Хмельницький національний університет
Факультет інформаційних технологій
Кафедра інженерії програмного забезпечення

ДИПЛОМНА РОБОТА

Метод підвищення ефективності автоматизації масштабування мікросервісів у відкритій системі автоматичного розгортання і управління контейнеризованими застосунками
Назва теми

Рівень вищої освіти Другий (магістерський)

Галузь знань 12 «Інформаційні технології»

Спеціальність 121 «Інженерія програмного забезпечення»

Освітньо-професійна програма «Інженерія програмного забезпечення»

Шифр ДРІПЗ 170108.01.03.00 ПЗ

Виконав студент II курсу група ІПЗм-21-1 _____ Я.Ю.Маєвський
Підпис Ініціали, прізвище

Керівник к. пед.наук., доцент _____ Н.І. Праворська
Науковий ступінь, звання Підпис Ініціали, прізвище

Нормоконтролер к.т.н., доцент _____ О.М. Яшина
Підпис Ініціали, прізвище

До захисту допускаю:
Завідувач кафедри інженерії
програмного забезпечення _____ Л. П. Бедратюк
Підпис Ініціали, прізвище

2 грудня 2022 р.

Хмельницький 2022

ХМЕЛЬНИЦЬКИЙ НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ

Факультет Програмування та комп'ютерних і телекомунікаційних систем

Кафедра Інженерії програмного забезпечення

Рівень вищої освіти Другий (магістерський)

Галузь знань 12 «Інформаційні технології»

Спеціальність 121 «Інженерія програмного забезпечення»

Освітньо-професійна програма «Інженерія програмного забезпечення»

ЗАТВЕРДЖУЮ
Завідувач кафедри ІПЗ
Л. П. Бедратюк
02 09 2022 р.

ЗАВДАННЯ

НА ДИПЛОМНУ РОБОТУ

Маєвському Ярославу Юрійовичу
Прізвище, ім'я, по батькові студента

1. Тема проекту (роботи) Метод підвищення ефективності автоматизації масштабування мікросервісів у відкритій системі автоматичного розгортання і управління контейнеризованими застосунками

Керівник проекту (роботи) Праворська Наталя Іванівна, к. пед.наук, доцент
Прізвище, ім'я, по батькові, науковий ступінь, вчене звання

Затверджена наказом ректора університету від 25.08.2022 р. № 102

2. Строк подання студентом проекту (роботи) на кафедру 04.12.2022 р.

3. Вихідні дані до проекту (роботи) Матеріали переддипломної практики

4. Зміст пояснювальної записки (перелік питань, які потрібно розробити) Дослідження предметної області та постановка задачі, аналіз способів віртуалізації застосунків, дослідження алгоритмі автоматичного масштабування контейнерів, дослідження і аналіз факторів, які впливають на холодний старт а також збільшують час розгортування контейнерів в системі Kubernetes, реалізація і тестування сформованого рішеня для покращення виявлених факторів

5. Перелік графічного матеріалу (із зазначенням обов'язкових креслень)
Презентаційні матеріали (слайди)

6. Консультанти розділів дипломного проекту

Розділ	Прізвище, ініціали та посада консультанта	Підпис, дата	
		завдання видав	завдання прийняв
Антиплагіат	Гурман І.В., доцент		
Нормоконтроль	Яшина О.М., доцент		

7. Дата видачі завдання « 01 » вересня 2022р.

КАЛЕНДАРНИЙ ПЛАН

Назва етапів (розділів) дипломного проекту (роботи)	Строк виконання етапів проекту (роботи)	Примітка
1 Ознайомлення з предметною областю; формулювання мети та задач дослідження; визначення об'єкта та предмета дослідження; визначення структури дипломної роботи	01.092 – 07.09.2022	
2 Робота над розділом 1 дипломної роботи – вивчення літературних джерел, аналіз відомих методів, підходів та засобів за темою роботи; висновки до розділу й постановка задачі	08.09 – 25.09.2022	
3 Робота над розділом 2 дипломної роботи –аналіз проблеми холодного старту; огляд автоматичних масштабувальників; висновки до розділу	26.09 – 10.10.2022	
4 Робота над науковими статтями	11.10 – 20.10.2022	
5 Робота над розділом 3 дипломної роботи координованого масштабувальника; аналіз стадій запуску Kubernetes; висновки до розділу	11.10 – 26.10.2022	
6 Робота над розділом 4 дипломної роботи – оцінювання продуктивності координованого і горизонтального масштабувальника; перевірка поставлених гіпотез; висновки до розділу	27.10 – 15.11.2022	
7 Узгодження постановки задачі, отриманих результатів та висновків; написання вступу, загальних висновків, оформлення джерел посилання та додатків, оформлення пояснювальної записки та графічних матеріалів згідно вимог стандартів	06.11 – 30.11.2022	
8 Попередній захист ДР	Листопад (згідно графіка)	
9 Перевірка ДР на плагіат, нормконтроль, отримання відгуків та рецензій. Брошурування (зшиття) пояснювальної записки)	22.11.2022 – 04.12.2022	
10 Підготовка до захисту та захист ДР	06.12.2022-10.12.2022	

