «Програмна інженерія», заявляю: моя кваліфікаційна робота на тему «Дослідження методів оркестрування та масштабування розподілених систем в хмарних середовищах», що буде представлена в екзаменаційну комісію для публічного захисту, виконана самостійно, в ній не містяться елементи плагіату і вона може бути опублікована в електронному архіві відкритого доступу ElArKhNURE. Всі запозичення з друкованих та електронних джерел мають відповідні посилання.

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

ЗМІСТ

Вступ 8
1 Аналіз предметної галузі 9
1.1 Контейнеризація програмних застосунків 9
1.2 Оркестрування контейнерів 11
1.3 Розподілені системи та їхні особливості в хмарних середовищах 14
1.4 Актуальність дослідження 16
2 Методи оркестрування та масштабування контейнерів 18
2.1 Мета дослідження 18
2.1.1 Додаток для проведення дослідження 18
2.2 Контейнеризація додатків за допомогою Docker 18
2.3 Розгортання контейнеризованих додатків в кластері Kubernetes 20
2.3.1 Вибір обчислювального вузла для розгортання репліки в Kubernetes 22
2.3.2 Менеджери пакетів як альтернативний метод розгортання у Kubernetes 25
2.4 Конфігурація маршрутизації запитів в Kubernetes 28
2.5 Перевірка стану реплік, механізм Helm Check в Kubernetes 30
2.6 Стратегії оновлення реплік в Kubernetes 32
2.7 Автоматичне масштабування контейнеризованих додатків 35
2.8 Традиційний метод масштабування в Kubernetes, ресурсні метрики 37
2.8.1 Отримання та збір ресурсних метрик кластера Kubernetes 38
2.8.2 Алгоритм роботи Horizontal Pod Autoscaler 39
2.8.3 Переваги та недоліки методу 40
2.9 Метод масштабування в Kubernetes на базі Prometheus, сервісні метрики ... 41
2.9.1 Користувацький ресурс ServiceMonitor 43
2.9.2 Збір ресурсних метрик Kubernetes за допомогою Prometheus 45
2.9.3 Збір сервісних метрик розгорнутих додатків за допомогою Prometheus . 46
2.9.4 Інтеграція метрик Prometheus з Kubernetes HPA 48
2.9.5 Переваги та недоліки методу 50
3 Розробка компоненту автоматичного масштабування 52
3.1 Мета розробки 52

3.2 Постановка задачі 53
3.3 Архітектура компоненту автоматичного масштабування 54
3.4 Конфігурування параметрів масштабування для груп метрик 56
3.5 Авторизація в Kubernetes API для взаємодії з ресурсами кластеру 57
3.6 Реалізація модулю пошуку цільових груп реплік 58
3.7 Реалізація модулю моніторингу 60
3.8 Реалізація модулю розрахунку необхідної кількості реплік 62
3.9 Реалізація модулю прийняття рішень масштабування 63
3.10 Універсальний шаблон розгортання контейнеризованих додатків 64
4 Опис експериментальних досліджень 66
4.1 Мета дослідження 66
4.1.1 Огляд існуючих досліджень методів масштабування в Kubernetes 66
4.2 Постановка експерименту 67
4.3 Проведення експериментальних досліджень 69
4.3.1 Традиційний метод з використанням HPA та Metrics Server 69
4.3.2 Традиційний метод з використанням HPA та Prometheus 70
4.3.3 Розроблений компонент масштабування на базі метрик Prometheus 72
4.4 Результати дослідження 73
Висновки 76
Перелік джерел посилання 78
Додаток А Перелік джерел посилання за науковими напрямками кервіника та науковців кафедри програмної інженерії ..... **Помилка! Закладку не визначено.**
Додаток Б Звіт результатів перевірки на унікальність тексту в базі ХНУРЕ ..... **Помилка! Закладку не визначено.**
Додаток В Слайди презентації ..... **Помилка! Закладку не визначено.**
Додаток Г Апробація результатів роботи ..... **Помилка! Закладку не визначено.**
Додаток Д Експертний висновок результатів перевірки кваліфікаційної роботи на відповідність оформлення вимогам ДСТУ 3008: 2015 104

ВСТУП

Технологічний прогрес почав еру інновацій, докорінно змінивши основу нашого існування. Динаміка розвитку інформаційних систем стала головною силою, що рухає сучасний світ вперед, де ефективність, адаптивність і масштабованість вийшли на перший план.

Хмарні технології є основою цього розвитку у наші дні, не лише революціонізуючи методології зберігання та обробки даних, але й докорінно змінюючи способи ведення бізнесу та міжособистісну динаміку, за допомогою якої люди взаємодіють з інформацією. Перехід до хмарних технологій означає відхід від традиційної локальної інфраструктури до динамічних, масштабованих і віддалено доступних обчислювальних ресурсів.

Відповідно до зростання складності та розміру інформаційних систем, потреба в ефективному оркеструванні ресурсів стає важливою. Оркестрування визначає методи та процеси, які дозволяють автоматично управляти життєвим циклом програмних застосунків у розподілених системах, зокрема автоматизацію розгортання, масштабування і відновлення додатків, що є критично важливим для підтримки високої доступності та надійності.

Кваліфікаційна робота включає комплексне теоретичне та практичне дослідження існуючих методів та практик, що застосовуються для оркестрування та масштабування розподілених систем. Результатом проведених досліджень є універсальний шаблон розгортання додатків в оркестрованих середовищах та компонент автоматичного масштабування, метою створення яких є спрощення інтеграції традиційних підходів з забезпеченням надійності та стабільності.

Таким чином, ця кваліфікаційна робота не тільки підкреслює важливість глибокого розуміння контейнеризації та оркестрування, але й вносить значний вклад у розвиток методів, що дозволяють більш ефективно використовувати можливості розподілених систем. Надалі, результати цього дослідження можуть слугувати основою для подальших наукових робіт і розробок у галузі оркестрації та автоматичного масштабування.

1 АНАЛІЗ ПРЕДМЕТНОЇ ГАЛУЗІ

1.1 Контейнеризація програмних застосунків

Контейнеризація [1] – це метод, який використовується для пакування програми разом з її залежностями та конфігураційними файлами в стандартизовану одиницю, яка називається контейнер [2]. Це дозволяє програмі (контейнеру) виконуватися в різних обчислювальних середовищах, наприклад в локальних середовищах розробки та тестування, бути розгорнутими на віддаленому сервері або у хмарі, не турбуючись про відмінності в базовій інфраструктурі.

Контейнери – це ізольовані середовища, які містять усе необхідне для запуску програми, зокрема бібліотеки, скрипти та інші компоненти, які будуть необхідні під час виконання. Головною перевагою контейнерів є ізоляція процесів та ресурсів, що дозволяє декільком додаткам працювати незалежно на одному сервері з власною файловою системою, мережею та простором процесів.

Ключовим компонентом в архітектурі контейнеризації є рушій [3]. Рушій контейнерів відповідальний за створення, управління та виконання контейнеризованих додатків, напряму взаємодіючі з ядром операційної системи серверу для ізоляції контейнерів та управляння їхніми ресурсами.

Ізоляція контейнерів рушієм досягається за допомогою використання простору імен операційної системи [4]. Розглянемо простори імен операційної системи Linux, які використовуються для ізоляції:

- простір імен PID [5] ізолює ідентифікатори запущених процесів, гарантуючи, що процеси у різних контейнерах не взаємодіють один з одним;
- простір імен NET [6] надає ізольовані мережеві інтерфейси, IP-адреси та таблиці маршрутизації для кожного контейнера;
- простір імен MNT [7] ізолює точки для монтування файлової системи, дозволяючи кожному контейнеру мати власну кореневу файлову систему;

- простір імен IPC [8] ізолює ресурси міжпроцесорної взаємодії, наприклад спільну пам’ять;
- простір імен користувачів операційної системи [9] ізолює ідентифікатори користувачів, дозволяючи запускати контейнери з різними рівнями доступу.

Для розподілення, обліку та ізоляції системних ресурсів запущених контейнерів, таких як використання центрального процесору, пам’яті або мережі, рушій контейнерів використовує контрольні групи операційної системи [10].

Контрольна група ЦП [11] (CPU cgroup) обмежує і визначає пріоритети використання процесору, контролюючи розподіл процесорного часу між процесами. Кожен підконтрольний процес всередині CPU cgroup має відносну вагу, що дозволяє розставити пріоритети використання процесорного часу. Наприклад, якщо контейнер А має відносну вагу 1024, а контейнер В – 512, то контейнер А отримає вдвічі більше процесорного часу, ніж контейнер В.

Контрольна група оперативної пам’яті [12] (Memory cgroup) відповідно керує розподілом та ізоляцією ресурсів пам’яті, зокрема оперативної пам’яті (ОЗП) та простором підкачки. Кожен контейнер має ліміт використання пам’яті, тому у випадку, якщо об’єм використання пам’яті поза межами наданого під час запуску контейнеру ліміту, процес буде завершеним за допомогою OOM Killer (Out-Of-Memory Killer) [13], при цьому кожен процес може мати власну поведінку зупинки утилітою OOM Killer.

Docker [14] є найпопулярнішою реалізацією рушія контейнерів та стандартом індустрії, надаючи повний набір інструментів і команд для створення, запуску та керування контейнерами. Тож на прикладі Docker розглянемо архітектуру та особливості компонентної бази сучасного рушія контейнерів.

Основним компонентом Docker є фоновий процес Docker Daemon [15] або dockerd, який відповідає за управління сутностями Docker – контейнерами, образами, мережами та томами.

Управління здійснюється за допомогою клієнту Docker CLI [16] (інтерфейс командного рядка), який здійснює запити до Docker Daemon через REST API.

Для створення контейнерів певного додатку використовуються Docker Images [17] (образи Docker), які представляють read-only (незмінні) шаблони, які містять набір інструкцій для збирання образу.

Зберігання та розповсюдження образів Docker для створення контейнерів здійснюється сервісом Docker Registry [18] (реєстр Docker).

Загальна схема взаємодії описаних вище компонентів Docker зображена на рисунку 1.1. Взаємодія компонентів розглянута на прикладі основних команд клієнту Docker.

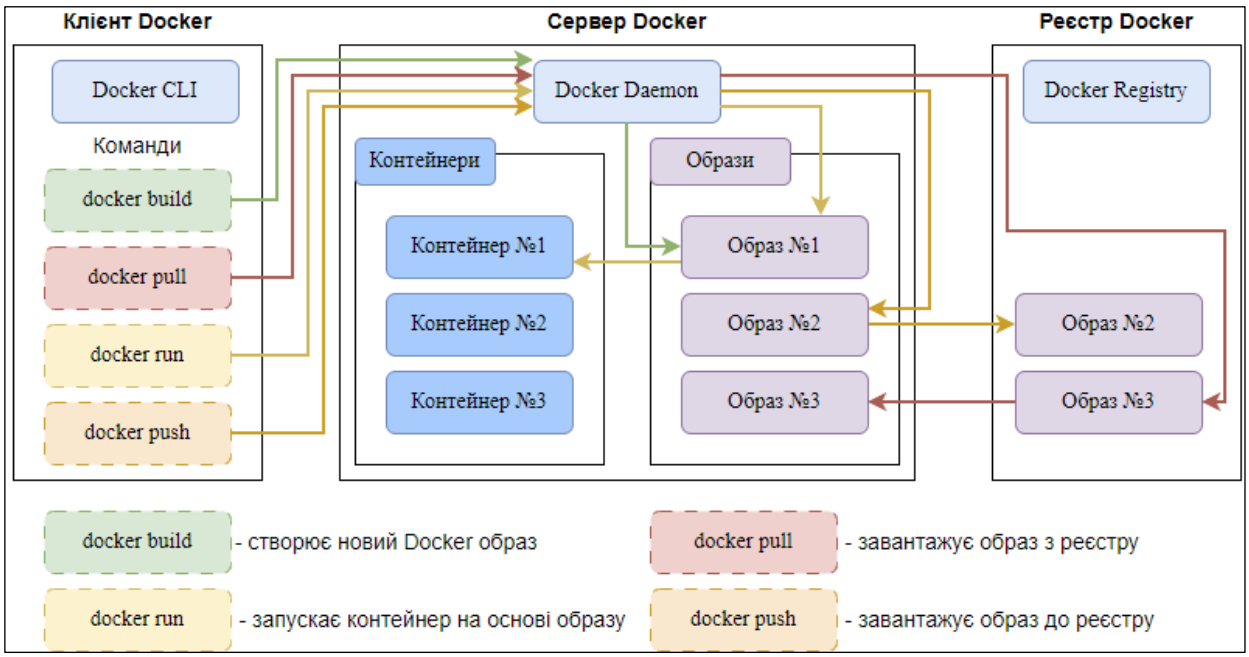

Рисунок 1.1 – Взаємодія компонентів Docker на прикладі команд Docker CLI

Існує декілька альтернатив рушію контейнеризації Docker, такі як Rocket [19] та Singularity [20], проте вони не будуть розглядатися в кваліфікаційній роботі, так як Docker де-факто є стандартом, маючи більший спектр застосувань, ширший набір інструментів та детальну документацію.

1.2 Оркестрування контейнерів

Великомасштабні розподілені системи, наприклад, системи побудовані з використання мікросервісної архітектури [21], потребують розгортання великої

кількості контейнерів щоденно та управління ними всіма одночасно, що робить незручним та неефективним розгортання кожного контейнера окремо, а в подальшому моніторинг їхнього стану.

Задля вирішення цієї проблеми, було розроблено оркестратори контейнерів, що надають автоматизоване управління контейнерними додатками. Оркестрування передбачає координацію різних завдань, пов'язаних з управлінням життєвим циклом контейнерів, розподілом ресурсів, мережею та виявленням сервісів, щоб забезпечити ефективну, надійну та масштабовану роботу контейнерних додатків. Платформи оркестрування контейнерів абстрагуються від складності управління окремими контейнерами і надають високорівневі інструменти та API для керування станом групи контейнерів одного додатку (реплік), вузлами та кластерами.

До базових абстракцій оркестратору контейнерів, які спрощують керування великомасштабних розподілених систем, відносять наступні (див. рис. 1.2).

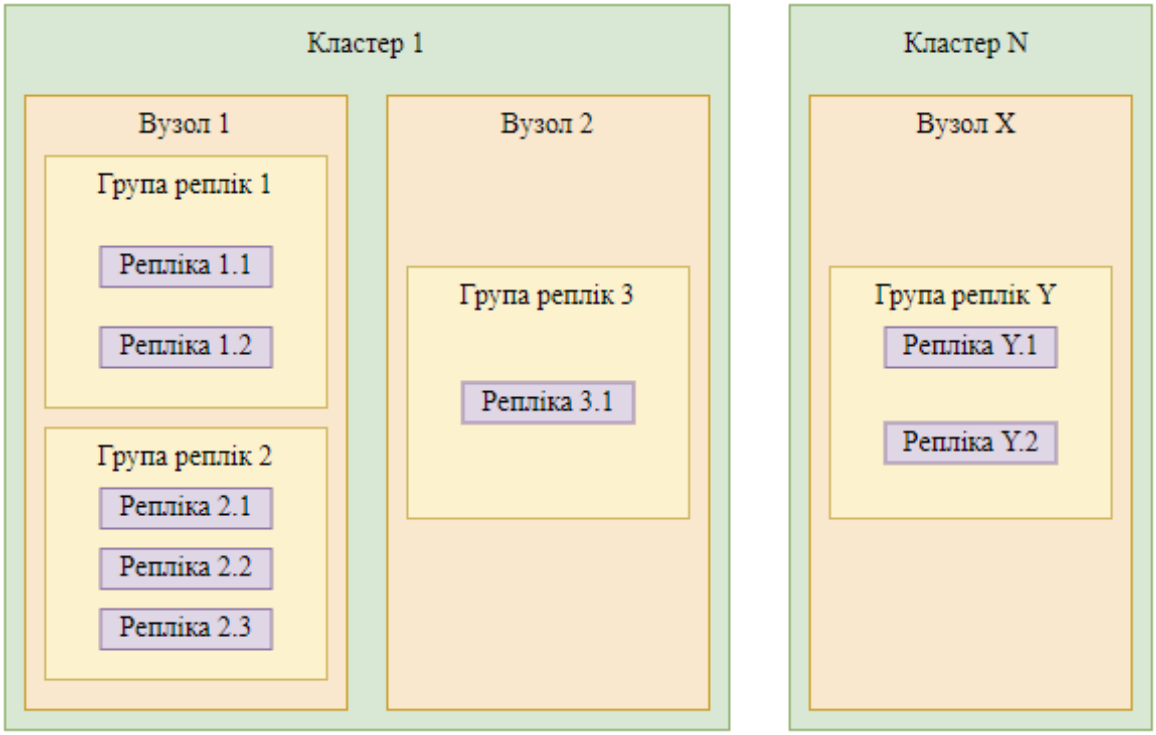

Рисунок 1.2 – Устрій базових абстракції оркестратору контейнерів

Абстракції упорядковано від найменшого рівня оркестрування до найбільшого:

– репліка – найменша базова одиниця, яка може бути розгорнута оркестратором, запакований та запущений контейнеризований додаток з усіма необхідними залежностями;
– група реплік – абстракція вищого рівня за репліку, яка керує розгортанням і масштабування контейнерів одного додатку, підтримуючи стабільну кількість реплік та відмовостійкість;
– вузол – фундаментальний компонент, який представляє один фізичний сервер або віртуальну машину, забезпечуючи обчислювальні ресурси, необхідні для розгортання контейнерів;
– кластер – набір вузлів, на яких виконуються контейнеризовані додатки.

Згідно дослідженню «A Comprehensive Feature Comparison Study of Open-Source Container Orchestration Frameworks» [22] (переклад «Комплексне дослідження порівняння можливостей фреймворків оркестрування контейнерів з відкритим вихідним кодом»), яке проводили з метою якісного та кількісного порівняння найпопулярніших оркестраторів контейнеризованих розподілених систем з точки зору універсальності, визначення їхніх спільних та унікальних рис та дослідження зрілості інструментів, оркестратор контейнерів Kubernetes [23] підтримує найбільшу кількість функціоналу для всіх 9 функціональних аспектів та 27 підаспектів, за якими проводилося порівняння (див. рис. 1.3).

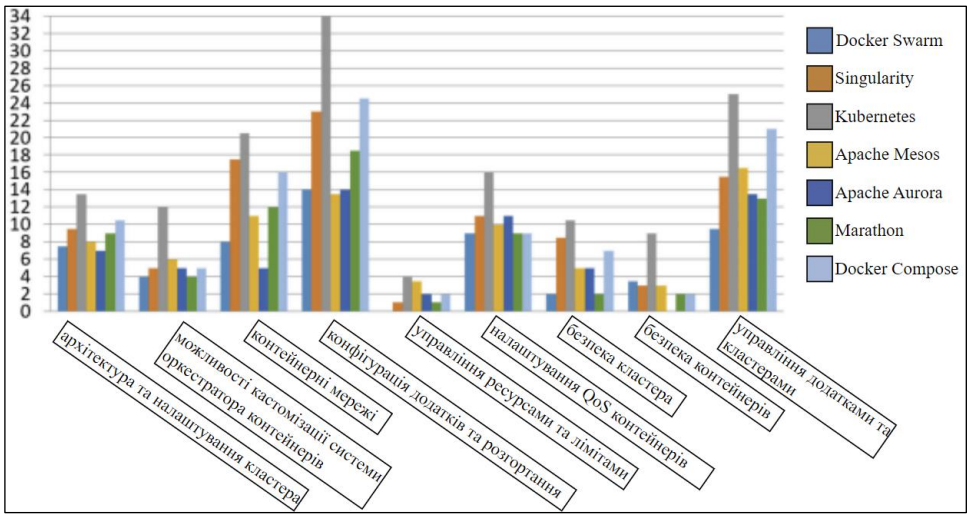

Рисунок 1.3 – Результат порівняння оркестраторів контейнерів за дев’ятьма ключовими аспектами

Отже, обґрунтованим вибором для проведення досліджень в області оркестрації контейнеризованих розподілених систем в цій кваліфікаційний роботі є оркестратор Kubernetes.

Kubernetes (часто скорочують як K8s) є системою управління контейнерами або оркестратором контейнерів, розробленим компанією Google. Наразі Kubernetes є частиною CNCF (Cloud Native Computing Foundation) [24] та відіграє ключову роль у їхній екосистемі. Шляхом приєднання Kubernetes до CNCF, оркестратор отримав широку підтримку від індустрії та наразі є її стандартом.

1.3 Розподілені системи та їхні особливості в хмарних середовищах

Розподілена система (Distributed System) представляє мережу слабко пов’язаних незалежних сервісів, де кожен з них працює як автономна одиниця, взаємодіючі з іншими сервісами через чітко визначені API.

Окремим прикладом розподіленої системи є мікросервісна архітектура, де різні функціональні компоненти, такі як каталог продуктів або автентифікація користувачів, є розподіленими автономними одиницями.