Студент _____ Я.Ю. Маєвський
Підпис Ініціали, прізвище

Керівник проекту (роботи) _____ Н.І. Праворська
Підпис Ініціали, прізвище

РЕФЕРАТ

Тема дипломної роботи: Метод підвищення ефективності автоматизації масштабування мікросервісів у відкритій системі автоматичного розгортання і управління контейнеризованими застосунками.

Автор роботи: Маєвський Ярослав Юрійович.

Керівник проекту: Праворська Наталія Іванівна.

Пояснювальна записка: 116 с., 24 рис., 5 табл., 3 дод., 59 джерел.

АВТОМАТИЧНЕ МАСШТАБУВАННЯ, КОНТЕЙНЕР, МІКРОСЕРВІСИ, KUBERNETES, ПРОБЛЕМА ХОЛОДНОГО ЗАПУСКУ, ГОРИЗОНТАЛЬНЕ МАСШТАБУВАННЯ, CNI, KEDA, KPA, DOCKER, CRIO-O, CONTAINERD, CRI.

Метою роботи є підвищення ефективності автоматизації масштабування контейнеризованих застосунків в Kubernetes, вирішення проблеми холодного запуску контейнерів при масштабуванні.

У дипломній роботі детально проаналізовано фактори, які впливають на автоматичне масштабування контейнерів в системі Kubernetes. Досліджено вплив проблеми холодного старту на швидкодію масштабувальників. Проведена оцінка існуючих алгоритмів автоматичного масштабування, в частності горизонтального, на базі якого було реалізовано новий алгоритм, який вирішує проблему холодного старту. Також був проаналізований вплив різних мережевих інтерфейсів на середовище контейнерів, розмір кластера, мови програмування і типи контейнера на час масштабування контейнеризованого програмного забезпечення.

Для реалізації тестового середовища було використано платформу Kubernetes, Docker як засіб контейнеризації застосунків, мережевий інтерфейс CNI, а також EFK стак з Prometheus для отримання і аналізу метрик середовища.

Практична значимість отриманих результатів полягає у розробці метода, який вирішує проблему холодного запуску контейнерів а також покращує швидкодію автоматичного масштабування в Kubernetes.

2 грудня 2022 р.
Дата

Підпис

ABSTRACT

Master's thesis: « A method of increasing the efficiency of automating the scaling of microservices in an open system of automatic deployment and management of containerized applications »

Author: Yaroslav Yurievich Maievskij.

Head of work: Natalia Ivanivna Pravorska.

Master's thesis consist of: 116 p., 24 pic, 5 tab., 3 add., 59 srs.

AUTOSCALING, CONTAINER, MICROSERVICES, KUBERNETES, COLD START PROBLEM, HORIZONTAL SCALING, CNI, KEDA, KPA, DOCKER, CRIO-O, CONTAINERD, CRI.

The aim of the work is to increase the efficiency of automating the scaling of containerized applications in Kubernetes, to solve the problem of cold startup of containers during auto-scaling.

In the thesis, the factors that affect the automatic scaling of containers in the Kubernetes system are analyzed in detail. The impact of the cold start problem on the speed of scalers was studied. An evaluation of the existing automatic scaling algorithms was carried out, in particular the horizontal one, on the basis of which a new algorithm was implemented that solves the problem of a cold start. The impact of different network interfaces on the container environment, cluster size, programming languages, and container types on the scaling time of containerized software was also analyzed.

To implement the test environment, we used the Kubernetes platform, Docker as a tool for containerizing applications, the CNI network interface, as well as the EFK stack with Prometheus for obtaining and analyzing environment metrics

The practical significance of the obtained results lies in the development of a method that solves the problem of cold startup of containers and also improves the speed of automatic scaling in Kubernetes.

2 грудня 2022 р.
Дата

ЗМІСТ

ПЕРЕЛІК СКОРОЧЕНЬ

БД	–	база даних
HPA	–	horizontal pode autoscaler
VPA	–	vertical pode autoscaler
KEDA	–	kubernetes event driven autoscallor
CHPA	–	custom horizontal pod autoscaler
CRI	–	kubernetes container runtime interface
CNI	–	container network interface
QPS	–	queries per second

ВСТУП

Починаючи з 2010-х років хмарно-орієнтовані технології активно розвиваються, а сам стиль розробки і розгортання програмного забезпечення значно змінився. В той же час все більше населення Землі отримують доступ до інтернету, таким чином динамічно збільшуючи клієнтську базу комерційних компаній. Все більше людей користуються он-лайн сервісами для різних потреб, від звичайного перегляду медіа контенту, бронюванню місць в лікарні, чи оплати покупок за допомогою фінансових веб-додатків.

З великим ростом навантаження на веб-сервіси, комерція прагне спростити впровадження і підтримку існуючого програмного забезпечення, а також зменшити фінансові витрати на підтримку серверів, тощо.

Для рішення цієї задачі Google представили в 2014 році Kubernetes – платформу для управління контейнеризованим програмним забезпеченням, як результат – вони розпочали еру віртуалізованого розгортання програмного забезпечення в контейнерах – повноцінних віртуальних машинах. Віртуалізація дозволила запускати безліч віртуальних контейнерів на одному фізичному сервері, що дало можливість краще використовувати фізичні ресурси сервера та можливості масштабованості програмного забезпечення. Як результат, це можливості автоматичного масштабування і контейнеризації відчутно зекономило вартість підтримки, впровадження і супроводу програмного забезпечення.

Але готові реалізовані рішення все ще в процесі активної розробки, і мають слабкі місця, які впливають на швидкодію розгортання і запуск програмного забезпечення. Одним з таких факторів є проблема холодного старту контейнерів під час автоматичного масштабування, що як результат – збільшує час запуску ПЗ, і як кінцевий результат – користувачам веб-застосунків прийдеться очікувати звільнення ресурсів сервера, або запуск нового контейнера, який буде опрацьовувати ці запити. Така ситуація створює проблему швидкодії програмного забезпечення і в цілому – негативного користувацького досвіду з цим ПЗ.

Таким чином, було вирішено дослідити предметну область і сформувати метод, який буде пом’якшувати проблему холодного старту і покращувати швидкодію запуску контейнеризованого програмного забезпечення.

Останнім часом архітектура мікросервісів набуває все більшої популярності як бажаний спосіб вирішення проблеми постійно зростаючої монолітності програмного забезпечення. Ідея мікросервісів полягає в декомпозиції складних програмних систем на сукупність невеликих сервісів обмеженої бізнес-сфер, кожен з яких взаємодіє один з одним за допомогою легкого комунікаційного механізму. Таким чином, розробники програмного забезпечення об'єднуються в невеликі команди, що обертається навколо одного єдиного мікросервісу для безперервної інтеграції та швидше, ніж будь-коли. Незалежне розгортання мікросервісів приносить велику стійкість і робить автоматичне масштабування ключових компонентів простим і швидким, таким чином гарантуючи якість обслуговування .

Концепція мікросервісів сприяє створенню безлічі інструментів і засобів для швидкого створення мікросервісних додатків, таких як Spring Cloud для розробки на Java [10], Istio для сервісних сіток [9] або Kubernetes для оркестрування контейнерів [8]. З появою Cloud Native появою більшість реалізаторів мікросервісів розгортають свої додатки в хмарних середовищах і застосовують контейнерний підхід. Контейнери надають користувачам більш передбачувані, ізольовані, але легкі середовища виконання, ніж традиційні віртуальні машини.

Тим не менш, самі по собі контейнери не мають уявлення про те, як замінити або масштабувати себе у відповідь на критичну помилку в роботі програмного забезпечення. Саме у цьому випадку Kubernetes як платформа для оркестрування контейнерів, що забезпечує порятунок рішення проблем з контейнерами. Kubernetes широко популярний в великих сервісах і користується довірою у великих технологічних гігантів. Серед її передових можливостей - автоматизоване масштабування, самовідновлення пошкоджених контейнерів, виявлення сервісів в мережі та оркестрування сховищ даних.