Зазвичай, розподілені системи розгортають у публічних хмарах, таких як Amazon AWS [25], Microsoft Azure [26] або Google Cloud Platform [27], так як публічні хмари надають ресурси на вимогу, їхню кількість можна легко збільшити або зменшити відповідно до мінливих вимог системи, це гарантує, що внутрішня інфраструктура розподіленої системи зможе ефективно справлятися з різними навантаженнями. З іншого боку, надання ресурсів на вимогу є економічно доцільною моделлю оплати, так як оплата здійснюється лише за ресурси, які використовуються, це усуває потребу у великих початкових інвестиціях в обчислювальне обладнання.

Всі перечисленні вище публічних хмари є реалізаціями IaaS [28], тобто вони надають повноцінну інфраструктуру для розгортання, а не лише обчислювальні потужності.

Розглянемо ключові послуги та інфраструктурні компоненти IaaS публічних платформ на прикладі Amazon AWS.

IaaS платформи надають віртуальні обчислювальні ресурси для виконання програмних застосунків в ізольованих віртуальних середовищах. Ця послуга є фундаментальною для розгортання різних компонентів розподіленої системи.

Представником цієї послуги в Amazon AWS є сервіс Amazon EC2 [29], який надає віртуальні сервери-екземпляри (EC2 Instance). EC2 пропонує широкий вибір типів екземплярів та їхніх розмірів для різних потреб, кожен з них має певну комбінацію віртуальних процесорів, оперативної пам’яті та пропускної можливості мережі. Наприклад, сервер-екземпляр типу m5 з розміром xlarge має 4 віртуальних процесори, 16 Гігабайт оперативної пам’яті та пропускну можливість мережі швидкістю до 10 Гігабіт на секунду [30].

Іншою важливою послугою IaaS є служби для створення та керування мережами, підмережами, балансувальними навантаження та брандмауерами. Ці послуги також є фундаментальними, бо за їхньою допомогою створюється підключення розподілених компонентів та балансування трафіку.

Для побудування мережі для комунікації серверів-екземплярів EC2 у Amazon AWS є окремий сервіс Amazon VPC [31] (Virtual Private Cloud, віртуальна приватна хмара). За її допомогою створюються ізольовані розділи хмари AWS, які пов’язують сервери-екземпляри EC2 у віртуальній мережі або підмережі, в результаті вони можуть комунікувати один з одним, використовуючи таблиці маршрутів і безпекові групи для контролю трафіку між ними.

Абсолютна більшість сучасних розподілених систем використовують реляційні бази даних, тому IaaS платформи надають служби для автоматизованого розгортання систем управління базами даних (СУБД), таких як PostgreSQL, Oracle DB, MySQL тощо. Послуга з надання СУБД має власну назву DBaaS.

Amazon AWS пропонує сервіс Amazon RDS (Relation Database Service, сервіс реляційних баз даних) для спрощення процесів налаштування, розгортання та масштабування реляційних баз даних у хмарі. Сервіс підтримує найпопулярніші СУБД, може виконувати регулярні автоматичні бекапи згідно з встановленою конфігурацією та забезпечує моніторинг продуктивності СУБД за допомогою утиліти Amazon CloudWatch [32], дозволяючи адміністратором баз

даних відстежувати основні показники у реальному часі та налаштовувати сигнали тривоги для повідомлення про критичні ситуації.

Розгортання всієї інфраструктури розподіленої системи на єдиній хмарній IaaS платформі надає значні переваги порівняно з розгортанням з використанням двох або більше хмар.

Головною перевагою є фізичне розміщення обчислювальних ресурсів в одному дата-центрі, що значно зменшує затримки передачі даних між різними серверами та базами даних, бо мінімізація затримок мережі є надкритичною для розподілених інформаційних систем.

Як приклад допустимо, що ми маємо розгорнуту розподілену систему в Amazon EC2 на сервері-екземплярі типу m5.xlarge, тобто міжсерверна комунікація у мережі здійснюється зі швидкістю 10 Гігабіт на секунду, та дві бази даних: перша розгорнута у Amazon RDS, друга в Microsoft Azure. Доступ до даних в базі даних, яка розгорнута в Amazon, буде здійснюватися зі швидкістю до 10 Гігабіт на секунду, тому що географічно вони знаходяться в одному дата-центрі і фізична відстань між обладнанням є оптимізованою для отримання заявленої швидкості. В той час як доступ до бази, яка розгорнута в іншій хмарі, потенційно буде набагато повільнішим, бо міжхмарний зв’язок передбачає передачу даних через Інтернет, що може залучати кількох мережевих провайдерів для передачі запиту та отримання відповіді [33].

1.4 Актуальність дослідження

Актуальність цього дослідження зумовлена швидким розвитком хмарних технологій та необхідністю в підвищенні ефективності управління контейнеризованими додатками в розподілених середовищах.

В роботі досліджуються сучасні підходи та методики, які дозволяють раціонально використовувати ресурси кластера, зокрема за допомогою різних стратегій масштабування заснованих на ресурсних та сервісних метриках додатків.

Інноваційність дослідження полягає в розробці універсального шаблону для розгортання контейнеризованих додатків в кластері Kubernetes на базі технології Helm та компоненту автоматичного масштабування з повною інтеграцією з інфраструктурою Prometheus.

В контексті широкого використання мікросервісних архітектур та хмарних технологій, питання оркестрації та масштабування набувають особливої ваги, що робить дане дослідження не тільки актуальний, а й необхідним для подальшого розвитку технологічних практик.

2 МЕТОДИ ОРКЕСТРУВАННЯ ТА МАСШТАБУВАННЯ КОНТЕЙНЕРІВ

2.1 Мета дослідження

Метою цього параграфу є дослідження базових компонентів, механізмів та концепцій роботи оркестратора контейнерів Kubernetes та рушія контейнерів Docker, як передумова для побудови власної універсальної моделі розгортання контейнеризованих додатків та створення інтегрованого компоненту автоматично масштабування на базі ресурсних та сервісних метрик.

2.1.1 Додаток для проведення дослідження

Для проведення дослідження оркестрування та масштабування реалізовано базовий додаток на мові програмування Java та фреймворку Spring Boot 3 [34], який може приймати та обробляти HTTP-запити. Далі згадується як «тестовий додаток».

Обґрунтуванням вибору саме мови програмування Java є легкість її пакування за допомогою утиліти автоматизації збірки Gradle [35] в JAR-файл, що спростить процес контейнеризації додатку у наступних дослідженнях.

Обґрунтуванням вибору фреймворку Spring Boot 3 є великий набір вбудованих бібліотек, які необхідні для виконання нефункціональних вимог, а саме налаштування коректної роботи додатку в просторі Kubernetes після його розгортання.

2.2 Контейнеризація додатків за допомогою Docker

Ключовим елементом у контейнеризації додатків у Docker є створення Dockerfile [36], який представляє собою текстовий файл, що містить набір інструкцій необхідних для створення образу контейнера.

Інструкції у Dockerfile – це ключові слова, які застосовують певну дію до образу, наприклад, копіювання файлу з робочого каталогу або виконання команди всередині образу. Наразі, Dockerfile підтримує 18 інструкцій для створення образів контейнеру, тож розглянемо основні з них:

- інструкція «FROM» зазвичай є першим рядком Dockerfile та посилається на існуючий образ з Docker Registry, який стане основою образу, який збирається;
- інструкція «COPY» використовується для додавання файлів та папок до файлової системи образу, копіювання виконується між хостом Docker та образом, який збирається, також ця інструкція дозволяє копіювати файли безпосередньо з інших образів Docker, використовуючи флаг «--from»;
- інструкція «RUN» виконує команди всередині образу та створює новий шар образу поверх попереднього, створений шар містить зміни у файловій системі, які буде застосовано командою;
- інструкція «ENV» використовується для встановлення змінних оточення, які будуть доступні під час виконання контейнеризованого додатку;
- Загалом, набір команд доступних у Dockerfile дозволяє створити образ будь-якого додатку – від мікросервісів, які майже не містять сторонніх залежностей, до багатомодульних монолітних додатків.

Для створення образу за реалізованим Dockerfile, використовується команда Docker CLI «docker build». Під час виконання цієї команди, Docker Daemon ініціалізує процес збірки, зчитуючи інструкції Dockerfile та створюючи контекст для побудови образу. Зазвичай, контекстом є коренева папка додатку, який збирається, та всі файли та підпапки в ньому.

Docker Daemon обробляє інструкції послідовно, створюючи новий шар образу після виконання інструкції. Для пришвидшення процесу збирання Docker використовує механізм кешування. Якщо Docker виявляє, що інструкція у Dockerfile не змінилася, і є кешована версія шару образу, отримана за цією інструкцією, то рушій контейнеризації повторно використає кешований шар.

Останнім кроком Docker Daemon завантажує зібраний образ в локальний Docker Registry для подальшого розгортання чи завантаження в публічні реєстри контейнерів, такі як Docker Hub.

Команда «docker run» створює і запускає новий контейнер із зібраного образу. Під час виконання команди Docker Daemon налаштовує файлову систему,

мережу і оточення контейнеру. Команда підтримує багато опцій, за допомогою яких можна налаштувати порти, змінні середовища, тощо.

На прикладі тестового додатку, реалізуємо та розглянемо Dockerfile для створення його образу.

Dockerfile для створення образу тестового додатку:

```
FROM amazoncorretto:21.0.1
WORKDIR /service
COPY build/libs/msjava-spring-prometheus-sample.jar msjava-spring-
prometheus-sample.jar
CMD ["java", "-jar", "msjava-spring-prometheus-sample.jar"]
```

В першому рядку вказується базовий образ для контейнеризації, а саме дистрибутив OpenJDK від Amazon версії 21.0.1. Базовий образ JVM необхідний для коректного функціонування Java-додатку в контейнері. Далі визначається робоча директорія, в які буде скопійовано JAR-файл додатку. В останньому рядку визначається необхідна команда для запуску контейнера.

Після створення образу контейнеру, додаток готовий до того, щоб бути розгорнутим в кластері оркестратору контейнерів.

2.3 Розгортання контейнеризованих додатків в кластері Kubernetes

Основним інструментом для опису і управління ресурсами в кластері Kubernetes є маніфест [37]. Маніфести описують декларативно у форматі YAML, вони служать інструкціями для Kubernetes про те, як розгортати і керувати компонентами системи, такими як контейнери, сервіси, мережеві політики тощо. Узагальнено, маніфести дозволяють визначити бажаний стан системи, а Kubernetes забезпечує його досягнення.

Кожен маніфест Kubernetes містить базові елементи, які визначають, як і які ресурси повинні бути створенні в кластері:

– елемент «apiVersion» вказує на версію Kubernetes API, яка повинна бути використана для створення ресурсу;

– елемент «kind» описує тип ресурси, що має бути розгорнутим у кластері;
– елемент «metadata» містить метадані про ресурс, такі як ім’я, простір імен, мітки та анотації, які допомагають ідентифікувати та організувати ресурси;
– елемент «spec» описує детальну специфікацію ресурсу, що включає всі конфігураційні параметри, необхідні для його створення та роботи.

Управління ресурсами в Kubernetes відбувається за допомогою інтерфейсу командного рядка kubectl [38], що підтримує широкий набір команд. Для розгортання нового або оновлення вже створеного ресурсу в Kubernetes використовується команда «kubectl apply».

Для розгортання одного або кількох тісно пов’язаних образів контейнеризованих додатків використовується ресурс Pod [39] (далі репліка), що представляє базову одиницю і фундаментальний будівельних блок в Kubernetes. Репліка не може бути напряму описана та розгорнута за допомогою маніфесту Kubernetes з елементом «kind» зі значенням Pod, натомість для її розгортання в кластері використовується маніфест ресурсу Deployment.

Deployment [40] є вищим рівнем абстракції над репліками, що забезпечує розгортання, оновлення та управління групою реплік. Маніфест ресурсу Deployment надає Kubernetes ключову інформацію про групу реплік, яка має бути розгорнутою, а саме: образ контейнера, який має використовуватися під час розгортання репліки, ресурсні вимоги, політику оновлення та кількість реплік.

Детально розглянемо специфікацію маніфесту ресурсу Deployment на прикладі реалізації цього маніфесту для тестового додатку.

Маніфест Deployment для розгортання тестового додатку:

```
apiVersion: apps/v1
kind: Deployment
metadata:
  ...
spec:
  replicas: 2
  selector:
```

```
    matchLabels:
      app: diploma-sample-java-app
  template:
    metadata:
      labels:
        app: diploma-sample-java-app
    spec:
      containers:
      - name: diploma-sample-java-app
        image: diploma-sample-java-app:latest
        resources:
          requests:
            memory: "256Mi"
            cpu: "250m"
        ports:
        - containerPort: 8080
```

Після застосування реалізованого маніфесту в інтерфейсі командного рядка kubectl, Kubernetes автоматично розгорне 2 репліки (відповідно до елементу конфігурації «spec.replicas»), на яких буде запущено по одному контейнеру з ідентифікатором «diploma-sample-java-app:latest» (відповідно до елементу конфігурації «spec.template.spec.image»). Контейнери розгорнуті на репліках гарантовано матимуть по 256МБ оперативної пам’ять та 250 міліядер процесорного часу (відповідно до елементів «spec.template.spec.resources.requests.memory» та «spec.template.spec.resources.requests.cpu»). 250 міліядер процесорного часу означають доступ до чверті ядра процесора, одно ядро еквівалентне 1000 міліядер.

При розгортанні ресурсу Deployment, Kubernetes також автоматично створює ресурс ReplicaSet [41], який відповідає за підтримку заданої у маніфесті кількості реплік. В свою чергу, ресурс ReplicaSet розгортає репліки згідно ресурсним вимогам та політикою оновлення.

2.3.1 Вибір обчислювального вузла для розгортання репліки в Kubernetes

Кластер Kubernetes, як і інших оркестраторів контейнерів, складається з багатьох обчислювальних вузлів, що представляють фізичні сервери чи віртуальні машини.

За вибір обчислювального вузла в кластері, на якому буде розгорнута репліка, відповідає компонент Scheduler [42] (далі планувальник). Процес вибору вузла складається з двох послідовних фаз: фаза фільтрації та фаза ранжування.

Основною метою фази фільтрації є відсіювання вузлів, які не відповідають базовим вимогам репліки, яка розгортається.

Першим кроком фази фільтрації є перевірка кількості вільних ресурсів кожного вузла (завантаженість центрального процесору, пам’ять, мережеві ресурси тощо), щоб задовільнити вимоги репліки, яка розгортається.

Другим кроком є перевірка політик розгортання репліки на певних вузлах, ці політики в Kubernetes мають назву Node Affinity та Node Anti-affinity [43].

Політика Node Affinity дозволяє вказати характеристики, на основі яких будуть визначатися вузли, на яких має розгортатися репліка (наприклад, репліка може бути розгорнута лише на вузлах, які мають певний тип обладнання). Політика Node Anti-Affinity навпаки використовуються щоб запобігти розгортанню реплік на певних вузлах.

Третім кроком є перевірка політик розгортання реплік на вузлах відносно інших реплік розгорнутих по всьому кластеру, політики мають назву Pod Affinity та Pod Anti-affinity [44].

Політика Pod Affinity дозволяє згрупувати репліки на одних вузлах, наприклад дві репліки, які часто взаємодіють між собою і потребують низької затримки підчас міжсервісної взаємодії, можуть бути розгорнуті на одному вузлі.

Політика Pod Anti-Affinity навпаки запобігає розміщенню реплік одного типу на одних і тих самих вузлах, що може бути корисно для зменшення ризику втрати даних у випадку відмови одного з вузлів, наприклад під час розгортання баз даних.

Четвертим кроком є механізм Taints та Tolerations [45]. Вузли мають Taints, які запобігають розгортанню реплік, що не мають відповідних Tolerations.

Taints представляють набір міток, який призначається вузлам. Мітки складаються з трьох компонентів – ключ, значення та ефект, який визначає набір

дій, які мають відбуватися з реплікою, яка не толерує цей Taint. До основних ефектів Taint відносять наступні:

- ефект «NoSchedule» визначає набір дій, при якому Scheduler не буде розміщувати нові репліки на вузлі;
- ефект «PreferNoSchedule» визначає набір дій, при якому Scheduler намагатиметься не розміщувати нові репліки на вузлі, проте це не є гарантованим;
- ефект «NoExecute» визначає набір дій, при якому Scheduler не буде розміщувати нові репліки на вузлі та вилучить існуючи репліки, які не відповідають цьому Taint.

Tolerations призначаються реплікам і дозволяють їм ігнорувати один або декілька міток Taints. Tolerations теж представляють собою набір міток, єдиною відміною цих міток від міток Taints є оператор порівняння, що є четвертим компонентом мітки, значення оператору визначає логіку порівняння міток Taints та міток Tolerations між собою.

Другим етапом вибору обчислювального вузла є фаза ранжування. На цьому етапі планувальник оцінює і порівнює придатність кожного з вузлів, що не були відсіяні після першої фази, використовуючи оцінювальні функції, щоб визначити найкращий вузол для розгортання репліки.

Розглянемо основні оцінювальні функції, які використовує планувальник під час ранжування вузлів.

Функція Resource Allocation Scoring [46] має на меті оцінити доступні обчислювальні ресурси вузлів відповідно до запиту репліки, яка розгортається. Загальна формула для обчислення оцінки доступних ресурсів вузла виглядає наступним чином (формула 2.1):

$$Score = \frac{Request_{resource}}{Capacity_{resource}} \times 100 \quad (2.1)$$

де $Request_{recource}$ – кількість ресурсу, який запитує репліка,

$Capacity_{recouse}$ – загальна кількість ресурсу, доступна на вузлі.

Функція Balanced Resource Allocation [47] оцінює загальне співвідношення використання процесорного часу та пам’яті між відсіяними вузлами кластера, балансуючи значення цих ресурсів для підтримання високої ефективності та відмовостійкості кластера. Загальна формула для обчислення співвідношення цих ресурсів поміж вузлами виглядає наступними чином (формула 2.2):

$$Score = 1 - \left| \frac{\frac{Usage_{cpu}}{Capacity_{cpu}} - \frac{Usage_{memory}}{Capacity_{memory}}}{\frac{Usage_{cpu}}{Capacity_{cpu}} + \frac{Usage_{memory}}{Capacity_{memory}}} \right| \quad (2.2)$$

де $Usage_{cpu}$ – кількість ЦП, який вже використовується вузлом,

$Capacity_{cpu}$ – загальна кількість ЦП, яка доступна на вузлі,

$Usage_{memory}$ – кількість пам’яті, яка вже використовується вузлом,

$Capacity_{memory}$ – загальна кількість пам’яті, яка доступна на вузлі.

2.3.2 Менеджери пакетів як альтернативний метод розгортання у Kubernetes

Як правило, для повноцінного розгортання додатку в кластері Kubernetes недостатньо розгорнути лише маніфест Deployment. Інтеграція додатку в велику екосистему часто вимагає створення та розгортання додаткових ресурсів Kubernetes, таких як Service, Ingress, ConfigMap та мережеві політики. Наприклад, ресурси типу Service визначають, як клієнтські запити розподіляються між різними розгорнутими репліками застосунку, в той час як Ingress контролює доступ до застосунку з зовнішніх мереж.

Важливо, щоб розгортання реплік та всіх додаткових ресурсів було атомарним, тобто унеможливити випадки, коли частина ресурсів розгорнута успішно, а чистина не розгорнута на кластері взагалі. Цю проблему вирішують менеджери пакетів Kubernetes.

Менеджер пакетів – це інструмент, що значно спрощує процес розгортання, оновлення та управління ресурсами в кластері, дозволяючи впорядковувати і розгортати комплексні контейнеризовані додатки як одне ціле.

Стандартом індустрії серед менеджерів пакетів Kubernetes є Helm [48]. Лідерство Helm серед інших менеджерів обумовлено тим, що він був одним із перших інструментів, що запропонував ефективне рішення для управління життєвим циклом додатків в Kubernetes. Так само як і Kubernetes, технологія Helm є частиною CNCF та оцінена як «Graduated», що означає досягнення високого рівня зрілості та стабільності інструмента, що є важливим маркером для організацій, які розглядають можливість використання цих технологій.

Пакет в термінології Helm називається чартом (Helm Chart), він містить шаблони маніфестів всіх необхідних ресурсів Kubernetes для атомарного розгортання додатку та його залежностей в кластері. Чарт в Helm складається з наступних компонентів:

- файл «Chart.yaml» представляє основний файл, який містить метадані про чарт: назву, версію, опис та інформацію про залежності;
- файл «values.yaml» містить значення за замовчуванням для параметрів, які використовуються у шаблонах чарту;
- директорія «templates» містить шаблони маніфестів Kubernetes, які генеруються за допомогою шаблонізатора Go, директорія може включати шаблони різних ресурсів, таких як Deployment, Service, Ingress тощо;
- директорія «charts» використовується для зберігання залежних чартів, які необхідно розгортати разом із основним чартом.

Шаблонізація маніфестів дозволяє створювати універсальні шаблони застосунків, які можна легко адаптувати та повторно використовувати для різних середовищ або конфігурацій без необхідності змінювати маніфести для кожного нового розгортання в кластері Kubernetes.