Програмне забезпечення побудоване за мікросервісною методологією добре поєднуються з Kubernetes. Kubernetes повністю керується на стороні власників

мікросервісів, що надає динамічне автоматичне масштабування сервісів, що є однією з привабливих особливостей для багатьох компаній. Вбудованим автомасштабувальником в Kubernetes є Horizontal Pod Autoscaler (HPA). Однак, відповідно до цілей рівня обслуговування (SLO), різні бізнес-моделі мають різні вимоги до продуктивності автомасштабування сервісу. Час відгуку є одним з критичних параметром SLO, що представляє собою час від запиту послуги кінцевими користувачами до її успішного завершення.

Коли мікросервіси масштабуються, частина користувачів системи може відчути тривалу затримку в роботі сервісу або недоступність послуги, що може негативно вплинути на досвід користувачів. Затримка холодного запуску визначається багатьма факторами на різних етапах. У середовищі Kubernetes процес масштабування складається з декількох взаємопов'язаних фаз. Він починається з автомасштабування, яке виявляє піковий трафік, після чого слідує планування контейнера. Поки не завершиться ініціалізація контейнера та виконання коду, ми можемо констатувати успішну відповідь.

Щоб досягти низького часу опрацювання запитів для сплеску трафіка, ми повинні розуміти процес холодного запуску та визначити ключові фактори, що впливають на нього. Наше дослідження вивчає проблему холодного старту та аналізує його робочий процес. Тільки з кращим знанням механізму ми можемо запропонувати стратегії пом'якшення наслідків для боротьби з побічним ефектом, і в той же час отримати гнучкий автоматичний масштабувальник. Таким чином, ми ставимо наступні дослідницькі питання, кожне з яких кожне з яких спирається на результати попереднього

Проблема полягає в тому, що при надходженні сплеску трафіку може не вистачати достатньої кількості серверних екземплярів для опрацювання трафіку, в той час як автоматичне масштабування іноді починається занадто повільно. Затримка приведення екземплярів контейнера до потрібної кількості екземплярів контейнера до бажаної кількості викликає занепокоєння у підприємств, орієнтованих на клієнта, і чутливих до затримок підприємств. Ця проблема добре

відома як проблема холодного старту при автоматичному масштабуванні контейнерів.

Актуальність теми. Пандемія COVID-19 сильно вплинула на життя людей в цілому. Багато існуючих комерцій не були готові до такого різкого росту он-лайн користувачів, так як звичайні люди все більше робили он-лайн замовлень, оплат і покупок. Різке збільшення трафіку на веб-застосунки критично вплинуло на сервери. Он-лайн сервіси від великого навантаження все довше опрацьовували запити користувачів, а іноді взагалі не витримували навантаження і переставали працювати. Як результат – веб-сервіс не працює до перезапуску, а також потенційні втрати користувачів і прибутку від них.

Цю проблему вирішує автоматичний масштабувальник навантаження, який в залежності від трафіку користувачів змінює кількість контейнерів, які будуть обслуговувати користувацький трафік. Але як показують дослідження, такі автоматичні масштабувальники працюють не ефективно, і як результат – призводять до надлишкових витрат на підтримку серверів. Причиною цього є проблема холодного старту контейнеризованих застосунків і не коректний вибір технологій і конфігурацій запуску середовища Kubernetes.

Дане дослідження дає змогу пом’якшити проблему холодного старту за рахунок реалізованого метода покращеного горизонтального масштабувальника, а також покращити швидкодію автоматичного масштабування контейнерів в цілому. Як результат, такий метод дозволить зекономити на утриманні веб-застосунків за рахунок економії на фізичних ресурсах сервера.

Нашою метою є виявлення факторів, що сприяють тривалому холодному старту. Базуючись на дослідженнях роботи, ми пропонуємо стратегії пом'якшення наслідків для користувачів контейнеризованих мікросервісів.

Усунення або пом'якшення негативного впливу холодного старту дозволяє мікросервісним системам масштабуватися, забезпечуючи таким чином досягнення бажаної якості послуг.

Об’єкт дослідження. Відкрита система автоматичного розгортання та масштабування та управління застосунками контейнерами Kubernetes.

Предмет дослідження. Методи підвищення ефективності масштабування мікросервісів в Kubernetes.

Метою дослідження є підвищення ефективності автоматизації масштабування контейнеризованого програмного забезпечення в Kubernetes.

Достовірність результатів забезпечується проведеним аналізом ефективності та тестування середовища з використанням сформованого модифікованого горизонтального автоматичного масштабувальника.

Апробація результатів кваліфікаційної роботи робота протестована на віртуальній машині AWS з використанням Kubernetes, з Docker як засіб контейнеризації, мережевий інтерфейс Cillium.

Основними завданнями роботи виступають:

1. Дослідження і аналіз факторів, які негативно впливають на швидкодію автоматичного масштабування контейнеризованих застосунків, а також створюють умови для існування проблеми холодного запуску;
2. Аналіз екосистеми Kubernetes і етапів підготовки нових контейнерів, пошук слабих місць, які можуть впливати на швидкодію;
3. Детальний аналіз способів віртуалізації програмного забезпечення і їх порівняння, а саме: Docker, Containerd, CRI-O, Kata Container, а також їх вплив на швидкодію програмного забезпечення;
4. Аналіз і порівняння існуючих алгоритмів автоматичного масштабування, а саме: горизонтальний автомасштабувальник, вертикальний автомасштабувальник, Keda і Knative.
5. Розробка модифікованого горизонтального автоматичного масштабувальника, який вирішує проблему холодного старту і покращує швидкодію розгортування ПЗ.
6. Дослідження способів для зменшення впливу знайдених факторів на масштабування контейнерів. Аналіз впливу мережевого інтерфейсу, розміру кластера, мови програмування, типу контейнера на швидкодію розгортання контейнера.

7. Створення середовища для тестування і реалізація сформованого рішення. Формування метрик для подальшого дослідження.

8. Оцінювання сформованого рішення з точки зору ефективності, порівнюючи його з готовими рішення, які використовуються з коробки, та надаючи результати стосовно продуктивності цього метода.

Наукова новизна:

1. Спроектовано та розроблено універсальний модифікований горизонтальний алгоритм автоматичного масштабування, який вирішує проблему холодного старту масштабування контейнерів.

2. Удосконалення методів організації середовища для запуску контейнерів, які покращують швидкодію розгортання, в порівнянні з рішенням, яке надається Kubernetes за замовчуванням.

1 ДОСЛІДЖЕННЯ ПРЕДМЕТНОЇ ОБЛАСТІ ТА ПОСТАНОВКА ЗАДАЧІ

У цьому розділі описуються поняття, що входять до сфери представленого дослідження. У розділі 1.1 виконано аналіз мікросервісної архітектури розробки програмного забезпечення, його характеристики та екосистеми. У Розділі 1.2 детально розглянуті технології контейнерів, які складають виокремлене середовище для виконання коду для контейнеризованих мікросервісних додатків. У розділі 1.3 описано платформу Kubernetes, яка забезпечує масштабованість для контейнерних робочих навантажень.

1.1 Мікросервісна архітектура

В останні роки мікросервіс був модним словом серед архітекторів програмного забезпечення [30]. Оскільки програмне забезпечення стає все більшим і складнішим, для кожного архітектора стає головним болем створення ефективного програмного забезпечення, що балансує між якістю та швидкістю.

Галузь стала свідком еволюції архітектури програмного забезпечення від N-рівневого моноліту через сервіс-орієнтовану архітектуру (SOA) до сучасних мікросервісів.

В епоху моноліту одне розгортання може охоплювати декілька модулів у різних бізнес-доменах. Всі модулі упаковані разом в один великий двійковий файл [45]. Незважаючи на те, що монолітний стиль простий у використанні, він має свої недоліки в обслуговуванні та масштабуванні. Крихітне виправлення помилки може вимагати перебудови та перерозподілу всього додатку [30].

SOA з'являється і вирішує багато проблем. Модулі багаторазового використання відокремлюються від моноліту і експортуються як сервіс для решти. У SOA існує всеохоплююча шина повідомлень, що полегшує зв'язок між сервісами

[21]. Підхід SOA успадковується і розширюється мікросервісами, як ми пояснимо в цьому розділі. На рис. 1 представлено три стилі архітектур, що відрізняються за рівнями декомпозиції.