За розгортання, оновлення та видалення чартів у кластері Kubernetes відповідає клієнт Helm CLI [49]. Взаємодія з клієнтом відбувається через інтерфейс командного рядка. Клієнт Helm взаємодіє напряму з Kubernetes API цільового кластеру, використовуючи конфігураційні файли Kubernetes для аутентифікації.

Ще однією значною перевагою Helm є публічний репозиторій, який дозволяє публікувати, зберігати та ділитися чартами. Це дозволяє розгортати комплексні інфраструктури вже існуючих чартів в кластері виконанням однієї команди в клієнті Helm.

Для практичного дослідження Helm, реалізовано чарт для розгортання тестового додатку, який буде розширюватися по мірі дослідження існуючих компонентів Kubernetes, які необхідні для стабільного функціонування сервісу.

Створено файл «values.yaml» із специфічними значеннями для тестового додатку: кількість реплік, необхідні обчислювальні ресурси, назва контейнеру в реєстрі Docker.

Файл «values.yaml» з конфігурацію значень шаблону тестового додатку:

```
replicaCount: 1
image:
  repository: diploma-sample-java-app
  tag: "latest"

resources:
  requests:
    cpu: 250m
    memory: 256Mi
```

В директорії «templates» Helm чарту створено шаблон для маніфесту ресурсу Deployment, який раніше було розглянуто в секції 3.3, проте маніфест мав специфічні значення для тестового додатку.

Шаблон маніфесту Deployment в чарті:

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: {{include "diploma-chart-sample.fullname" .}}
  labels:
    {{- include "diploma-chart-sample.labels" . | nindent 4}}
spec:
  replicas: {{.Values.replicaCount }}
  selector:
    matchLabels:
      {{- include "service.selectorLabels" . | nindent 6 }}
  template:
    spec:
      containers:
```

```
- name: {{.Chart.Name}}
  image: "{{.Values.image.repository}}:{{.Values.image.tag}}"
  resources:
    {{- toYaml .Values.resources | nindent 12}}
  ports:
    - name: http
      containerPort: {{.Values.service.port}}
      protocol: TCP
```

Для розгортання чарту у кластері Kubernetes використовується команда «helm upgrade –install [назва_чарту]», після її виконання Helm підставить значення із файлу «values.yaml» в шаблони та розгорне всі заповнені маніфести в кластері.

Таким чином, створений Helm Chart може використовуватися для розгортання будь-яких контейнеризованих додатків, так як конфігурація шаблону Deployment винесена в окремий файл та може бути змінена без необхідності впровадження оновлень самого маніфесту.

2.4 Конфігурація маршрутизації запитів в Kubernetes

Механізм Service Discovery [50] (виявлення сервісів) використовується для динамічної маршрутизації та під’єднання до сервісів в кластері. Kubernetes надає вбудовані механізми для виявлення сервісів, які дозволяють додаткам будувати маршрути та обмінюватися інформацією між собою. Цей механізм є ключовим в розробці та проєктуванні розподілених інформаційних систем, наприклад мікросервісної інфраструктури.

Для надання доступу до групи реплік одного контейнеризованого додатку, в Kubernetes використовує ресурс Service [51]. В ньому визначається правила доступу та конфігуруються кінцеві точки (IP-адреса та порт) для взаємодії. Таким чином, коли репліка потребує звернення до іншого сервісу, вона використовує IP-адресу та порт для з’єднання, які є статичними в межах кластеру, а оркестратор автоматично направляє трафік на одну з реплік іншого сервісу.

Kubernetes підтримує декілька типів сервісів:

– тип ClusterIP маршрутизує запиту до внутрішньої IP-адреси кластеру;
– тип NodePort дозволяє маршрутизувати запити ззовні до певного порта на вузлі, який потім перенаправляє їх до певної розгорнутої репліки;

– тип LoadBalancer маршрутизує зовнішні запити до сервісу, забезпечуючи балансування навантаження між різними репліками.

Таким чином сервісам-клієнтам не обов’язково мати інформацію про кожну репліку сервісу, а достатньо лише мати IP-адресу та порт надбудованої абстракції Service (див. рис. 2.1).

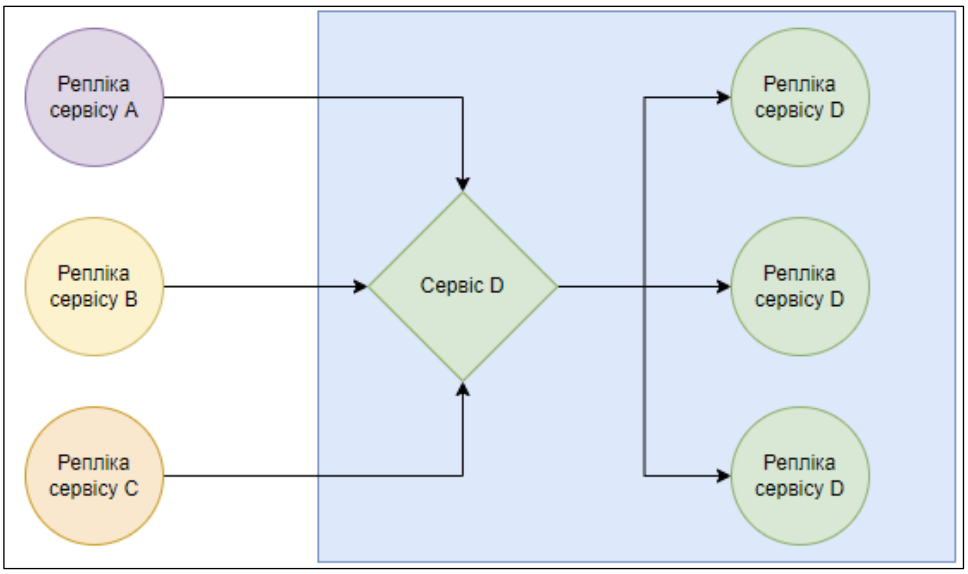

Рисунок 2.1 – Діаграма Service Discovery механізму

Для рівномірного розподілення навантаження між репліками одного сервісу в Kubernetes за замовчуванням використовується алгоритм Round Robin (див. рис. 2.2), тобто вхідні запити розподіляються між доступними репліками по черзі.

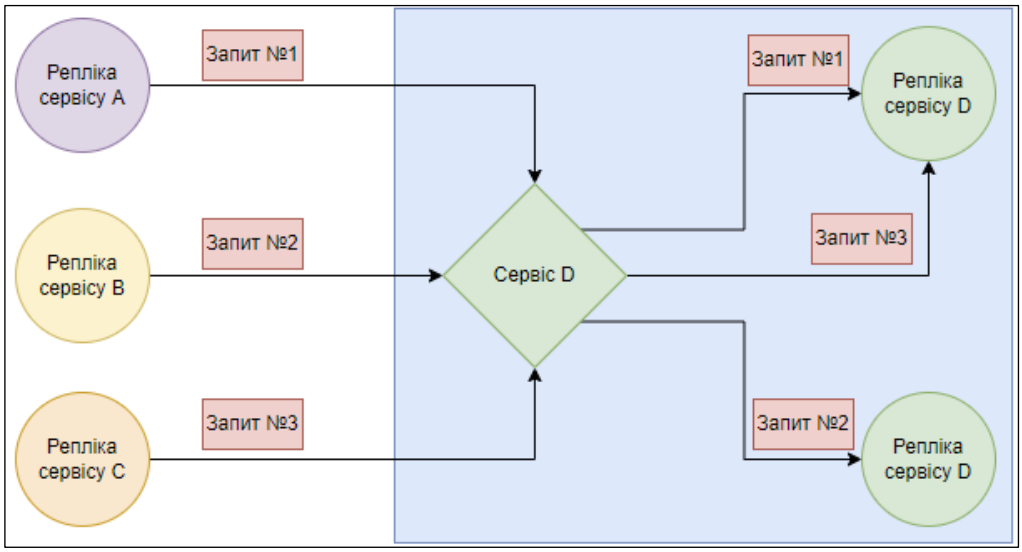

Рисунок 2.2 – Візуалізація балансування запитів алгоритмом Round Robin

Доповнимо Helm чарт шаблоном для створення ресурсу Service з типом LoadBalancer [52] для маршрутизації зовнішнього трафіку до групи реплік тестового додатку та балансування навантаження.

Шаблон ресурсу Service LoadBalancer в Helm чарті тестового додатку:

```
apiVersion: v1
kind: Service
metadata: ...
spec:
  type: LoadBalancer
  ports:
    - port: {{ .Values.loadbalancer.port }}
      targetPort: {{ .Values.loadbalancer.targetport }}
      protocol: TCP
  selector:
    {{- include "diploma-chart-sample.selectorLabels" . | nindent 4}}
```

Нові значення для шаблону Service LoadBalancer в «values.yaml»:

```
...
loadbalancer:
  port: 8086
  targetport: 8086
```

Згідно цієї конфігурації під час розгортання додатку за допомогою Helm, в кластері Kubernetes буде створено сутності Service з типом ClusterIP, Service з типом LoadBalancer.

2.5 Перевірка стану реплік, механізм Helm Check в Kubernetes

Механізм Health Check [53] (перевірка працездатності) використовується для визначення статусу розгорнутого контейнеризованого додатку в кластері Kubernetes. Такі перевірки допомагають оркестратору зрозуміти справність контейнеру та прийняти рішення щодо його життєвого циклу, наприклад перезапуск контейнеру у разі негативної відповіді або коригування кількості реплік. Алгоритм механізму Health Check наведений на рисунку 2.3.

Kubernetes підтримує два механізми перевірки працездатності контейнерів:

– Liveness Probes [54] механізм визначає чи працює контейнер, періодично перевіряючи реакцію програми за допомогою запиту на заздалегідь

налаштовану кінцеву точку HTTP або TCP. Якщо тест працездатності не вдався, то оркестратор перезавантажує контейнер;

– Readiness Probes [55] механізм працює та налаштовується подібно до Liveness Probes, але має інше призначення – він сигналізує Kubernetes, коли контейнер готовий приймати запити.

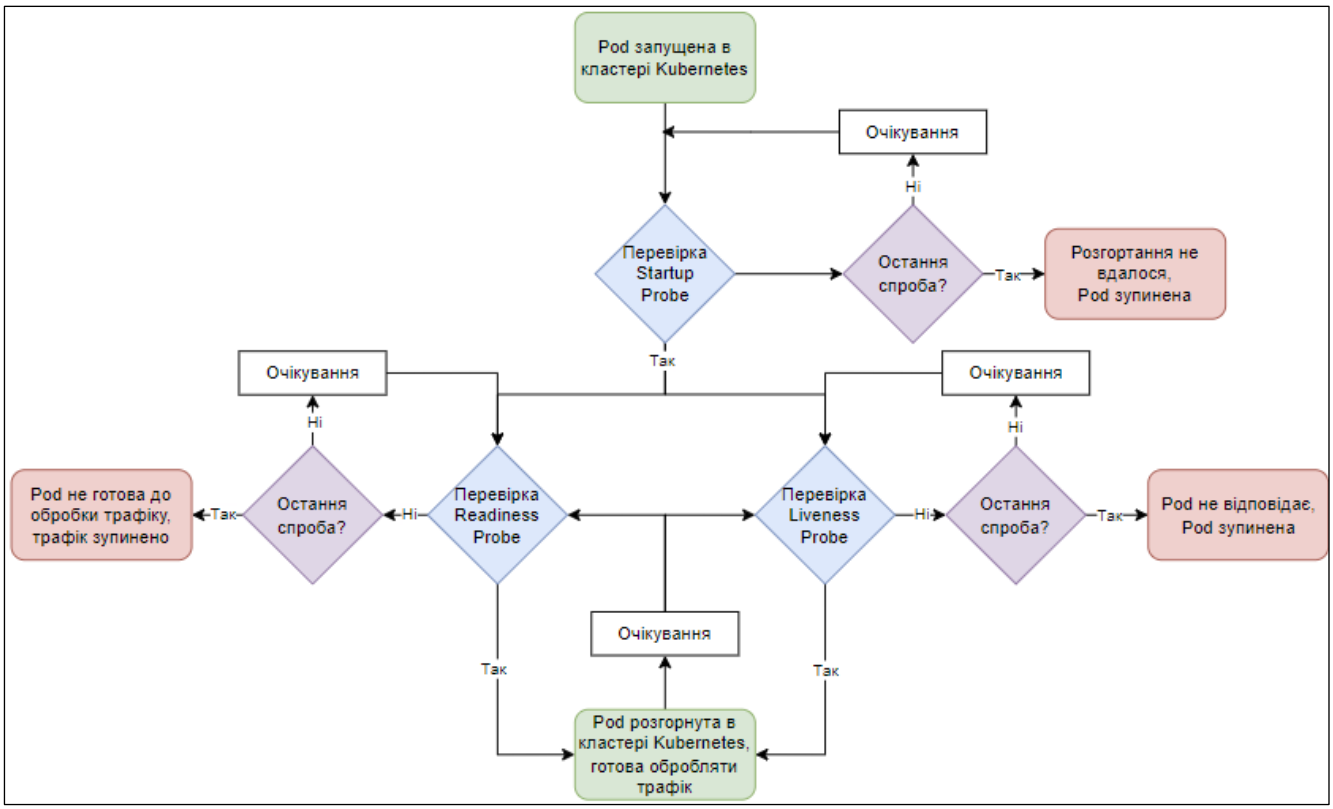

Рисунок 2.3 – Візуалізація алгоритму Health Check в Kubernetes

Конфігурування механізму Health Check для тестового додатку потребує додаткових змін в самому сервісі, а саме додавання кінцевої точки HTTP, яка буде слугувати сигналізатором для Kubernetes, що розгорнута репліка є справною та не потребує перезавантаження, або про те, що контейнер готовий приймати запити.

У пункті 2.1.1 згадується, що фреймворком для розробки тестового додатку обрано Spring Boot 3. Цей фреймворк має модуль Spring Boot Actuator [56], після під'єднання і налаштування якого сервіс буде відкривати кінцеву точку HTTP зі шляхом «/actuator/health», який можна інтегрувати з механізмом Health Check для визначення стану розгорнутого додатка.

Модуль Actuator додано та налаштовано для тестового додатку, відповідні зміни внесені до шаблону маніфесту Deployment та файлу «values.yaml» в чарті.

Зміни в шаблоні маніфесту Deployment для підтримки Health Check:

```
livenessProbe:
  {{- toYaml .Values.livenessProbe | nindent 12 }}
readinessProbe:
  {{- toYaml .Values.readinessProbe | nindent 12 }}
```

Налаштування кінцевих точок Health Check для тестового додатку:

```
livenessProbe:
  enabled: true
  initialDelaySeconds: 30
  periodSeconds: 10
  timeoutSeconds: 5
  failureThreshold: 6
  successThreshold: 1
  httpGet:
    path: /actuator/health
    port: 8080

readinessProbe:
  enabled: true
  initialDelaySeconds: 30
  periodSeconds: 10
  timeoutSeconds: 5
  failureThreshold: 6
  successThreshold: 1
  httpGet:
    path: /actuator/health
    port: 8080
```

Після реалізації даної конфігурації, стан розгорнутих реплік тестового додатку буде відстежуватися спеціальним агентом Kubernetes, який знаходиться на кожному вузлі кластера та відподає за керування репліками, – kubelet [57].

2.6 Стратегії оновлення реплік в Kubernetes

Так як сучасні інформаційні системи постійно оновлюються, а процеси оновлення та розгортання нових версій реплік мають бути непомітними для користувачів системи, оркестратори контейнерів підтримують різні стратегії для оновлення групи реплік.

Стратегії оновлення реплік конфігуруються безпосередньо в маніфесті ресурсу Deployment, який є відповідальним за управління життєвим циклом

групи реплік. Kubernetes підтримує 2 стратегії оновлення: Recreate та Rolling Update.

Найпростішою інтегрованою стратегією оновлення реплік в Kubernetes є Recreate (повторне створення). Ця стратегія передбачає одночасне припинення роботи всіх розгорнутих реплік додатку та створення нових реплік з оновленою версією. Покрокову візуалізацію виконання стратегії зображено на рисунку 2.4.

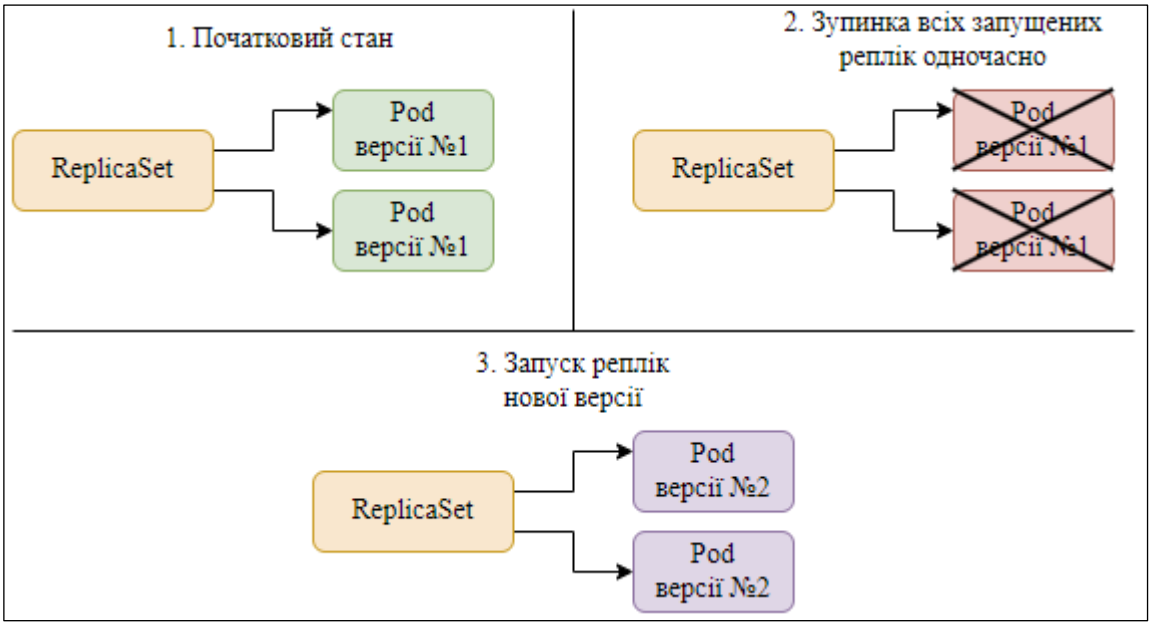

Рисунок 2.4 – Візуалізація стратегії оновлення реплік Recreate

Основною перевагою стратегії Recreate є консистентність стану додатку, так як ця стратегії гарантує, що жодної репліки старої версії не буде розгорнуто одночасно з новими. Також ця стратегія є технічно простішою для впровадження, так як немає необхідності управляти паралельними версіями.

Головним недоліком стратегії Recreate є повна недоступність реплік додатку під час оновлення, що є критичним для застосунків з високими вимогами до доступності.

Стандартом серед стратегій оновлення є стратегія Rolling Update (поступове оновлення). Вона передбачає поступове заміщення старих версій реплік новими, одна за одною або невеликими групами, без зупинки всіх реплік додатку одночасно. Нові репліки запускаються та перевіряються на коректну роботу за допомогою Readiness Probe механізму Health Check, перш ніж

зупиняються репліки старих версій. Покрокову візуалізацію виконання стратегії зображено на рисунку 2.5.

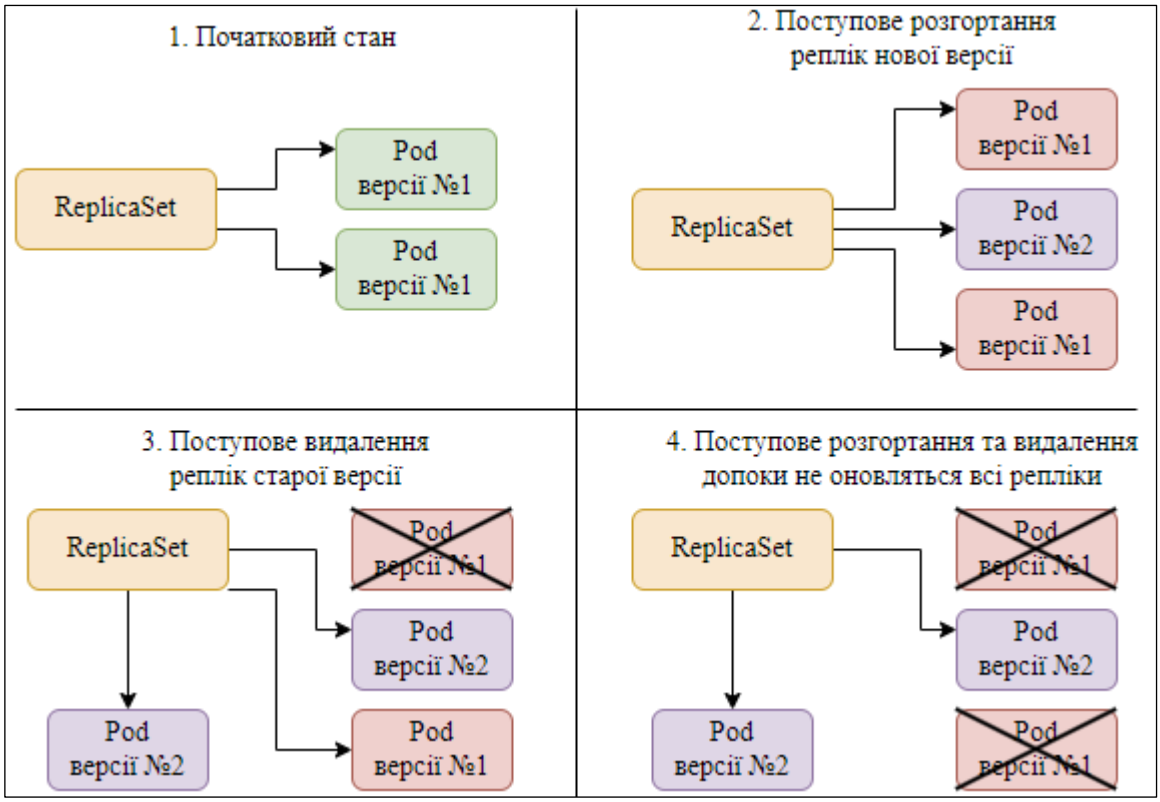

Рисунок 2.5 – Візуалізація стратегії оновлення реплік Rolling Update

Kubernetes дозволяє конфігурувати параметри виконання Rolling Update:

– параметр «maxUnavailable» вказує на максимальну кількість реплік, які можуть одночасно бути недоступними під час оновлення, зменшення значення цього параметра збільшує надійність, проте сповільнює виконання стратегії оновлення;
– параметр «maxSurge» конфігурує кількість реплік, які можуть бути створенні під час оновлення понад налаштовану максимальну кількість реплік додатку, це дозволяє збільшити швидкість розгортання нової версії, але вимагає більше ресурсів.

Найбільшою перевагою Rolling Update є неперервне обслуговування запитів під час оновлення, так як стратегія унеможливлює випадки, в яких не розгорнуто хоча б однієї репліки додатку. В разі невдалого оновлення

Kubernetes, тобто якщо Readiness Probe не вдається перевірити для реплік нової версії, Kubernetes не буде продовжувати виконання стратегії.

Недоліками цієї стратегії є можливі проблеми зі сумісністю одночасно розгорнутих реплік нових та старих версій та використання додаткових ресурсів під час розгортання.

Розширимо шаблон маніфесту ресурсу Deployment чарту Helm можливістю конфігурування стратегії оновлення реплік та налаштуємо стратегію для тестового додатку.

Модифікований шаблон ресурсу Deployment чарту:

```
strategy:
  type: {{ .Values.strategy.type }}
  rollingUpdate:
    maxUnavailable: {{ .Values.strategy.maxUnavailable }}
    maxSurge: {{ .Values.strategy.maxSurge }}
```

Конфігурація стратегії розгортання в values.yaml тестового додатку:

```
strategy:
  type: RollingUpdate
  maxSurge: 1
  maxUnavailable: 0
```

Така конфігурація гарантує, що процес безперервного оновлення не допустить ситуацій, коли жодної репліки не буде розгорнуто, і дозволить контролювати масштабування, не допускаючи розгортання більше однієї репліки за один раз.

2.7 Автоматичне масштабування контейнеризованих додатків

Автоматичне масштабування – це процес адаптації розгорнутого додатку до змінюваних вимог навантаження шляхом збільшення або зменшення ресурсів, які використовуються для його роботи.

Поділяють два принципово різних методи регулювання обчислювальних ресурсів додатку під час масштабування – вертикальний та горизонтальний.

Метод вертикального масштабування [58] заснований на збільшенні чи зменшенні обчислювальних ресурсів наявного вузла, на якому працює додаток.

Як правило, цей метод використовується для застосунків, які не можуть бути розподілені на декілька вузлів, монолітних системах.

Перевагою цього методу є те, що його впровадження не потребує значних змін в архітектурі застосунку, так як відсутня необхідність в складних механізмах розподілу навантаження та синхронізації даних між різними вузлами.

Головним недоліком методу є фізичні обмеження на те, скільки пам’яті, процесорної потужності та інших ресурсів може підтримувати один сервер, тобто вертикальне масштабування має межу, після підвищення якої подальше підвищення ресурсів стає неможливим або неефективним.

Горизонтальне масштабування [59] на відміну від вертикального не регулює обчислювальні ресурси серверу, на якому виконується застосунок, а збільшує або зменшує кількість розгорнутих додатків на різних вузлах. Зазвичай, цей метод використовується у розподілених системах з використанням хмарних платформ, наприклад системах побудованих за допомогою мікросервісної архітектури, де є можливість динамічно додавати нові вузли до інфраструктури.

Перевагою цього методу є теоретично необмежена масштабованість, оскільки нові вузли можуть додаватися до кластеру нескінченно. Іншою перевагою над вертикальним масштабуванням є висока доступність, оскільки горизонтально масштабований додаток може продовжувати працювати навіть при відмовах окремих вузлів.

Недоліком методу є необхідність реалізації стратегій балансування навантаженнями та синхронізації даних між розгорнутими репліками додатку. Горизонтально масштабовані додатки вимагають високої продуктивності до мережевої інфраструктури для ефективного обміну даними між вузлами.

Отже, вертикальне масштабування забезпечує меншу складність реалізації, проте його можливості обмежені фізичними ресурсами одного сервера. Горизонтальне масштабування, навпаки, надає значну гнучкість за рахунок можливості додавання необмеженої кількості вузлів,

2.8 Традиційний метод масштабування в Kubernetes, ресурсні метрики

Традиційно, автоматичне горизонтальне масштабування в Kubernetes відбувається за рахунок вбудованого компоненту Horizontal Pod Autoscaler (горизонтальний автоматичний масштабувач кількості реплік, далі HPA) [60].

HPA розгортається в кластері за принципом «один Deployment – один екземпляр компоненту HPA», тож кожна група реплік має відокремлену розгорнуту сутність HPA з специфічною конфігурацією.

Стандартна імплементація вбудованого Kubernetes HPA передбачає наступний набір параметрів:

- «minReplicas» - мінімальна кількість реплік, яка буде підтримуватися навіть під час наднизького навантаження на групу реплік;
- «maxReplicas» - максимальна кількість реплік, за рамки якої HPA не буде збільшувати кількість реплік під час надвисокого навантаження;
- «metrics» - набір метрик, на базі яких здійснюється масштабування;
- «targetValue» - порогове значення метрики, при перевищені якого кількість реплік має збільшуватися, а при не досягненні значення – зменшуватися;
- «targetCPUUtilizationPercentage» та «targetMemoryUtilizationPercentage» - порогові значення утилізації процесору та пам’яті в процентах відповідно;
- «stabilizationWindowSeconds» - період часу у секундах, протягом якого HPA очікує стабілізацію рівня навантаження на групу реплік, перед тим як виконати наступне масштабування;
- «behavior» - стратегія масштабування кількості реплік, або набір правил та параметрів, який визначає коли і як буде прийняте рішення про зміну кількості реплік на основі метрик, які спостерігаються.

Традиційний метод масштабування передбачає масштабування кількості реплік на базі ресурсних метрик, які можуть бути надані вбудованими

механізмами для моніторингу в Kubernetes: утилізація ЦП, використання пам’яті, навантаження на мережу тощо.

2.8.1 Отримання та збір ресурсних метрик кластера Kubernetes

Для отримання значення метрики в реальному часі, таким чином дозволяючи HPA та іншим операторам моніторингу контролювати стан кластера та реагувати на зміни навантаження, існує інтерфейс Kubernetes Metric API [61].

В базовій конфігурації, Kubernetes не збирає ресурсні метрики з розгорнутих реплік автоматично, тому для цього треба розгорнути в кластері репліку додатку Kubernetes Metrics Server [62].

Kubernetes Metrics Server – це утилітний сервіс розроблений Kubernetes Community, який збирає метрики використання ресурсів з Kubelet-ів та надає до них доступ через Kubernetes Metric API. Метрики, отримані за допомогою цієї утиліти, призначені лише для використання в Kubernetes HPA.

Kubelet – це компонент Kubernetes, який працює на кожному вузлі кластера і відповідає за управління контейнерами на ньому. Kubelet збирає метрики використання ресурсів та стан контейнерів на вузлі.

Таким чином Metrics Server – це проксі-обгортка над інформацією про ресурси всіх компонентів типу Kubelet. Алгоритм взаємодії Kubernetes Metric API з Metrics Server наведений на рисунку 2.6 нижче.

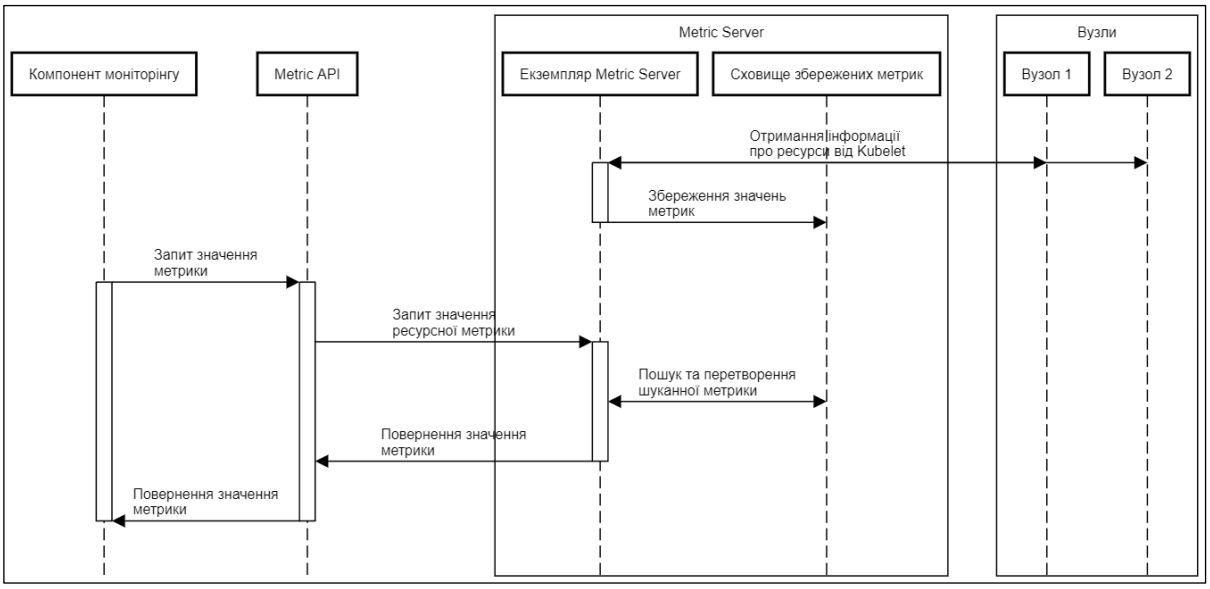

Рисунок 2.6 – Діаграма послідовності взаємодії Metric API з Metric Server

В Metrics Server базовою частотою опитування всіх Kubelet в кластері є 15 секунд. Цей параметр є незмінним, тобто його не можливо переконфігурувати під час розгортання екземпляру утиліти.

Як правило, Metrics Server розгортається в кластері за принципом «один кластер – одна репліка сервісу», але для більшої надійності в умовах високої загальної завантаженості кластера, можна розгортати більше ніж один екземпляр утиліти. В такому випадку, кожна репліка утиліти буде збирати метрики з вузлів, та лише одна репліка буде активно обслуговувати запити з Kubernetes Metric API.

2.8.2 Алгоритм роботи Horizontal Pod Autoscaler

Загальний алгоритм роботи компоненту HPA представлений на діаграмі зображеній на рисунку 2.7.

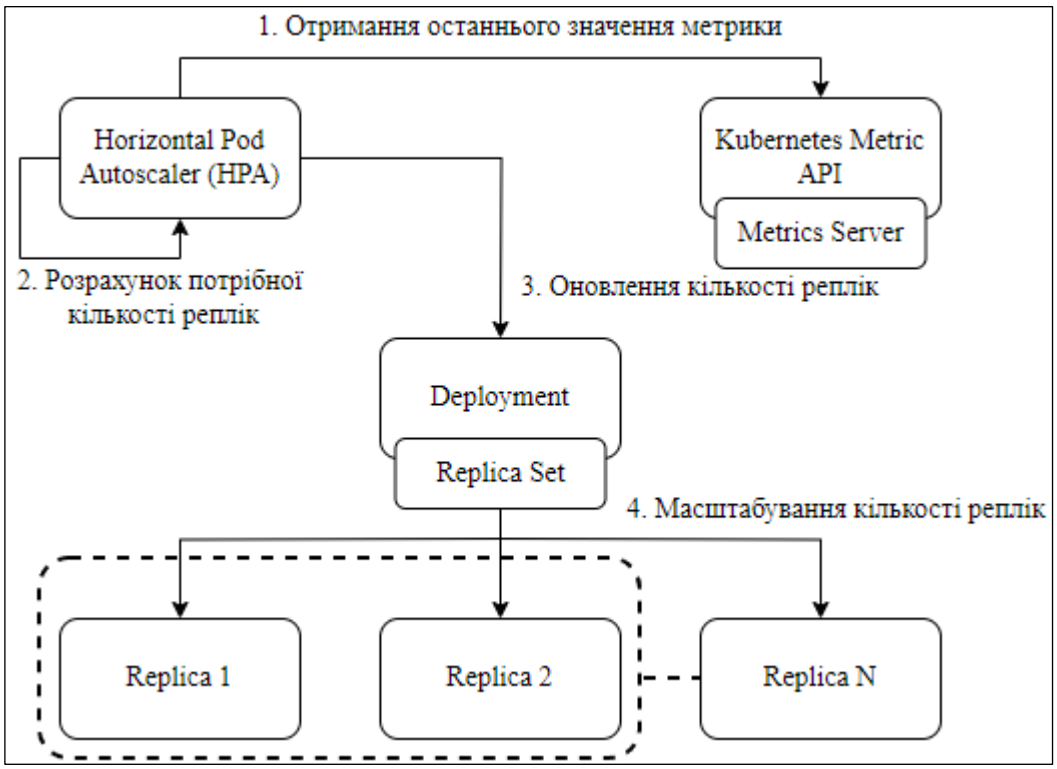

Рисунок 2.7 – Загальний алгоритм роботи Kubernetes HPA

Як можна побачити з цієї діаграми, HPA отримує значення метрики з Kubernetes Metric API, розраховує потрібну кількість реплік відповідно до дійсного значення та оновлює кількість реплік в сутності Deployment, яка

займається управлінням життєвим циклом реплік, тобто збільшує або зменшує кількість реплік відповідно до запиту HPA.

Без додаткової конфігурації, HPA використовує просту логіку для визначення потрібної кількості реплік відповідно до отриманого значення метрики, яка спостерігається. Формула 2.3 розрахунку значення потрібної кількості реплік наступна:

$$R = \frac{M_c}{M_t}$$
$$P_t = P_c \times R, \tag{2.3}$$

де M_c – дійсний показник метрики,

M_t – цільовий показник метрики,

R – співвідношення дійсного показника метрики до цільового (ratio),

P_c – дійсна кількість розгорнутих реплік,

P_t – цільова кількість розгорнутих реплік.

На основі обчисленого значення P_t, HPA вирішує, чи масштабувати кількість реплік вгору або вниз: якщо бажана кількість реплік (P_t) більша за поточну (P_c), HPA збільшує кількість реплік, якщо менша – зменшує.

2.8.3 Переваги та недоліки методу

Головною перевагою цього методу є повна інтеграція з кластером, на якому розгорнуто HPA та Metrics Server, так як ці технології є вбудованими частинами Kubernetes. Повна інтеграція надає наступні переваги цьому підходу:

- просте та швидке розгортання в кластері всієї інфраструктури, яка необхідна для функціонування методу, так як і HPA, і Metrics Server легко розгортаються за допомогою стандартних YAML-маніфестів;
- Metrics Server автоматично збирає метрики з усіх вузлів та реплік кластеру, використовуючи вбудовані механізми Kubernetes;

– автоматичне масштабування на основі ресурсних метрик гарантує, що жодна репліка з високонавантаженої групи не перетне ліміт по використанню ресурсів, якщо HPA, який контролює стан цієї групи, конфігуровано правильно.

Проте, масштабування лише на базі ресурсних метрик є й головним недоліком цього методу, оскільки ці метрики не відображають реальне навантаження на систему та якість обслуговування користувачів.

Використання сервісних метрик в комбінації з ресурсними дозволяє точніше реагувати на зміну умов, забезпечуючи стабільну роботу сервісів та ефективне використання ресурсів.

2.9 Метод масштабування в Kubernetes на базі Prometheus, сервісні метрики

В основі цього методу також використовується HPA та Kubernetes Metric API, проте замість Metrics Server для збору та надання значень метрики використовується Prometheus [63].

Prometheus є стандартом індустрії у сфері моніторингу, збирання сервісних та ресурсних метрик, створений для роботи у високодинамічних контейнерних середовищах. Зараз Prometheus є окремим проектом з відкритим вихідним кодом і частиною CNCF.

За своєю суттю, Prometheus – це база даних, яка спеціалізована на зберіганні та обробці часових рядів (time series) [64], що представляють собою послідовність пар часових маркерів та числових значень, які відображають певні аспекти функціонування системи або окремих сервісів. Така реалізація робить можливим виявлення трендів та аномалій в роботі кластеру, та дозволяє операторам системи, таким як Kubernetes HPA, реагувати на потенційні проблеми, забезпечуючи стабільну та ефективну роботу інфраструктури.

Інтеграція розгорнутої інфраструктури Prometheus з Kubernetes HPA дозволяє реалізовувати більш складні сценарії масштабування кількості реплік, використовуючи не лише ресурсні метрики, але й сервісні, які збирає Prometheus. Сервісні метрики відображають специфічні аспекти стану додатків,

які розгорнуті в кластері, такі як кількість необроблених системних повідомлень в черзі, час на обробку користувацьких запитів тощо.

Prometheus не є вбудованою частиною та не входить до набору стандартних інструментів Kubernetes, тому його необхідно розгортати та налаштовувати в кластері окремо.

Наразі, існує публічний Helm чарт «kube-prometheus-stack» [65], який автоматично встановлює та розгортає всю необхідну інфраструктуру в кластері:

Команда для розгортання Helm чарт kube-prometheus-stack в кластері:

```
helm update --install prometheus prometheus-community/kube-prometheus-stack
```

Після інсталяції інфраструктури для отримання доступу до інтерфейсу Prometheus, треба виконати операцію переспрямування порту, тобто зробити контейнеризований додаток Prometheus доступним поза кластером.

Команда для переспрямування запитів на виокремлений порт:

```
kubectl port-forward prometheus-kube-promet-prometheus-0 9090:9090
```

Інтерфейс Prometheus став доступним за адресою localhost:9090 (див. рис. 2.8), та користувач має можливість виконати запит PromQL, переглянути інформацію про репліки, з яких збираються метрики, тощо.

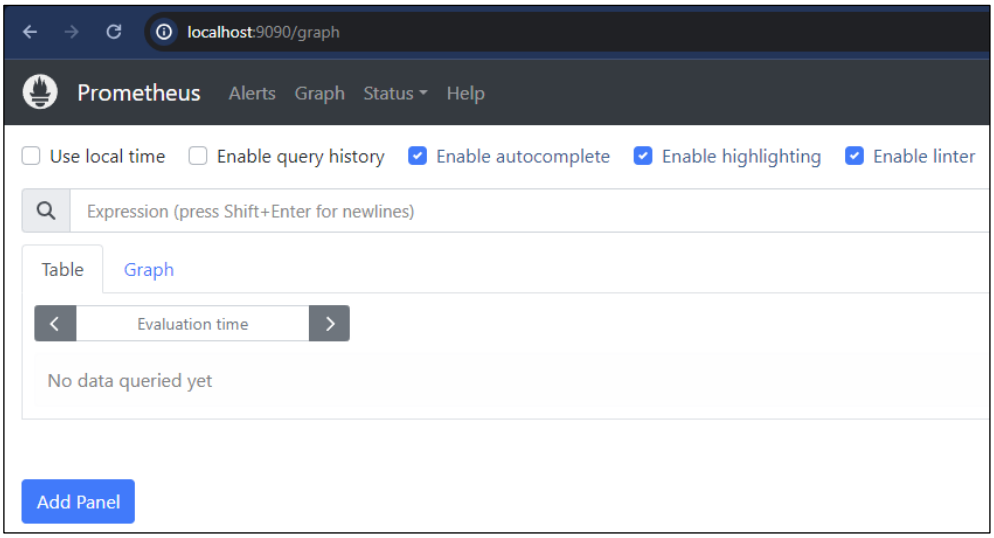

Рисунок 2.8 – Інтерфейс додатку Prometheus розгорнутого в кластері Kubernetes

2.9.1 Користувацький ресурс ServiceMonitor

Для отримання актуальних метрик Prometheus використовує Scraping Strategy (стратегія збирання). Стратегія полягає у систематичному зчитуванні значень метрик з попередньо налаштованих URL-адрес за визначеним інтервалом часу. Цей процес забезпечує неперервне оновлення метричних даних, що важливо для своєчасного виявлення змін у стані системи або окремих додатків.

Визначення конфігурації збирання метрик для кожної окремої групи реплік здійснюється за допомогою ServiceMonitor [66], що є користувацьким ресурсом Kubernetes. Тобто для того, щоб розгорнута інфраструктура Prometheus отримала інформацію з якого Kubernetes Service треба збирати метрики, для кожного Service має бути створений окремий користувацький ресурс ServiceMonitor під час розгортання.

Детально розглянемо приклад Kubernetes маніфесту для розгортання сутності ServiceMonitor в кластері.

Маніфест для розгортання користувацького ресурсу ServiceMonitor:

```
apiVersion: monitoring.coreos.com/v1
kind: ServiceMonitor
metadata:
  name: diploma-servicemonitor
  labels:
    release: prometheus
spec:
  selector:
    matchLabels:
      app: diploma-app
  namespaceSelector:
    matchNames:
    - diploma-namespace
  endpoints:
  - port: web
    interval: 30s
    path: /metrics/prometheus
    scheme: http
```

Поле «metadata.labels.release» використовується в якості мітки, щоб розгорнута інфраструктура Prometheus при скануванні Kubernetes ресурсів типу

ServiceMonitor могла ідентифікувати та розпізнавати ресурси, і включати їх у конфігурацію для збору метрик.

Перелік міток і правил для ідентифікації ServiceMonitor створюється під час розгортання інфраструктури Prometheus та може бути змінений.

Конфігурація міток ресурсів ServiceMonitor в Prometheus:

```
serviceMonitorSelector:
    matchLabels:
      release: Prometheus
```

В полі «spec.selector.matchLabels» вказується перелік міток, які мають співпадати з мітками відповідної розгорнутої сутності Service для подальшого збору метрик. Відповідно поле «spec.namespaceSelector» вказує на простір імен Kubernetes, в якому має відбуватися пошук розгорнутої сутності Service.

Поле «spec.endpoints» використовується для конфігурації кінцевої точки («spec.endpoints.path»), протокол взаємодії («spec.endpoints.scheme») та часовий інтервал («spec.endpoints.interval») за яким Prometheus буде збирати метрики з ідентифікованого Kubernetes Service.

За збір конфігурації для збору метрик відповідає компонент Prometheus Operator [67], який є стандартною складовою інфраструктури Prometheus та розгортається автоматично під час встановлення Helm Chart.

Загальний алгоритм збору конфігурації та взаємозв’язків міток зображений на рисунку 2.9.

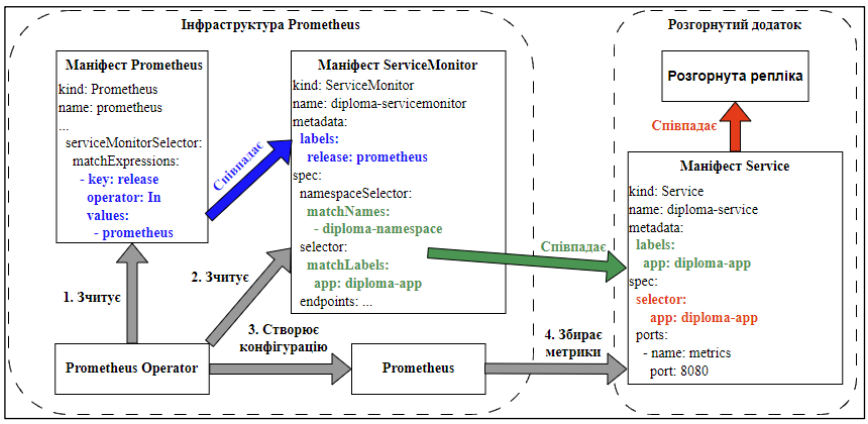

Рисунок 2.9 – Алгоритм пошуку та створення конфігурації для збору метрик в Prometheus Operator

2.9.2 Збір ресурсних метрик Kubernetes за допомогою Prometheus

Для збору ресурсних метрик з запущених реплік Prometheus використовує вбудований в Kubelet інструмент cAdvisor [68] (скорочено від Container Advisor).

cAdvisor аналізує показники використання пам'яті, центрального процесору, файлової системи і мережі для всіх контейнерів, які працюють на відповідну вузлі кластеру Kubernetes.

Отримати доступ до метрик cAdvisor можна здійснивши запит до Kubernetes API за наступним URL:

URL для доступу до метрик cAdvisor:

http://<URL серверу>/api/v1/nodes/<ім'я вузла>/proxy/metrics/cAdvisor

Оглянемо перелік основних ресурсних метрик, які надає cAdvisor:

- «container_cpu_load_average_10s» - середнє значення навантаження процесору контейнера за останні десять секунд;
- «container_memory_usage_bytes» - поточне значення використання пам'яті контейнером в байтах;
- «container_fs_io_time_seconds_total» - загальна кількість секунд, які були витрачені на введення/виведення даних у файловій системі контейнеру;
- «container_network_receive_bytes_total» - загальна кількість байтів, отриманих мережею контейнеру.

При розгортанні Helm Chart «kube-prometheus-stack», автоматично створюється ServiceMonitor для ресурсних метрик cAdvisor.

Шаблон маніфесту ServiceMonitor конфігурації збору метрик cAdvisor:

```
{{- if .Values.kubelet.serviceMonitor.cAdvisor }}
  - port: https-metrics
      scheme: https
    path: /metrics/cadvisor
    {{- if .Values.kubelet.serviceMonitor.interval }}
    interval: {{ .Values.kubelet.serviceMonitor.interval }}
    {{- end }}
```

```
    {{- if .Values.kubelet.serviceMonitor.proxyUrl }}
    proxyUrl: {{ .Values.kubelet.serviceMonitor.proxyUrl }}
    {{- end }}
    {{- if .Values.kubelet.serviceMonitor.scrapeTimeout }}
    scrapeTimeout: {{ .Values.kubelet.serviceMonitor.scrapeTimeout }}
    {{- end }}
{{- end }}
```

2.9.3 Збір сервісних метрик розгорнутих додатків за допомогою Prometheus

Для усестороннього дослідження збору сервісних метрик розгорнутих додатків за допомогою Prometheus, розширимо тестовий та готовий до розгортання в кластері Kubernetes додаток.

В якості інструменту, який надає інтерфейс для збору та експорту метрик було обрано розширення «io.micrometer:micrometer-registry-prometheus», що є частиною бібліотеки Micrometer [69], стандартом індустрії.

Micrometer надає гнучкий API для збору різноманітних метрик, таких як лічильники, таймери і гістограми. За допомогою вбудованого в бібліотеку PrometheusRegistry [70] метрики автоматично реєструються та готові для збору інфраструктури Prometheus.

В якості тестової метрики, яку експортує додаток, обрано лічильник кількості запитів до кінцевої точки HTTP «/api/seeds/predefined». Реалізовано клас-контролер з інкрементуванням лічильника в PrometheusRegistry.

Тестовий клас-контролер на Java для запису метрики-лічильника:

```
@RestController
@RequestMapping("/api/seeds")
@RequiredArgsConstructor
public class MetricsSampleController {

    private final MeterRegistry meterRegistry;
    private final int seed = RandomUtil.getPositiveInt();

    @GetMapping("/predefined")
    public @ResponseBody RandomSeedResponse getPodSeed() {
        var response = new RandomSeedResponse(seed);
        Counter.builder("requests_endpoint_predefined_seed")
                .register(meterRegistry)
                .increment();
        return response;
    }
}
```

За допомогою бібліотеки Spring Boot Actuator та її внутрішньої інтеграції з Prometheus, створено кінцеву точку HTTP «/actuator/prometheus», яка надає інформацію про значення метрик репліки. В цьому стані тестовий додаток готовий до збору та експорту метрик.

Останнім кроком перед розгортанням тестового додатку в кластері Kubernetes, є розширення Helm чарту шаблоном маніфесту ServiceMonitor для надання інформації про конфігурацію збирання метрик.

Helm чарт шаблон маніфесту ServiceMonitor для тестового додатку:

```
apiVersion: monitoring.coreos.com/v1
kind: ServiceMonitor
metadata:
  name: {{ include "service.fullname" . }}
  labels:
    release: dev-prometheus
    {{- include "service.labels" . | nindent 4 }}
spec:
  selector:
    matchLabels:
      {{- include "service.selectorLabels" . | nindent 6 }}
  endpoints:
    - port: http
      path: "/actuator/prometheus"
```

Як можна побачити з наведеного вище шаблону, збір метрик має здійснюватися з кінцевої точки HTTP «/actuator/prometheus», що була заздалегідь налаштована в конфігурації Spring Actuator.

Після розгортання тестового додатку за допомогою Helm в кластері Kubernetes, Prometheus автоматично визначив розгорнуту репліку як цільову (див. рис. 2.10). Інформацію про активні цільові репліки Prometheus (активні репліки, з яких збираються метрики) можна знайти на вкладці «Targets» інтерфейсу Prometheus або за допомогою запиту до Prometheus API за кінцевою точкою HTTP «/api/v1/targets».

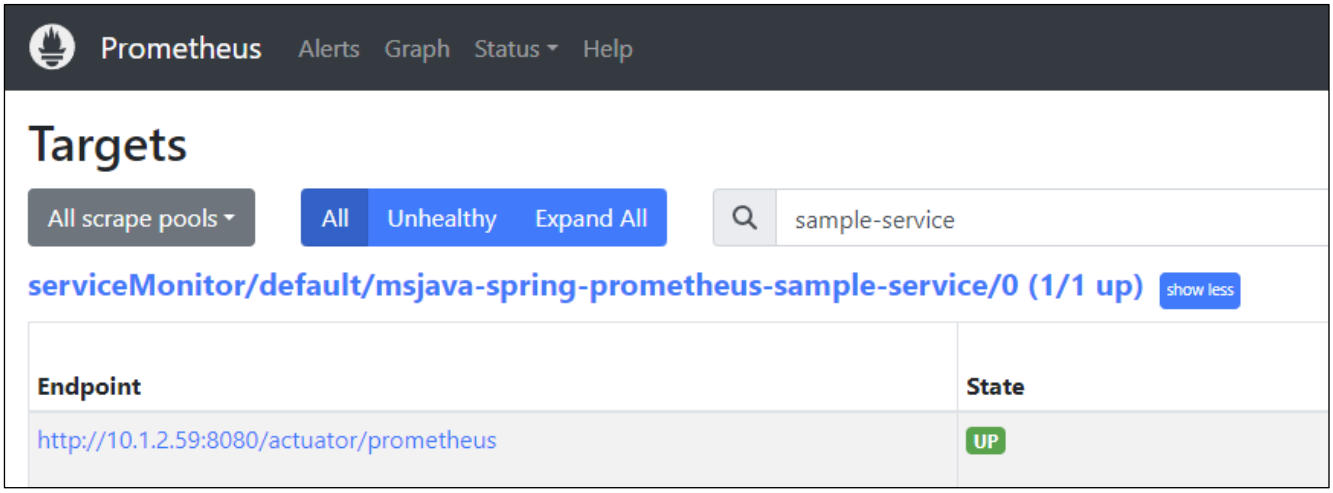

Рисунок 2.10 – Веб-інтерфейс цільових реплік для збору метрик в Prometheus

2.9.4 Інтеграція метрик Prometheus з Kubernetes HPA

Для використання користувацьких метрик, які збирає Prometheus Operator з розгорнутих реплік в кластері, для автоматичного масштабування сутністю HPA необхідного адаптувати метрики у формат, який використовується в Kubernetes.

Задля інтеграції та адаптації метрик Prometheus з Kubernetes HPA та іншими компонентами, які використовують Kubernetes Metrics API, використовується утиліта Prometheus Adapter [71], яка має бути встановлена в кластері.

Команда для розгортання Helm Chart prometheus-adapter в кластері:

```
helm upgrade --install dev-prometheus-adapter prometheus-
community/prometheus-adapter --set prometheus.url=http://dev-
prometheus-kube-promet-prometheus.default.svc
```

Prometheus Adapter отримує метрики з серверів Prometheus і перетворює їх у формат, який може бути використаний HPA. Це дозволяє Kubernetes використовувати користувацькі або специфічні для конкретного додатку метрики, зібрані Prometheus, для прийняття рішень щодо масштабування. Загальний алгоритм збору та адаптації метрик відображено на рисунку 2.11.

Процес сканування та перетворення метрик відбувається за допомогою заздалегідь описаних правил в конфігурації Prometheus Adapter. Розглянемо загальну структуру правила.

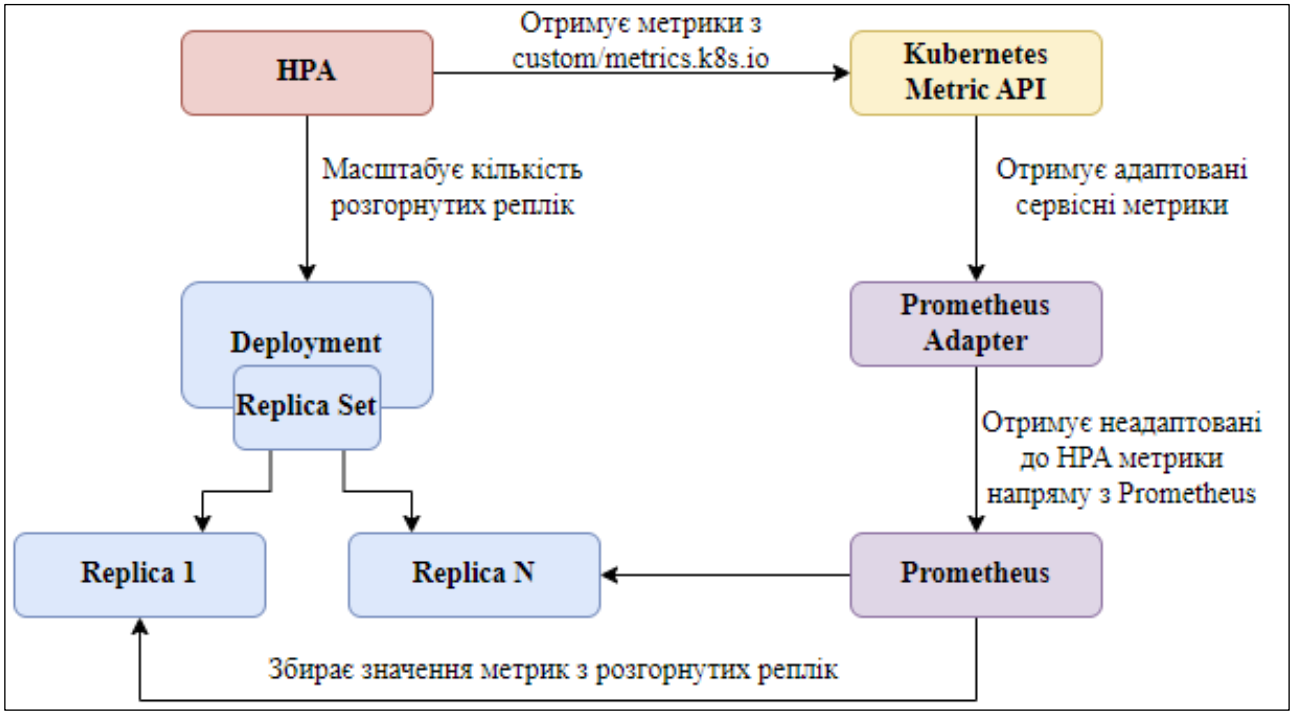

Рисунок 2.11 – Збір та адаптація метрик Prometheus до Kubernetes Metric API

Приклад правила prometheus-adapter для адаптації Prometheus метрик:

```
- seriesQuery: '{__name__=~"^some_metric_count$"}'
  resources:
    template: <<.Resource>>
  name:
    matches: ""
    as: "my_external_metric"
  metricsQuery: sum(<<.Series>>{<<.LabelMatchers>>})
```

В полі «seriesQuery» описується загальна вибірка метрик на PromQL, які мають бути адаптувані в Kubernetes Metrics API.

В полі «metricsQuery» вказується цільовий PromQL-запит, який визначає дійсне значення метрики. Всі метрики, які присутні в загальній вибірці «seriesQuery» будуть вставлені в блок «<<.Series>>» та будуть доступні в Kubernetes Metrics API.

Адаптуємо метрику, яка була описана в розділі 2.9.3, за допомогою Prometheus Adapter, а саме метрику «request_endpoint_predifined_seed», що описує лічильник отриманих HTTP-запитів на кінцеву точку «/api/seeds/predefined».

Правило prometheus-adapter для користувацької метрики в Metrics API:

```
seriesQuery: '{__name__=~"^requests_endpoint_predifined seed .*”,
container!="POD}'
  seriesFilters: [ ]
  resources:
    overrides:
      namespace:
        resource: namespace
      pod:
        resource: pod
  name:
    matches: ^(.*)_total$
    as: ""
  metricsQuery:
rate(<<.Series>>{<<.LabelMatchers>>,container!="POD"}[5m])
```

Після імплементації правила та розгортання Prometheus Adapter з оновленою конфігурацією, метрика з’явиться в окрему Kubernetes Metrics API для користувацьких метрик, а саме в custom.metrics.k8s.io, та буде доступна для використання в Kubernetes HPA для автоматично масштабування кількості репліки базуючись на сервісних метриках.

2.9.5 Переваги та недоліки методу

Головною перевагою методу є використання інфраструктури Prometheus у якості інструменту для збирання та отримання значень метрики, що дозволяє значно розширити можливості автоматичного масштабування, не обмежуючись лише ресурсними метриками.

Використання часових рядів дозволяє точно виявляти тренди та аномалії, забезпечуючи вчасне реагування на зміни в навантаженні та продуктивності системи. Мова запитів PromQL в Prometheus дозволяє виконувати складні маніпуляції з метриками та використовувати результат у якості цільового показника для автоматичного масштабування в HPA.

Недоліком методу є неповна інтеграція Prometheus з Kubernetes, так як для отримання значень метрики в HPA необхідно використовувати Prometheus Adapter, тобто для кожної метрики, яка використовується в інфраструктурі автоматичного масштабування Kubernetes, має бути заздалегідь налаштоване правило в Prometheus Adapter.

Цей недолік є критичним з боку розробки розподілених систем, які складаються з безлічі різних контейнеризованих додатків, де кожен застосунок має власні цільові метрики для автоматичного масштабування.

Іншим недоліком є необхідність створення користувацького ресурсу ServiceMonitor для кожної групи реплік, яка надає метрики. Цей процес може бути автоматизованим за допомогою додавання цього ресурсу до Helm чарту з базовим набором шаблонів ресурсів Kubernetes, який перевикористовується для всіх додатків, що розгортаються в кластері.

3 РОЗРОБКА КОМПОНЕНТУ АВТОМАТИЧНОГО МАСШТАБУВАННЯ

3.1 Мета розробки

Метою розробки є створення компоненту автоматичного масштабування для оркестратора Kubernetes на базі ресурсних та сервісних метрик Prometheus, та створення універсального шаблонізованого методу розгортання додатків з підтримкою реалізованого компоненту за замовчуванням. Основою шаблонізованого методу розгортання є Helm чарт для тестового контейнеризованого додатку, створений під час проведення досліджень методів оркестрування та масштабування.

Причини розробки зумовлені виявленими під час дослідження обмеженнями в традиційних підходах до оркестрації та автоматичного масштабування, серед яких:

- відсутність у загальному доступі універсальних, шаблонізованих моделей для розгортання контейнеризованих додатків, що інтегрують базові функції Kubernetes, такі як автоматизоване масштабування, вибір стратегії оновлення, моніторинг стану розгорнутих реплік, налаштування маршрутизації та балансування навантаження, а також збір метрик за допомогою Prometheus;
- потреба у додатковому розгортанні та конфігуруванні контейнеризованих застосунків та ресурсів Kubernetes для налаштування автоматизованого масштабування на базі ресурсних та сервісних метрик, зокрема HPA, Prometheus Adapter, Metrics Server;
- обмеження використання запитів PromQL напряму в компоненті масштабування як цільової метрики, кожна сервісна метрика на базі PromQL запиту має бути адаптована в Prometheus Adapter перед використанням в HPA.

Отже, отримане рішення має значно полегшити процес розгортання контейнеризованих додатків в Kubernetes та налаштування автоматичного

масштабування на базі сервісних та ресурсних метрик, за рахунок повної інтеграції рішення з інфраструктурою Prometheus.

3.2 Постановка задачі

Результатом розробки має стати компонент автоматичного масштабування інтегрований для оркестратора контейнеризованих додатків Kubernetes. В якості метрик для розробленого додатку мають використовуватися PromQL запити до Prometheus без адаптування метрик в Prometheus Adapter, забезпечити повну інтеграцію компоненту з системою Prometheus.

Компонент має розгортатися в кластері за принципом «один кластер – одна репліка компоненту автоматичного масштабування». Автоматичне розгортання компоненту та всіх необхідних ресурсів Kubernetes для його роботи забезпечити за допомогою Helm чарту з можливість конфігурування параметрів компоненту.

Забезпечити підтримку наступних механізмів масштабування в компоненті:

- налаштування мінімальної та максимальної кількості реплік для кожної групи, яка керується компонентом;
- вибір PromQL запиту як цільової метрики для автоматичного масштабування, відповідно й вибір порогового значення для метрики для кожної групи, яка керується компонентом;
- налаштування часового інтервалу обстеження значень метрики, в якому мають виявлятися тренди та аномалії для подальших рішень щодо масштабування;
- механізм стабілізації кількості реплік після виконання рішення щодо масштабування, який блокуватиме будь-які нові прийняті рішення компоненту певний налаштований проміжок часу.

Пошук та синхронізація налаштувань груп реплік, які підлягають масштабуванню компонентом, має відбуватися автоматично в межах кластеру, в якому розгорнуто компонент. Стан знайдених груп реплік має відслідковуватися

і регулюватися компонентом автоматично та незалежно від інших груп одразу після виконання пошуку.

Для розгортання груп реплік, які будуть масштабуватися компонентом, запровадити універсальний Helm чарт. Додатково передбачити кращі практики оркестрації Kubernetes в реалізованому чарті.

3.3 Архітектура компоненту автоматичного масштабування

Компонент автоматичного масштабування для Kubernetes реалізовано на мові програмування Java 21 з використанням фреймворку Spring Boot 3, що є сучасним стандартом. Вибір Java 21 обґрунтовується потребами обробки великої кількості паралельних задач і збором метрик в реальному часі.

Для під’єднання до Kubernetes кластеру, в якому розгортається компонент, використано офіційну бібліотеку «io.kubernetes.client-java». Бібліотека надає досконалий клієнт для взаємодії з Kubernetes API, підтримуючи різні типи авторизації в кластері, а також модифікування, створення та видалення будь-яких ресурсів доступних в Kubernetes, таких як Deployment, Pod, ConfigMap, і пропонуючи синхронні та асинхронні варіанти взаємодії.

Загальна архітектура компоненту складається з чотирьох вертикально розміщених модулів, які обмінюються повідомленнями:

- модуль пошуку цільових груп реплік, які мають бути масштабованими, та їхньої конфігурації;
- модуль моніторингу значення метрик;
- модуль розрахунку актуальної кількості реплік, базуючись на значеннях метрики за останній цикл моніторингу;
- модуль прийняття рішень щодо масштабування.

Обмін повідомленнями між модулями реалізовано за допомогою патерну Message Queue (черга повідомлень) [72] на базі інтерфейсу BlockingQueue [73] з Java SDK. Архітектура компоненту передбачає наступний перелік черг для коректного функціонування:

– черга MonitoredDeploymentEventQueue містить повідомлення про створення, оновлення чи видалення цільових груп метрик, які підлягають до автоматичного масштабування компонентом;
– черга MetricCycleEventQueue містить повідомлення про останній цикл моніторингу значень метрик для цільової групи реплік;
– черга ScalingDecisionEvent містить повідомлення про рішення прийняті щодо масштабування кількості реплік в цільовій групі.

Загальна архітектура компоненту та взаємозв'язків між модулями та сторонніми системи зображена на рисунку 3.1.

Рисунок 3.1 – Діаграма загальної архітектури компоненту

3.4 Конфігурування параметрів масштабування для груп метрик

Однією з вимог до компоненту автоматичного масштабування є підтримка індивідуального конфігурування значень параметрів для кожної групи реплік контейнеризованого додатку, тобто кожна група реплік, що розгорнута в кластері, повинна мати власний розгорнутий ресурс з конфігураціями, який буде використовувати компонент під час масштабування саме цієї групи реплік.

Визначено наступний перелік конфігураційних параметрів для груп реплік, які впливають на поведінку компоненту автоматичного масштабування:

- мінімальна та максимальна кількість реплік допустима для цільової групи;
- цільова метрика, яка є показником стану цільової групи реплік, у вигляді запиту PromQL до Prometheus;
- порогове значення цільової метрики;
- тривалість циклу збору метрик в секундах;
- тривалість стабілізаційного періоду в секундах.

Для зберігання наведених конфігураційних параметрів масштабування було обрано ресурс ConfigMap, який є стандартним об’єктом Kubernetes для збереження даних в форматі ключ-значення. Ресурс шаблонізовано та додано до універсального Helm чарту, який використовується для розгортання додатків.

Шаблонізований ресурс ConfigMap з конфігурацією масштабування:

```
{{- if .Values.autoscaling.enabled}}
apiVersion: v1
kind: ConfigMap
metadata:
  name: {{include "service.fullname" .}}-prometheus-autoscaling-
config
  labels:
    {{- include "service.selectorLabels" . | nindent 4}}
data:
  min-pods: "{{.Values.autoscaling.minReplicas}}"
  max-pods: "{{.Values.autoscaling.maxReplicas}}"
  prometheus-query: "{{.Values.autoscaling.prometheusQuery}}"
  threshold: "{{.Values.autoscaling.threshold}}"
  metric-window-seconds:
"{{.Values.autoscaling.policy.metricWindowSeconds}}"
```

```
  cooldown-seconds: "{{.Values.autoscaling.policy.cooldownSeconds}}"
{{end}}
```

Іншою вирішеною проблемою за допомогою розширення Helm чарту є ідентифікація груп реплік, які мають бути масштабованими. Для цього було розширено шаблон ресурсу Deployment анотацією, яка означає, що група реплік, якою керує Deployment, є цільовою групою для компоненту масштабування.

Розширення ресурсу Deployment міткою-анотацією:

```
{{- if and .Values.autoscaling.enabled}}
annotations:
  prometheus-autoscaler-target: "true"
{{end}}
```

Таким чином, для додатків з увімкненою функцію автоматичного масштабування («autoscaling.enabled: true»), під час розгортання буде автоматично створено ресурс ConfigMap, в якому будуть заповнені значення для всіх передбачених конфігураційних параметрів, а сутність Deployment буде проанотована як ціль масштабування.

3.5 Авторизація в Kubernetes API для взаємодії з ресурсами кластеру

Для виконання запитів до Kubernetes API, розгорнутий компонент автоматичного масштабування мусить мати відповідні права доступу в інфраструктурі Kubernetes.

Опис і надання прав доступу до ресурсів Kubernetes додаткам, які розгорнуті в кластері, здійснюється за допомогою створення інших ресурсів, які є частиною безпекової інфраструктури Kubernetes, а саме ServiceAccount [74], ClusterRole [75] та ClusterRoleBinding [76].

Ресурс ServiceAccount представляє певний обліковий запис додатку в Kubernetes, що дозволяє ідентифікувати розгорнуті репліки та їхні права доступу до ресурсів в кластері.

Ресурс ClusterRole презентує певну роль в системі та визначає права доступу цієї ролі, такі як права на читання даних про репліки, регулювання

кількості реплік тощо. Для призначення ClusterRole до певного ServiceAccount використовується сутність ClusterRoleBinding.

Відповідно до потреб інфраструктури Kubernetes, перелічені ресурси були створені та додані до Helm чарту компоненту. Визначені наступні права доступу:

– зчитування та оновлення інформації про ресурси типу Deployment;
– зчитування інформації про ресурсу типу ConfigMap.

Після розгортання відповідних ресурсів в кластері, авторизація в Java клієнті Kubernetes відбувається автоматично, так як кожна репліка асоційована з відповідним ServiceAccount, який надає необхідні API токени та сертифікати TLS для взаємодії з Kubernetes API.

3.6 Реалізація модулю пошуку цільових груп реплік

В структурі проєкту всі Java-класи, які відносяться до цього модулю, знаходяться в директорії «scanning», ключові класи модулю наступні: MonitoredDeploymentConfigurationComposer, MonitoredDeploymentService, MonitoredDeploymentStorage та MonitoredDeploymentScanner.

Клас MonitoredDeploymentScanner відповідальний за сканування кластеру Kubernetes з метою пошуку цільових груп реплік та їхньої конфігурації. При кожному скануванні клас отримує дані про всі розгорнуті в кластері ресурси типу Deployment та ConfigMap. Наступним кроком є відсіювання ресурсів Deployment, які не проанотовані міткою «prometheus-autoscaler-target» зі значенням «true», тобто вони не являються цільовими для компоненту і над ними не мають виконуватися операції масштабування.

Далі для знайдених цільових ресурсів типу Deployment виконується пошук конфігурації автоматичного масштабування, тобто відповідного ConfigMap. Для цього використовуються метод співставлення міток метаданих обох ресурсів. Перелік міток, які використовуються для співставлення, можна конфігурувати під час розгортання компоненту.

Після того, як для всіх ресурсів Deployment було знайдено відповідну ConfigMap, пари надходять до класу

MonitoredDeploymentConfigurationComposer, який відповідальний за валідацію конфігураційних даних та їхнє перетворення в головний доменний об’єкт компоненту – MonitoredDeploymentConfiguration.

Об’єкти типу MonitoredDeploymentConfiguration надходять до класу MonitoredDeploymentService, який синхронізує вхідні дані з вже збереженими об’єктами. Процес синхронізації має на меті визначити чи змінилася наявна конфігурація масштабування та відправити відповідне повідомлення до черги MonitoredDeploymentEventQueue, щодо зміну стану:

- у випадку, якщо конфігурація цільової групи реплік не була знайдена серед наявних, MonitoredDeploymentService відправить повідомлення з типом «Created» і збереже дані до MonitoredDeploymentService;
- у випадку, якщо конфігурація цільової групи реплік була знайдена серед наявних, проте вона була оновлена, буде відправлено повідомлення з типом «Updated» і виконано оновлення даних;
- якщо наявна конфігурація не відрізняється від отриманої під час сканування, не буде відправлено жодного повідомлення до черги, так як стан системи не було змінено;
- для випадків, в яких об’єкт MonitoredDeploymentConfiguration не був отриманий під час сканування, проте він був раніше збережений компонентом, відправляється повідомлення з типом «Removed» та виконується видалення даних об’єкту.

Дані про знайдені конфігурації масштабування зберігаються в оперативній пам’яті компоненту за допомогою класу MonitoredDeploymentStorage. Так як компонент розгортається за принципом «один кластер – одна репліка компоненту», не існує потреби у використанні розподіленої пам’яті, наприклад реляційної бази даних, Redis, Memcached тощо.

Інтервал сканування кластеру за замовченням становить 30 секунд і є достатнім для актуалізації даних, проте цей параметр може бути змінений під час розгортання компоненту.

Таким чином цей модуль досягає повної синхронізації даних розгорнутих додатків в кластері, що потребують масштабування, з компонентом та сигналізує інші модулі у випадках зміни стану, що забезпечує консистентність даних та надійну роботу.

3.7 Реалізація модулю моніторингу

В структурі проєкту модуль займає директорію «scaling/observer», основа модулю складається з наступних Java-класів: MetricObservationProcessor та MetricObserverTask.

Точкою входу в модуль є клас-процесор MetricObservationProcessor, він прослуховує повідомлення про зміну стану конфігураційних об’єктів цільових груп реплік з черги MonitoredDeploymentEventQueue у паралельному режимі. Фіксована базова кількість потоків прослуховування повідомлень - 3, вона може бути змінена під час розгортання проєкту для збільшення або зменшення пропускної здатності.

Головним функціоналом модулю є створення паралельної асинхронної задачі моніторингу метрик, MetricObserverTask, для кожної цільової групи реплік. Тобто модуль має підтримувати такий стан компоненту, щоб значення метрик для кожної цільової групи відстежувалися незалежно від інших в окремому потоці з власною конфігурацію.

Управління асинхронними задачами MetricObserverTask відбувається на базі отриманих повідомлень з черги MonitoredDeploymentEventQueue:

- при отримані повідомлення з типом «Created», MetricObservationProcessor створить паралельну задачу для нової групи реплік, відслідковування значень метрик для групи почнеться миттєво після додавання задачі;
- при отриманні повідомлення з типом «Updated», клас-процесор видалить вже існуючу задачу, якщо вона існує, та створить нову з оновленою конфігурацією;

– відповідно при отриманні повідомлення з типом «Removed», клас-процесор зупинить та видалить існуючу задачу, метрики більше не будуть відстежуватися для цієї групи реплік.

Ключовою відмінністю розробленого компоненту від існуючих аналогів є збір значень цільових метрик у циклах, а не лише дійсного останнього показника.

Цикл представляє собою серію метричних даних, зібраних протягом фіксованих інтервалів часу. Інтервал збору метрик в циклі конфігурується на рівні компоненту, значення за замовченням – 10 секунд. Тривалість циклу є специфічною для кожної цільової групи реплік. Тобто, якщо для цільової групи налаштована тривалість циклу збору метрик 30 секунд, то один цикл буде складатися з трьох останніх значень метрик, а інтервали між отриманням цих значень дорівнюють 10 секундам.

Збір значень метрик відбувається за допомогою клієнту Prometheus, тобто компонент напряму взаємодіє з Prometheus API використовуючи PromQL запит налаштований в конфігурації групи реплік як цільову метрику.

Кожна асинхронна задача MetricObserverTask має власний контекст виконання з інформацією про 5 останніх отриманих циклів для групи реплік та дійсний цикл, який знаходиться в процесі відстеження. При закінченні дійсного циклу збору, задача відправляє повідомлення в чергу MetricCycleEventQueue, яке містить інформацію про цільову групи реплік та останній цикл отриманих значень метрик.

Архітектура класів модулю моніторингу зображена на рисунку 3.2. На діаграмі розглядається процес комунікації основних сутностей між собою та їхній устрій в модулі.

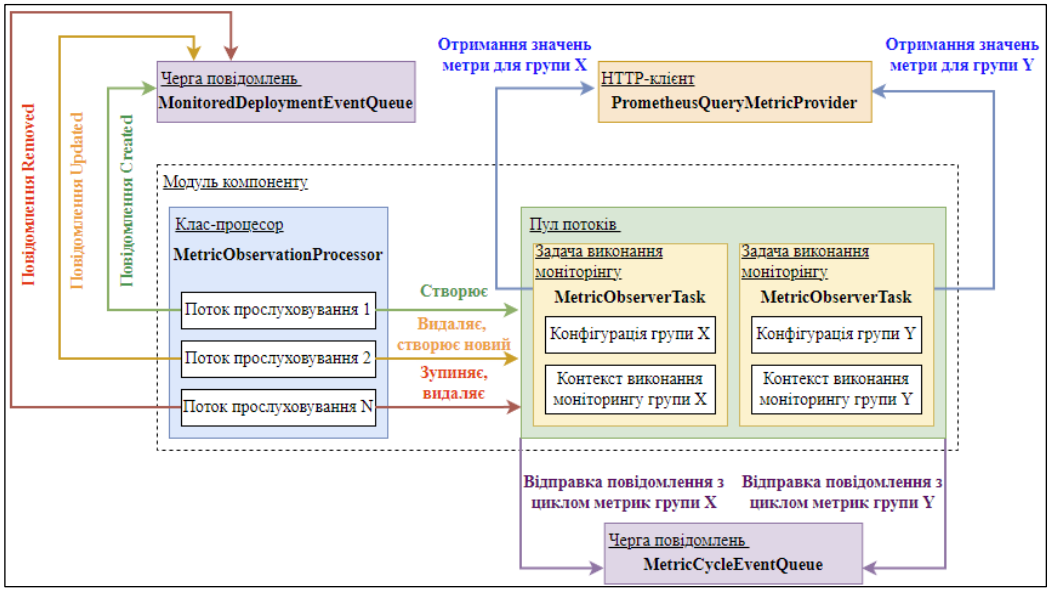

Рисунок 3.2 – Архітектура модулю моніторингу

3.8 Реалізація модулю розрахунку необхідної кількості реплік

Модуль знаходиться в директорії «scaling/decision» та складається лише з одного класу – ScalingDesicionProcessor.

Головною метою цього класу є розрахунок необхідної кількості реплік цільової групи на базі повідомлень з черги MetricCycleEventQueue, тобто на основі даних про останній цикл збору значень метрик. Прослуховування повідомлень реалізовано в такому ж стилі, що й в модулі моніторингу: повідомлення обробляються паралельно, кількість потоків обробки за замовченням – 3.

При старті обробки повідомлення, клас-процесор отримує дійсну кількість розгорнутих реплік цільової групи за допомогою клієнта Kubernetes. Наступним кроком є розрахунок кількості реплік на базі отриманого циклу значень метрики. Формула для розрахунку кількості реплік виглядає наступним чином:

$$P_t = \min\left(\max\left(\left\lceil P_c \times \frac{M_a}{M_t} \right\rceil, P_{min}\right), P_{max}\right) \quad (3.1)$$

де M_a – середнє арифметичне значення метрики для останнього циклу,

M_t – порогове значення метрики,

P_{min} – мінімальна кількість реплік,

P_{max} – максимальна кількість реплік,

P_c – дійсна кількість реплік,

P_t – цільова кількість реплік.

Як можна побачити з формули 3.1, в якості показника дійсного значення метрики (M_a) використовується середнє арифметичне значення останнього циклу обстежень, що зменшує вплив випадкових коливань або аномалій. Це є основною відмінністю від формули 2.1, яка використовується в ресурсі Kubernetes HPA.

Аргументи P_{min} та P_{max} використовуються для контролю мінімальної та максимальної кількості реплік заданою конфігурацію масштабування групи.

Розраховане значення завжди округляється до найбільшого цілого, щоб запобігти недостатньої кількості ресурсів у випадку, коли середнє значення метрики перебуває близько порогу.

У випадку, якщо дійсна кількість метрик не збігається з розрахованою, ScalingDesicionProcessor відправляє повідомлення в чергу ScalingDecisionEvent з новою розрахованою кількістю реплік для цільової групи. В іншому випадку, розгорнута кількість реплік відповідає дійсному навантаженню і дій не потрібно.

3.9 Реалізація модулю прийняття рішень масштабування

Модуль прийняття рішень знаходиться в директорії «scaler/applier» та складається з одного класу-процесору: DecisionApplierProcessor.

Так само як і інші класи-процесори в компоненті, DecisionApplierProcessor обробляє повідомлення з черги ScalingDecisionEvent в паралельному стилі з використанням трьох потоків за замовченням. Процесор має на меті прийняти рішення щодо масштабування цільової групи на основі нової розрахованої оптимальної кількості реплік.

Механізм прийняття рішення базується на концепції Cooldown Period (період стабілізації). Цей механізм запобігає надмірному або занадто частому

масштабуванні кількості реплік, що призводить до нестабільної роботи системи та неефективному використанню ресурсів.

Після кожної операції масштабування цільовій групі реплік потрібен час для стабілізації, це дозволяє їй адаптуватися до нової кількості розгорнутих ресурсів, тож період стабілізації є корисним у випадках, коли значення метрики можуть швидко змінюватися від суттєвих коливань навантаження.

Період стабілізації реалізовано за допомогою співставлення часу виконання останньої операції масштабування для цільової групи реплік з параметром конфігурації групи «cooldownSeconds», тож для підтримки роботи механізму процесор зберігає інформацію про останню виконану операцію кожної групи реплік після її успішного виконання.

3.10 Універсальний шаблон розгортання контейнеризованих додатків

Універсальний шаблон розгортання контейнеризованих додатків в Kubernetes реалізовано на основі Helm чарту. Шаблон є необхідним елементом розробленого компоненту масштабування, бо він автоматично налаштовує та розгортає всі необхідні ресурси в кластері для коректної роботи компоненту з групою реплік.

Базою універсального шаблону є маніфести ресурсів Kubernetes, які були досліджені в минулому розділі, тобто після розгортання додатку за допомогою шаблону, розгорнута група реплік буде підтримувати основні механізми Kubernetes (механізм Health Check, стратегії оновлення реплік, балансування HTTP-трафіку між репліками тощо), збір метрик системою Prometheus та масштабування розробленим компонентом.

До розробленого універсального шаблону увійшли маніфести наступних ресурсів Kubernetes:

– маніфест ресурсу Deployment, який контролює життєвий цикл групи реплік, та надає Kubernetes налаштування механізму Health Check, стратегії оновлення та помічається як цільовий ресурс для компоненту

автоматичного масштабування у разі, якщо для додатку, який розгортається, функція увімкнена;

- маніфест користувацького ресурсу ServiceMonitor надає інформацію системі Prometheus про порти та кінцеві точки, з яких мають збиратися метрики;
- маніфест ресурсу LoadBalancer конфігурує балансування запитів між репліками групи та надає порти для доступу до них ззовні та зсередини кластера;
- маніфест ресурсу ConfigMap використовується як шаблон конфігурації для параметрів автоматичного масштабування групи реплік, які будуть використані розробленим компонентом.

Таким чином, розгортання додатку універсальним шаблоном в комбінації з розробленим компонентом автоматичного масштабування створює комплексну інфраструктуру, яка є надійною під час різних умов експлуатації.

4 ОПИС ЕКСПЕРИМЕНТАЛЬНИХ ДОСЛІДЖЕНЬ

1.1 Мета дослідження

Метою дослідження є визначення найоптимальнішого методу автоматичного масштабування, який здатен забезпечити максимальну доступність та продуктивність контейнеризованих застосунків у різних експлуатаційних умовах, за допомогою якісного порівняння різних підходів в оркестраторі контейнерів Kubernetes.

Іншою метою є перевірка та тестування розробленої універсальної моделі розгортання та компоненту автоматичного масштабування відносно існуючих підходів. Порівняльна оцінка розробленого компоненту є особливо важливою, оскільки вона вносить вклад у розуміння взаємодії нових та традиційних методів в сфері масштабування контейнеризованих додатків.

4.1.1 Огляд існуючих досліджень методів масштабування в Kubernetes

Тема досліджень існуючих методів масштабування методом якісного порівняння не є новою, проте є досі актуальною, так як кожного року з’являються нові технології в цій сфері.

В науковій статті «Horizontal Pod Autoscaling in Kubernetes for Elastic Container Orchestration» [77] за 2020 рік порівнюються два основних підходи автоматичного масштабування – метод з використанням ресурсних метрик та метод з використанням сервісних.

Метою цього дослідження було виявлення залежності реакції на зміну навантаження в Kubernetes HPA від налаштованого періоду збору метрик, джерела метрик та кількості вузлів в кластері.

В результаті дослідження встановлено, що частота збору метрик суттєво впливає на реакцію компоненту масштабування HPA: встановлення коротших періодів призводить до покращення швидкості реакції на зміни навантаження, але можуть викликати збільшення кількості зайвих операцій масштабування. Також встановлено, що використання сервісних метрик замість ресурсних дозволяє компоненту HPA набагато швидше реагувати на зміни навантаження.

4.2 Постановка експерименту

Всі експериментальні дослідження виконуються в локально розгорнутому кластері Kubernetes з одним обчислювальним вузлом, що являється фізичним сервером з наступними характеристиками:

- центральний процесор AMD 5800X3D, 8 ядр, частота оновлення 4.5 ГГц;
- оперативна пам’ять 32ГБ, частота оновлення 3200 МГц;
- оперативна система Windows 11;
- рушій контейнеризації Docker, версія 25.0.3;
- оркестратор контейнерів Kubernetes, версія 1.29.1.

В ході дослідження будуть порівнюватися якісні характеристики наступних методів автоматичного масштабування:

- традиційний метод автоматичного масштабування на базі HPA та використання ресурсних метрик наданих додатком Metrics Server (метод детально описаний у розділі 2.8);
- традиційний метод автоматичного масштабування на базі HPA та використання ресурсних та сервісних метрик зібраних та наданих Prometheus (метод детально описаний у розділі 2.9);
- розроблений компонент автоматичного масштабування з використанням ресурсних та сервісних метрик зібраних та наданих Prometheus (розробка та деталі методу детально описані у розділах 3.3-3.9).

В якості застосунку для масштабування використано тестовий додаток реалізований на мові програмування Java 21 з використанням Spring Boot 3. Додаток реалізовано так, що він інтенсивно використовує ресурс центрального процесору при обробці кожного HTTP-запиту. Штучне навантаження створено за допомогою хешування великого масиву даних алгоритмом SHA-256.

Інфраструктура тестового додатку розгортається в кластері Kubernetes за допомогою розробленого універсального шаблону, що забезпечує балансування навантаження між репліками застосунку. Для кожної репліки надано наступні

ліміти використання центрального процесору: запит – 100 міліядр, ліміт – 150 міліядр.

Штучне навантаження на групу реплік генерується за допомогою інструменту Apache JMeter [78]. Генерація навантаження на групу реплік виконується за допомогою надсилання запитів на кінцеву точку HTTP, при обробці яких репліки будуть інтенсивно навантажувати центральний процесор. Навантаження генерується в 2 етапи:

– 10000 запитів на хвилину протягом 10 хвилин (сумарно 100,000 запитів);
– 6000 запитів на хвилину протягом 10 хвилин (сумарно 60,000 запитів).

Визначені наступні ліміти мінімальної та максимальної допустимої кількості реплік під час проведення експериментів:

– мінімальна кількість реплік – 4;
– максимальна кількість реплік – 24.

Вихідні дані кожного експерименту, які використовуються для якісного порівняння методів масштабування, наступні:

– кількість реплік, створених під час генерування штучного масштабування відносно етапу інтенсивності навантаження;
– час, за який компонент автоматичного масштабування адаптував кількість реплік до дійсного навантаження;
– сумарний час середнього навантаження на процесор за межами порогового значення метрики.

Періодом збору метрик за замовчення для традиційних методів масштабування, які використовують HPA, є 60 секунд.

Розроблений компонент автоматичного масштабування використовує концепцію циклів збору метрик, тому значенням періоду збору метрик за замовчення для нього буде цикл з 6 обстежень з інтервалом в 10 секунд.

Період стабілізації є однаковим для методів з використанням HPA та розробленого компоненту – 90 секунд.

Пороговою метрикою для автоматичного масштабування для всіх методів є середнє використання центрального процесору більше за 90% від наданих ресурсів.

4.3 Проведення експериментальних досліджень

4.3.1 Традиційний метод з використанням HPA та Metrics Server

Після постановки експерименту для традиційного методу масштабування з використанням HPA та Metrics Server, отримані дані про стан групи реплік (див. рис. 4.1) та середнє значення метрики використання процесору (див. рис. 4.2).

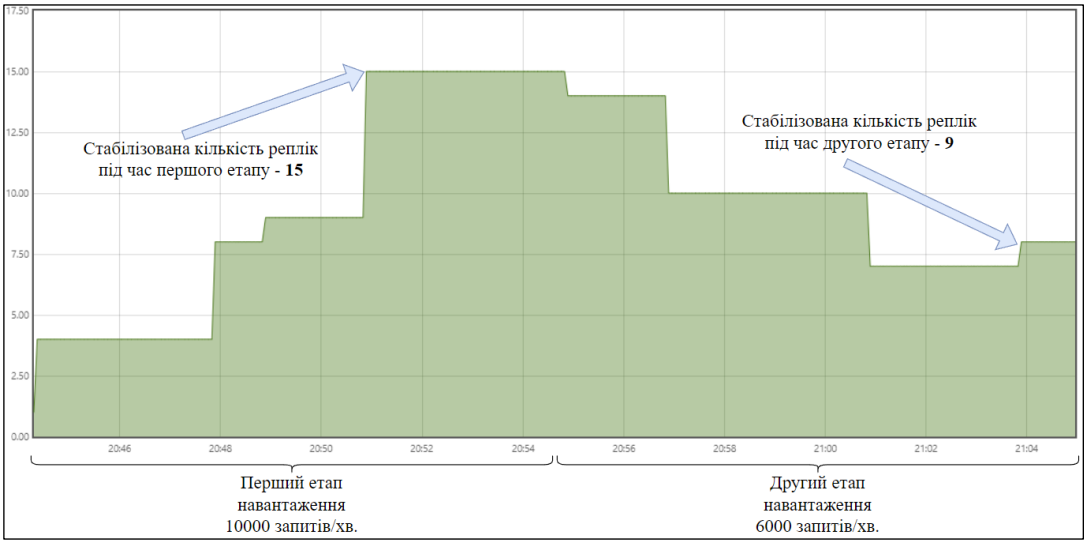

Рисунок 4.1 – Стан групи реплік для першого експерименту

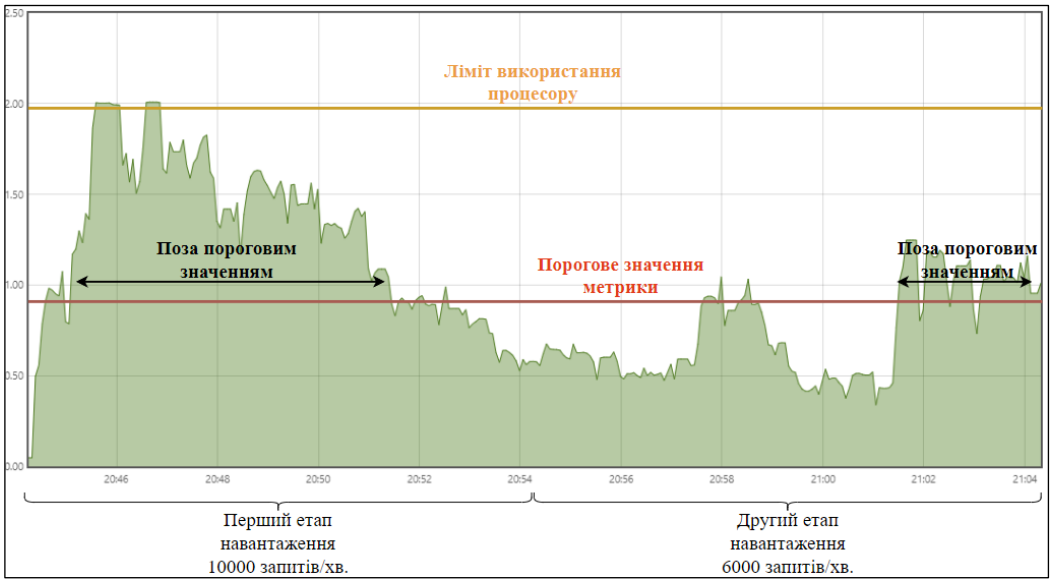

Рисунок 4.2 – Середнє значення цільової метрики для першого експерименту

Під час першого етапу навантаження, ресурс HPA збільшив кількість реплік тестового додатку 3 рази стабілізувавшись на 15 репліках (4 → 8 → 9 → 15) , далі кількість реплік залишалася стабільною. Для повної стабілізації на цьому етапі експерименту компоненту автоматичного масштабування знадобилося 425 секунд, при цьому загально 63 секунд середнє значення порогової метрики знаходилося на ліміті використання ресурсів (200 міліядр), що може призвести до зупинки реплік оркестратором.

На другому етапі експерименту, компонент HPA спочатку поступово зменшив кількість реплік до 7 за 3 операції масштабування та збільшив до 8 стабілізувавшись на цьому значенні (15 → 14 → 10 → 7 → 8). Загалом на цьому етапі середнє значення метрики перебувало поза порогом 211 секунд, проте не знаходячись в критичному стані.

Відсоток запитів, які були оброблені репліками тестового додатку під час навантаження мінімальна – 0.078%.

4.3.2 Традиційний метод з використанням HPA та Prometheus

Під час постановки експерименту для методу з використанням HPA та Prometheus, отримані наступні дані щодо стану реплік (див. рис. 4.3) та середнього значення цільової метрики (див. рис. 4.4).