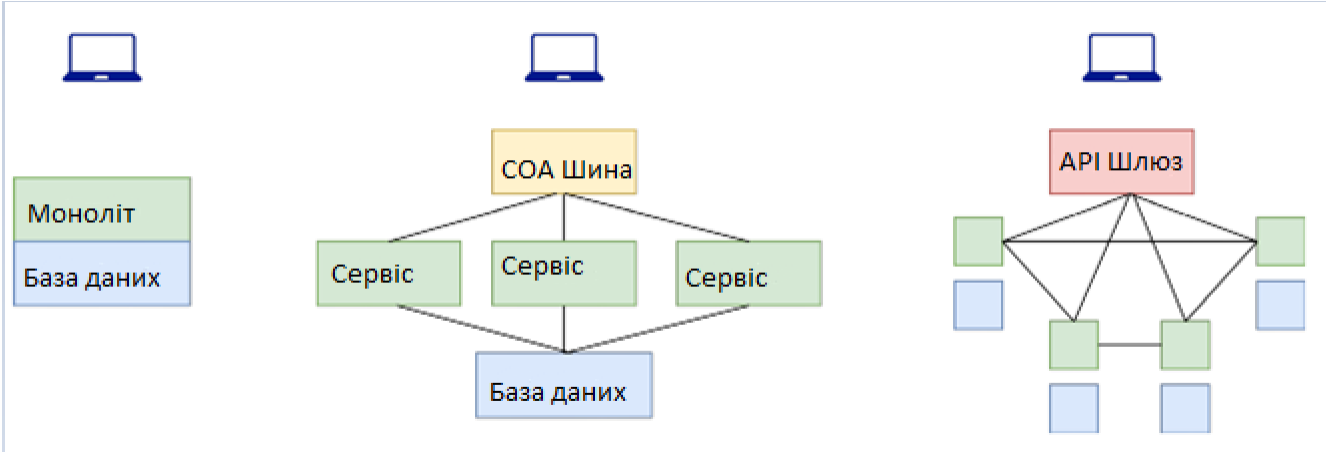


Рисунок 1 – Три архітектурні стилі побудови програмного забезпечення. [39].

Мікросервіси – це не розмір послуг, а спосіб координації та співпраці. Як правило, мікросервіс працює в чітко визначеній сфері бізнесу і володіє виключно простором даних. Мікросервіси та зовнішні суб'єкти взаємодіють один з одним за принципом "виробник-споживач". Існує декілька принципів проектування, які роблять архітектуру мікросервісів помітною.

Обмежений контекст. Визначення меж сервісу є першою проблемою на шляху проектування мікросервісних додатків. Сем Ньюман у своїй книзі припускає, що мікросервіс повинен складатися з сильно пов'язаних функцій, узгоджених з межею бізнес-можливостей, і бути агностичним до інших мікросервісів [41]. Кожен мікросервіс є незалежною, атомарною одиницею, що забезпечує специфічні для домену операції CRUD.

Однією з поширених практик є доменно-орієнтоване проектування (Domain-Driven Design, DDD), введене Еріком Евансом (Eric Evans) [14]. Відповідно до DDD, різні модулі повинні мати різні обмежені контексти, що визначають діапазон застосовності цього модуля. Встановлення повсюдної мови допомагає знайти

"золоту середину" на межі. Одні й ті ж терміни або предмети можуть мати різне значення та призначення в різних сферах.

Простір даних. Інший принцип проектування полягає в тому, що мікросервіси володіють своїм простором даних. Спільне володіння простором даних відхиляється від мети декомпозиції та незалежного розгортання. Враховуючи, що два мікросервіси випадково є співвласниками однієї і тієї ж таблиці бази даних, модифікація схеми даних одного мікросервісу може призвести до несумісності в інших мікросервісах.

Вирішити цю проблему допомагає набір паттернів моделювання даних, таких як делегування сервісів даних, озера даних [17], розподілені транзакції [5], джерела подій та CQRS (Command Query Responsibility Segregation – розмежування відповідальності за виконання командних запитів) [19]. Озера даних вирішують проблему залежності даних шляхом дублювання та переміщення даних в озера даних, доступні тільки для читання. У той час як джерело даних, що належить мікросервісу, служить єдиним джерелом істини, синхронізовані озера даних забезпечують запит даних.

Модель розподілених транзакцій забезпечує коректне та узгоджене оновлення даних у ланцюжках мікросервісів. Однією з реалізацій розподілених транзакцій є використання дизайну Sgags [49]. У Sgags будь-який мікросервіс повинен реалізовувати дві функції: фіксацію та відкат для підтримки розподіленої транзакції. І останнє, але не менш важливе, ще одним домінуючим патерном даних в мікросервісах є подієвий сорсинг та CQRS. Ідея джерела подій полягає в тому, що замість того, щоб зберігати стани даних, в сховище подій записується журнал обліку операцій з даними. Подієвий сорсинг працює з патерном CQRS, що передбачає запит даних до сховища подій.

Міжслужбовий зв'язок. Мікросервіси взаємодіють один з одним за допомогою легкого, асинхронного механізму зв'язку. Мікросервіс за задумом не знає про існування інших. Тому міжсервісний зв'язок віддає перевагу інтерфейсам на основі контрактів, таким як RESTful API, gRPC або GraphQL. Обмін об'єктами повідомлень і формат повинні бути вичерпно і явно визначені як частина

"контракту". Асинхронна модель, керована подіями, значно зменшує витрати на координацію та затримки. На рисунку 2 представлена реальна архітектура мікросервісу та модель комунікації.

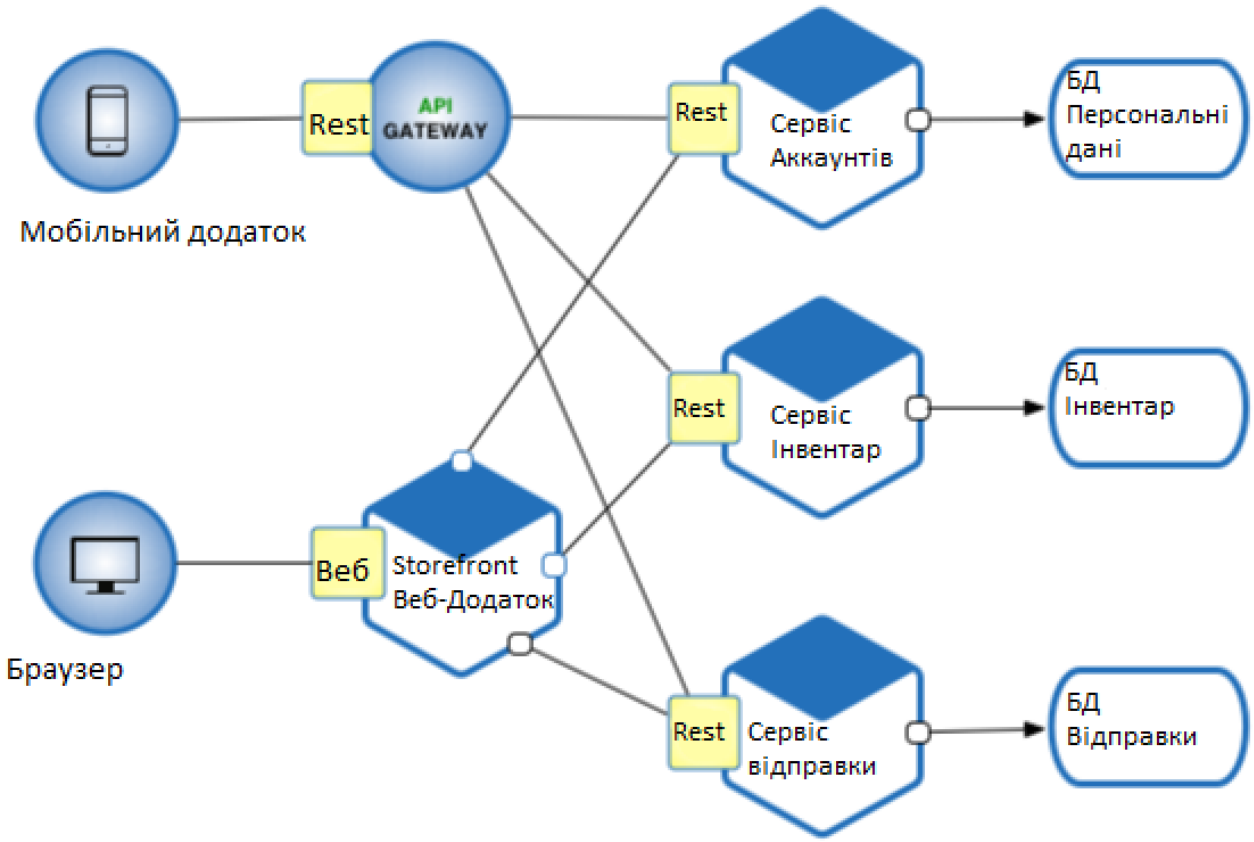


Рисунок 2 – Три архітектурні стилі побудови програмного забезпечення.

1.2 Характеристики мікросервісної архітектури

З основних характеристик мікросервісної архітектури можна виділити чотири ключових властивості [62]. Варто зазначити, що навіть незважаючи на те, що мікросервіси користуються наступними перевагами, все ще існує багато проблем для тих, хто вперше впроваджує їх, як в бізнес-проектуванні, так і у виборі технології, наприклад, проблема "холодного старту", яка обговорюється в цій дисертації. Таким чином, ми розглянемо чотири основні характеристики нижче:

Децентралізоване управління: Команда мікросервісу має права і є поліглотом відносно вибору технології для кожної частинки всієї програми. У такій

міжфункціональній, самоорганізованій команді витрати на координацію зведені до мінімального рівня. Команди можуть обирати бажані технологічні стеки та способи розробки проектів, такі як Scrum або ощадлива розробка програмного забезпечення.