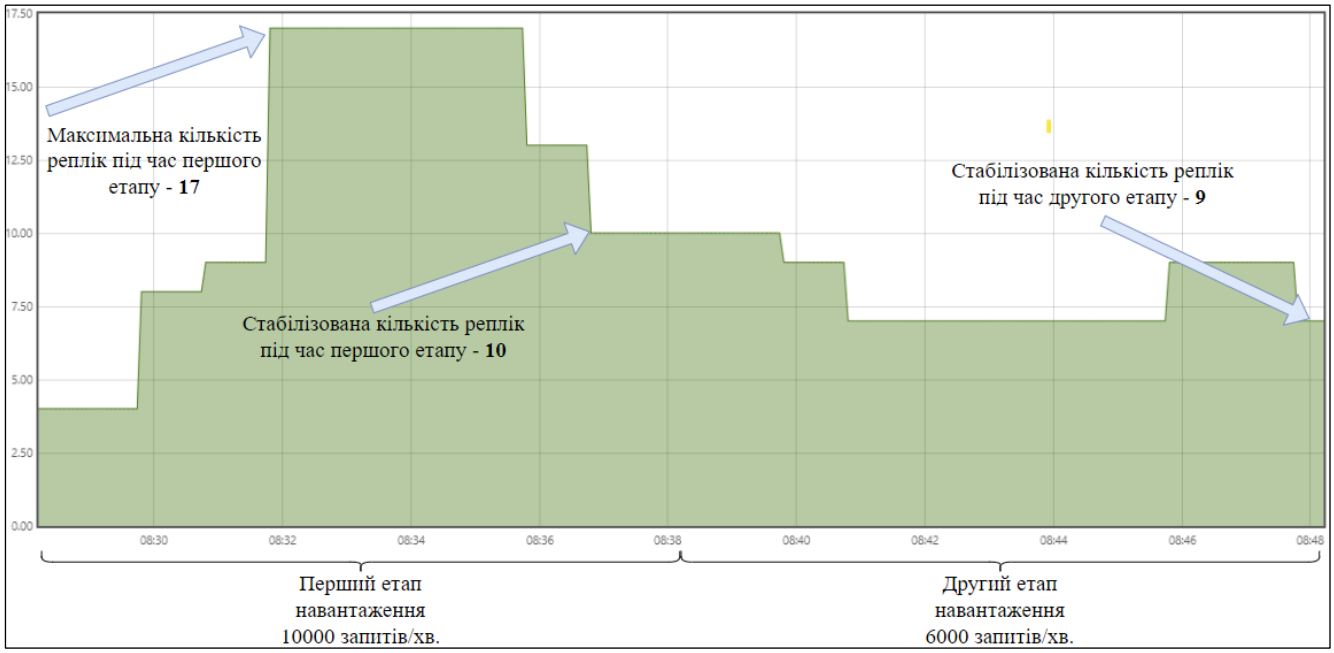

Рисунок 4.3 – Стан групи реплік для другого експерименту

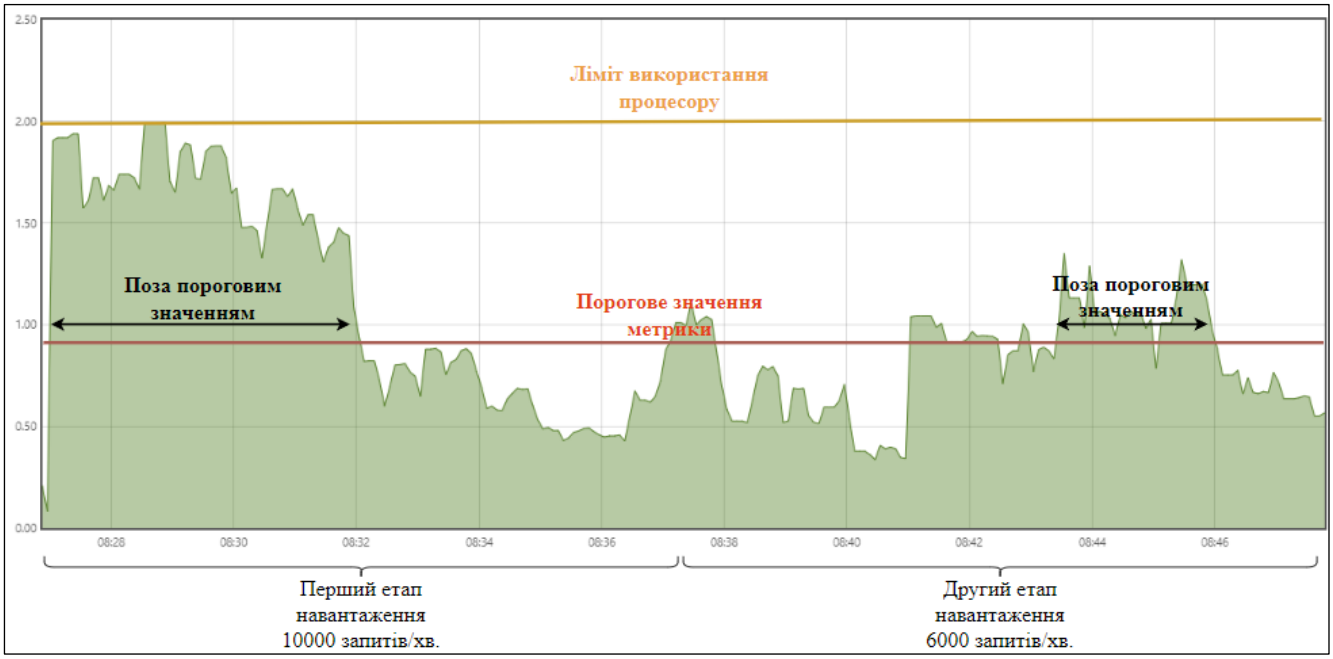

Рисунок 4.4 – Середнє значення цільової метрики для другого експерименту

Під час першого етапу навантаження, ресурс HPA поступово збільшив кількість реплік тестового додатку 3 рази та значно зменшив 2 рази стабілізувавшись на 10 репліках (4 → 8 → 9 → 17 → 13 → 10). Для стабілізації кількості реплік до значення, яке відповідає навантаженню, компоненту HPA в комбінації з метриками Prometheus та Prometheus Adapter знадобилося 297 секунд, середнє значення використання процесору репліками на рівні ліміту наданих ресурсів знаходилось протягом 20 секунд.

На другому етапі експерименту, кількість реплік було поступово зменшено до 7 за 2 операції масштабування та збільшено до 9 за одну операцію, стабілізувавшись на цьому значенні (10 → 9→ 7 → 9). Середнє значення метрики використання процесору знаходилось за порогом протягом 147 секунд.

Як і під час постановки першого експерименту, відсоток невдало оброблених запитів мінімальний під час обох етапів – 0.106%.

В обох традиційних методах відсоток невдало оброблених запитів виявився незначним, що свідчить про високу ефективність цих методів. Однак, стратегія з використанням Prometheus та Prometheus Adapter показала кращу

швидкість адаптації до змін у навантаженні та менший час перебування групи реплік в зоні надвисокого навантаження.

4.3.3 Розроблений компонент масштабування на базі метрик Prometheus

Під час постановки третього експерименту з використанням розробленого компоненту автоматичного масштабування, який базується на сервісних та ресурсних метриках Prometheus без використання Prometheus Adapter, отримані наступні дані щодо стану реплік (див. рис. 4.5) та середнього значення цільової метрики (див. рис. 4.6).

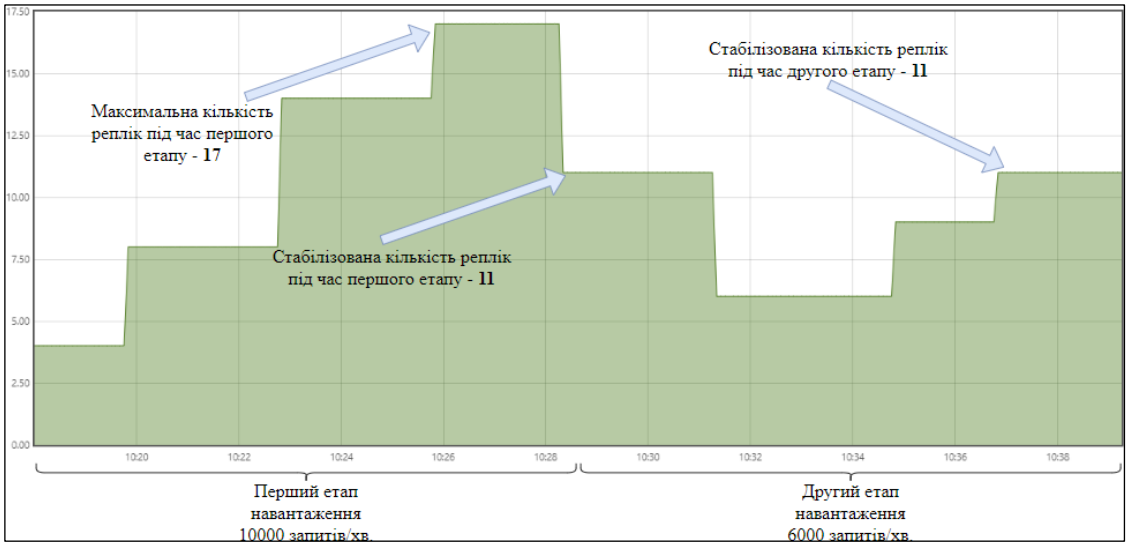

Рисунок 4.5 – Стан групи реплік для третього експерименту

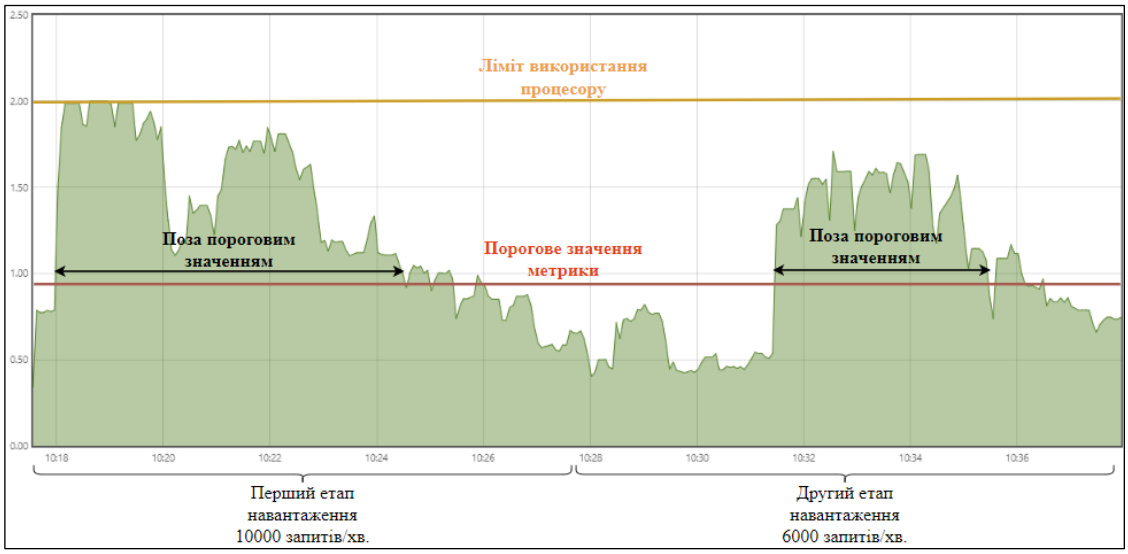

Рисунок 4.6 – Середнє значення цільової метрики для третього експерименту

Під час першого етапу навантаження, розроблений компонент поступово збільшив кількість реплік тестового додатку 3 рази та значно зменшив один раз стабілізувавши групу реплік на 11 репліках (4 → 8 → 14 → 17 → 11). Схожий патерн регулювання кількості реплік можна помітити в другому експерименті, де використовується HPA з метриками Prometheus та Prometheus Adapter. Для стабілізації кількості реплік до значення, яке відповідає навантаженню, розробленому компоненту знадобилося 306 секунд, середнє значення використання процесору репліками на рівні ліміту знаходилось 49 секунд.

На другому етапі експерименту, кількість реплік було спочатку зменшено до 6 за одну операцію та збільшено до 11 за дві операції (11 → 6→ 9 → 11). Середнє значення метрики використання процесору знаходилось за порогом протягом 212 секунд.

4.4 Результати дослідження

На основі отриманих даних складено порівняльну таблицю досліджених методів та інструментів автоматичного масштабування (див. табл. 4.1).

Таблиця 4.1 – Порівняння досліджених методів масштабування в Kubernetes

Критерій порівняння	HPA, Metrics Server	HPA, Prometheus, Prometheus Adapter	Розроблений компонент масштабування
Перший етап навантаження (різке високе навантаження)			
Кількість операцій для повної стабілізації	3	5	4
Максимальна кількість реплік	15	17	17
Стабілізована кількість реплік	15	10	11
Час поза пороговим значенням (секунд)	425 с.	297 с.	306 с.
Час в зоні надвисокого	63 с.	20 с.	49 с.

Кінець таблиці 4.1

Критерій порівняння	HPA, Metrics Server	HPA, Prometheus, Prometheus Adapter	Розроблений компонент масштабування
навантаження (секунд)			
Другий етап навантаження (середнє навантаження після високого)			
Кількість операцій для повної стабілізації	4	3	3
Максимальна кількість реплік	14	9	11
Стабілізована кількість реплік	8	9	11
Час поза пороговим значенням (секунд)	211 с.	147 с.	212 с.
Загальні показники			
Відсоток невдалих запитів до реплік	0.078%	0.106%	0.056%

Розроблений компонент на базі метрик Prometheus демонструє час адаптації на першому етапі навантаження порівняний з традиційним методом HPA і Prometheus (306 секунд проти 297), і є значно швидшим за метод з HPA та Metrics Server, що свідчить про ефективність розробленого компоненту в реагуванні на зміну навантаження. Проте показник у зоні надвисокого навантаження є набагато гіршим за метод HPA з Prometheus (49 секунд проти 20), що є індикатором необхідності оптимізації реалізованого компоненту.

Використання циклу значень метрик у розробленого компоненту масштабування замість останнього значення як в традиційних методах, напряму впливає на стабілізовану кількість реплік. Це значення є дещо більшим відносно традиційних методів на другому етапі навантаження (11 реплік проти 8 та 9), що знижує ризик недостатньої кількості ресурсів, але збільшує рівень їх використання.

Механізм стабілізації після виконання операції масштабування відпрацьовує майже однаково для методів, що досліджуються – сумарна

кількість виконаних операцій масштабування знаходиться в діапазоні від 7 до 8, що є незначною різницею.

Загальний відсоток невдало виконаних запитів до реплік під час навантаження є найнижчим у розробленого компоненту, проте ця різниця є незначною відносно показників інших методів масштабування. Порівнюючи по цьому критерію, всі представники продемонстрували високу надійність, так як найбільшим відсоток невдалих запитів є 0.106% (169 запитів з 160,000).

Отже, компонент автоматичного масштабування на базі метрик з Prometheus для оркестратору Kubernetes продемонстрував конкурентоспроможність, проте він потребує подальшої оптимізації та розширення функціональних можливостей для значного випередження традиційних методів.

ВИСНОВКИ

В кваліфікаційній роботі розглянуто методи контейнеризації на прикладі тестового додатку, детально розглянуті особливості архітектури сучасного рушія контейнеризації Docker, його компоненти та процес взаємодії між ними.

Детально досліджені механізми внутрішньої роботи оркестратора Kubernetes, такий як вибір обчислювального вузла для розгортання реплік, та найкращі практики оркестрування контейнеризованих додатків – постійна перевірка стану реплік, динамічна маршрутизація запитів та стратегії оновлення.

Розглянуто два традиційних концептуально різних метода масштабування реплік в Kubernetes – на базі ресурсних метрик з Metrics Server та сервісних метрик з Prometheus. Знайдено переваги та недоліки кожного методу. Об’єднання найкращих аспектів обох методів послугувало основою для створення власного компоненту автоматичного масштабування для Kubernetes.

В роботі детально описано механізми створеного компоненту автоматичного масштабування, його архітектуру, варіанти розгортання в кластері Kubernetes та процес інтегрування груп реплік з ним.

Виведено власну концепцію збору метрик для масштабування та інтегровано її в розробленому компоненті. Збирання циклів метрик замість використання останнього отриманого значення, як в традиційних методах, посприяло значному зниженню впливу аномалій та коливань значень метрик на розрахунок необхідної кількості реплік.

Використання серії метричних даних замість останніх показників дозволило глибше аналізувати тенденції навантаження, що допомагає уникнути надмірного чи недостатнього розподілу ресурсів. Окрім цього, ця концепція відкриває шляхи для подальших досліджень щодо розробки алгоритмів автоматичного масштабування на базі тенденцій та динаміки навантаження.

Традиційні методи та розроблений компонент було досліджено та порівняно на практиці в рівних умовах для високого та середнього навантаження. Отримані результати дослідження вказують, що є місце оптимізації розробленого компоненту, проте в дійсному стані він є

конкурентоспроможним та може застосовуватися не тільки в контрольованих експериментальних умовах, а й у реальних проєктах з високими вимогами щодо адаптування стану ресурсів системи до змін навантаження.

Проведені дослідження методів оркестрування та створений компонент автоматичного масштабування послугували основою для розробки універсального шаблону розгортання додатків на базі технології Helm чартів. Шаблон значно спрощує процес налаштування масштабування та розгортання застосунків в кластері Kubernetes для інших розробників: для базового розгортання додатків, які будуть масштабовані, їхньої інфраструктури, та компоненту масштабування достатньо виконання двох команд в Helm CLI та створення одного централізованого конфігураційного файлу, який містить всю необхідну інформацію про застосунок, що розгортається.

Отже, проведене дослідження в ході кваліфікаційної роботи демонструє, що інтеграція сучасних підходів масштабування та оркестрування контейнеризованих додатків в розподілені інформаційні системи значного спрощує управління ними, забезпечуючи автоматизацію більшості процесів, високу надійність та адаптивність до мінливих потреб користувачів.

ПЕРЕЛІК ДЖЕРЕЛ ПОСИЛАННЯ

1. What is Containerization? – Containerization Explained – AWS. URL: https://aws.amazon.com/what-is/containerization (дата звернення 24.03.2024).

2. What is Container? | Docker. URL: https://www.docker.com/resources/what-container/ (дата звернення 25.03.2024).

3. Containerization Engines: How They Work and Top 7 Options. URL: https://www.aquasec.com/cloud-native-academy/container-platforms/container-engines/ (дата звернення 25.03.2024).

4. Namespaces(7) – Linux manual page. URL: https://man7.org/linux/man-pages/man7/namespaces.7.html (дата звернення 25.03.2024).

5. Pid_namespaces(7) – Linux manual page. URL: https://man7.org/linux/man-pages/man7/pid_namespaces.7.html (дата звернення 25.03.2024).

6. Network_namespaces(7) – Linux manual page. URL: https://man7.org/linux/man-pages/man7/network_namespaces.7.html (дата звернення 25.03.2024).

7. Mount_namespaces(7) – Linux manual page. URL: https://man7.org/linux/man-pages/man7/mount_namespaces.7.html (дата звернення 26.03.2024).

8. Ipc_namespaces(7) – Linux manual page. URL: https://man7.org/linux/man-pages/man7/ipc_namespaces.7.html (дата звернення 26.03.2024).

9. User_namespaces(7) – Linux manual page. URL: https://man7.org/linux/man-pages/man7/user_namespaces.7.html (дата звернення 26.03.2024).

10. Cgroups(7) – Linux manual page. URL: https://www.man7.org/linux/man-pages/man7/cgroups.7.html (дата звернення 26.03.2024).

11. Using cgroups to limit CPU utilization | Flaming Bytes. URL: https://www.flamingbytes.com/blog/cgroup-limit-cpu/ (дата звернення 26.03.2024).

12. Kernel.org/doc/Documentation/cgroup-v1/memory.txt (official Linux doc). URL: https://www.kernel.org/doc/Documentation/cgroup-v1/memory.txt (дата звернення 26.03.2024).

13. Our of Memory management (official Linux doc). URL: https://www.kernel.org/doc/gorman/html/understand/understand016.html (дата звернення 26.03.2024).

14. Docker Docs. URL: https://docs.docker.com/ (дата звернення 28.03.2024).

15. Docker daemon configuration overview | Docker Docs. URL: https://docs.docker.com/config/daemon/ (дата звернення 28.03.2024).

16. Use the Docker Command Line | Docker Docs. URL: https://docs.docker.com/engine/reference/commandline/cli/ (дата звернення 28.03.2024).

17. Docker Image. URL: https://www.geeksforgeeks.org/what-is-docker-image/ (дата звернення 28.03.2024).

18. Registry | Docker Docs. URL: https://docs.docker.com/registry/ (дата звернення 28.03.2024).

19. The Rocker containerization system. URL: https://lwn.net/Articles/624349/ (дата звернення 28.03.2024).

20. Singlularity. URL: https://hpc.nih.gov/apps/singularity.html (дата звернення 28.03.2024).

21. Microservice Architecture Pattern. URL: https://microservices.io/patterns/microservices.html (дата звернення 28.03.2024).

22. Eddy Truyen, A Comprehensive Feature Comparison Study of Open-Source Container Orchestration Frameworks / Eddy Truyen, Dimitri Van Landuyt, Davy Preuveneers, Bert Lagaisse, Wouter Joosen. 2019, Applied Science, Doi: 10.3390/app9050931 (дата звернення 06.04.2024).

23. Bjarne Johansson, Kubernetes Orchestration of High Availability Distributed Control Systems / Bjarne Johansson, Mats Ragberger, Thomas Nolte, Alessandro V. Papadopoulos. 2022 IEEE International Conference on Industrial Technology (ICIT), Doi: 10.1109/ICIT48603.2022.10002757 (дата звернення 02.04.2024).

24. Who We Are | CNCF. URL: https://www.cncf.io/about/who-we-are/ (дата звернення 07.04.2024).

25. Why choose AWS as your IaaS provider. URL: https://aws.amazon.com/what-is/iaas/ (дата звернення 28.03.2024).

26. O. Makieiev, Study of methods of creating service-oriented software systems in Azure / N. Kravets Computer Systems and Information Technologies, 2023 (2), Pp. 38-47. Doi: 10.31891/csit-2023-2-5 (дата звернення 28.03.2024).

27. Compute Engine | Google Cloud. URL: https://cloud.google.com/compute (дата звернення 28.03.2024).

28. Infrastructure as a Service | Cloud Native. URL: https://glossary.cncf.io/infrastructure-as-a-service/ (дата звернення 30.03.2024).

29. Cloud Compute Capacity – Amazon EC2 – AWS. URL: https://aws.amazon.com/ec2/ (дата звернення 30.03.2024).

30. Amazon EC2 M5 Instances – general purpose compute workload. URL: https://aws.amazon.com/ec2/instance-types/m5/ (дата звернення 30.03.2024);

31. Private Cloud – Amazon Virtual Private Cloud (VPC) – AWS. URL: https://aws.amazon.com/vpc/ (дата звернення 30.03.2024).

32. APM (Application Performance Monitoring) Tool – Amazon CloudWatch – AWS. URL: https://aws.amazon.com/cloudwatch/ (дата звернення 30.03.2024).

33. Шевченко Д.А., Evaluation and optimization of software product infrastructure, друк. "Журнал Wschodnioeuropejskie Czasopismo Naukowe (East European Scientific Journal), 58, ст.. 56-62, Printed in the ""Jerozolimskie 85/21, 02-001 Warsaw, Poland»" (дата звернення 30.03.2024).

34. Spring Boot Reference Documentation. URL: https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/ (дата звернення 10.04.2024).

35. Gradle User Manual. URL: https://docs.gradle.org/current/userguide/userguide.html (дата звернення 10.04.2024).

36. Docker file | Docker docs. URL: https://docs.docker.com/reference/dockerfile/ (дата звернення 12.04.2024).

37. Objects in Kubernetes | Kubernetes. URL: https://kubernetes.io/docs/concepts/overview/working-with-objects/ (дата звернення 16.04.2024).

38. Command line tool (kubectl) | Kubernetes. URL: https://kubernetes.io/docs/reference/kubectl/ (дата звернення 16.04.2024).

39. Pods | Kubernetes. URL: https://kubernetes.io/docs/concepts/workloads/pods/ (дата звернення 16.04.2024).

40. Deployments | Kubernetes. URL: https://kubernetes.io/docs/concepts/workloads/controllers/deployment/ (дата звернення 16.04.2024).

41. ReplicaSet | Kubernetes. URL: https://kubernetes.io/docs/concepts/workloads/controllers/replicaset/ (дата звернення 16.04.2024).

42. Kubernetes Scheduler | Kubernetes. URL: https://kubernetes.io/docs/concepts/scheduling-eviction/kube-scheduler/ (дата звернення 20.04.2024).

43. Node Affinity: Key Concepts, Examples, and Troubleshooting. URL: https://komodor.com/learn/node-affinity/ (дата звернення 20.04.2024).

44. Assigning Pods to Nodes | Kubernetes. URL: https://kubernetes.io/docs/concepts/scheduling-eviction/assign-pod-node/ (дата звернення 20.04.2024).

45. Taints and Tolerations | Kubernetes. URL: https://kubernetes.io/docs/concepts/scheduling-eviction/taint-and-toleration/ (дата звернення 20.04.2024).

46. Kubernetes Scheduling: Understanding the Math Behind the Magic. URL: https://romanglushach.medium.com/kubernetes-scheduling-understanding-the-math-behind-the-magic-2305b57d45b1 (дата звернення 23.04.2024).

47. Quizheng Huo, Refined Balanced Resource Allocation Based on Kubernetes / Chengyang Li, Shaonan Li, Yongqiang Xie, Zhongbo Li, Institution of Systems Engineering, 100141, China (дата звернення 23.04.2024).

48. What is Helm in Kubernetes? – Sysdig. URL: https://www.sciencedirect.com/science/article/abs/pii/S0950584918302507 (дата звернення 23.04.2024).

49. Helm | Cheat Sheet. URL: https://helm.sh/docs/intro/cheatsheet/ (дата звернення 23.04.2024).

50. Basic Guide to Kubernetes Service Discovery – Dev Community. URL: https://dev.to/nomzykush/basic-guide-to-kubernetes-service-discovery-dmd (дата звернення 26.04.2024).

51. Service | Kubernetes. URL: https://kubernetes.io/docs/concepts/services-networking/service/ (дата звернення 26.04.2024).

52. ClusterIP, NodePort and LoadBalancer | Kubernetes Services. URL: https://www.baeldung.com/ops/kubernetes-service-types (дата звернення 26.04.2024).

53. Configure Liveness, Readiness, and Startup Probes | Kubernetes. URL: https://kubernetes.io/docs/tasks/configure-pod-container/configure-liveness-readiness-startup-probes/ (дата звернення 28.04.2024).

54. Guide to Kubernetes Liveness Probe. URL: https://spacelift.io/blog/kubernetes-liveness-probe (дата звернення 28.04.2024).

55. Kubernetes Readiness Probe. URL: https://spacelift.io/blog/kubernetes-readiness-probe (дата звернення 28.04.2024).

56. Building a RESTful Web Service with Spring Boot Actuator. URL: https://spring.io/guides/gs/actuator-service (дата звернення 28.04.2024).

57. kubelet | Kubernetes. URL: https://kubernetes.io/docs/reference/command-line-tools-reference/kubelet/ (дата звернення 28.04.2024).

58. What is Vertical Scaling? | Dremio. URL: https://www.dremio.com/wiki/vertical-scaling/ (дата звернення 30.04.2024).

59. Horizontal scaling: Definitions, features and benefits. URL: https://datascientest.com/en/horizontal-scaling-definition-features-and-benefits (дата звернення 30.04.2024).

60. Horizontal Pod Autoscaler | Kubernetes. URL: https://kubernetes.io/docs/tasks/run-application/horizontal-pod-autoscale/ (дата звернення 01.05.2024).

61. Metrics Pipeline. URL: https://kubernetes.io/docs/tasks/debug/debug-cluster/resource-metrics-pipeline/#metrics-api (дата звернення 01.05.2024).

62. Kubernetes Metrics Server | metrics-server. URL: https://kubernetes-sigs.github.io/metrics-server/ (дата звернення 01.05.2024).

63. Overview | Prometheus. URL: https://prometheus.io/docs/introduction/overview/ (дата звернення 03.05.2024).

64. Дудар З., Порівняння методів прогнозування часових рядів / Широкопетлєва М.С., Пономаренко О.А., 2018 Порівняння методів прогнозування часових рядів друк. Бионика интеллекта. Харьков.: ХНУРЭ, 2018. Вип.2 (91). С.41-47 (дата звернення 03.05.2024).

65. Helm Chart Repository: kube-prometheus-stack. URL: https://artifacthub.io/packages/helm/prometheus-community/kube-prometheus-stack (дата звернення 04.05.2024).

66. Using ServiceMonitor: Observability for Prometheus. URL: https://observability.thomasriley.co.uk/prometheus/configuring-prometheus/using-service-monitors/ (дата звернення 04.05.2024).

67. Introduction | Prometheus Operator. URL: https://prometheus-operator.dev/docs/prologue/introduction/ (дата звернення 04.05.2024).

68. Monitoring Docker container metrics using cAdvisor | Prometheus. URL: https://prometheus.io/docs/guides/cadvisor/ (дата звернення 04.05.2024).

69. Micrometer Application Observability. URL: https://micrometer.io/ (дата звернення 04.05.2024).

70. Prometheus/client_java: Prometheus instrumentation library for JVM. URL: https://github.com/prometheus/client_java (дата звернення 06.05.2024).

71. GitHub – Kubernetes-sigs/Prometheus-adapter. URL: https://github.com/kubernetes-sigs/prometheus-adapter (дата звернення 06.05.2024).

72. Asynchronous Messaging Patterns | MuleSoft Blog. URL: https://blogs.mulesoft.com/migration/asynchronous-messaging-patterns/ (дата звернення 06.05.2024).

73. BlockingQueue (Java Platform SE 8). URL: https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/BlockingQueue.html (дата звернення 06.05.2024).

74. Service Accounts | Kubernetes. URL: https://kubernetes.io/docs/concepts/security/service-accounts/ (дата звернення 08.05.2024).

75. Using RBAC Authorization | Kubernetes. URL: https://kubernetes.io/docs/reference/access-authn-authz/rbac/ (дата звернення 08.05.2024).

76. Kubectl create clusterrolebinding | Kubernetes. URL: https://kubernetes.io/docs/reference/kubectl/generated/kubectl_create/kubectl_create_clusterrolebinding/ (дата звернення 08.05.2024).

77. Thanh-Tung Nguyen, Horizontal Pod Autoscaling in Kubernetes for Elastic Container Orchestration / Thanh-Tung Nguyen, Yu-Jin Yeom, Taehong Kim, Dae-Heon Park, Sehan Kim. 2020 Sensors, Doi: 10.3390/s20164621 (дата звернення 10.05.2024).

78. Apache JMeter – User’s Manual. URL: https://jmeter.apache.org/usermanual/index.html (дата звернення 18.05.2024).