Незалежне розгортання: Оскільки мікросервіси чітко пов'язані з бізнес-доменами, стає можливим незалежне розгортання та управління випусками компонентів мікросервісів. Команди можуть мати різні темпи розробки. Мікросервісний підхід забезпечує паралельний випуск декількох продуктів за умови, що інтерфейси сервісів добре розроблені та задокументовані на ранній стадії.

Гнучкість та автоматизація: Інструменти автоматизації відіграють центральну роль у мікросервісному стилі побудови програмного забезпечення. Реалізатори мікросервісів використовують DevOps, GitOps, Infrastructure as Code (IaC) тощо для прискорення доставки та скорочення часу виходу на ринок. Крім того, впровадження функцій виправлення помилок та вдосконалення не обов'язково вимагатиме перебудови та перерозгортання всієї системи, як це відбувається в монолітній архітектурі.

Тонке масштабування: Мікросервісна система обіцяє відмовостійкість. Ключові компоненти можна масштабувати на вимогу. Оркестрування послуг та автоматичне масштабування досягається за допомогою таких платформ, як Kubernetes.

1.3 Контейнеризація програмного забезпечення

Технологія віртуалізації програмного забезпечення в вигляді контейнерів відмінно поєднується з мікросервісами. Незважаючи на те, що технологія контейнеризації з'явилася ще в ранніх версіях ядра Linux, сучасні контейнери отримали своє ім'я до виходу Docker в 2013 році. У порівнянні з віртуальними машинами, контейнер - це легке мінімальне середовище, що реалізує ізоляцію на рівні процесів. Контейнери розміщуються в операційній системі хоста і не

потребують гіпервізора, як віртуальні машини, що значно зменшує накладні витрати і підвищує портативність.

Основна технологія, яка робить сучасні контейнери можливими та успішними, включає простори імен Linux, групи управління (cgroup) та об'єднані файлові системи [47]. Простори імен Linux розділяють ресурси ядра, так що процеси контейнера добре ізольовані, захищені та агностичні до процесів з інших контейнерів. Простір імен дає ілюзію контейнерним процесам, що вони володіють виключно ресурсами ядра та обчислювальними ресурсами. Наприклад, простір імен PID забезпечує незалежне управління процесом у відриві від кореневого дерева процесів та інших просторів імен PID[42].

Кореневий процес в контейнері буде бачити, що його ідентифікатор процесу дорівнює 1. Інші ресурси ядра, які можуть бути ізольовані за допомогою контейнерів, включають мережевий стек, простір користувача, файлові системи. Cgroup – це ще одна функція ядра, яка накладає політику обмеження ресурсів та облік використання ресурсів ядра [36]. За допомогою cgroup операційна система може контролювати, координувати та узгоджувати використання ресурсів контейнерів. В результаті, вона уникає умов гонки за ресурсами середовища виконання програмного забезпечення.

Файлова система Union дозволяє об'єднувати декілька файлових систем в стек, представляючи єдиний рівень доступу до даних [15]. Набір файлів монтується в одну точку монтування об'єднання. Знизу вгору файли та каталоги будуть об'єднані або перекриті. Файлова система об'єднання має декілька реалізацій, таких як overlay2, aufs та devicemapper. Це технологія, яка підтримує процес створення снапшотів контейнерів. Нижній шар, доступний тільки для читання, і верхній шар, доступний для читання і запису, працюють спільно, щоб забезпечити передбачуване середовище виконання для процесів.

1.4 Інструменти віртуалізації контейнерів

Контейнерне середовище виконання – утиліта, яка надає ізольоване середовище виконання коду, побудоване на основі вищезгаданих технологій. Docker є одним з багатьох і найбільш відомих контейнерних рушіїв, сумісних з OCI. OCI1, скорочено від Open Containers Initiative (Ініціатива відкритих контейнерів), є керівною організацією в контейнерній індустрії, метою якої є стандартизація формату образів контейнерів, набору файлів системи виконання, життєвого циклу контейнерів та API для розповсюдження образів.

Відповідно до специфікації OCI, все більше середовищ виконання контейнерів було винайдено та розроблено з надзвичайною швидкістю. Різниця між різними середовищами виконання контейнерів полягає в їх функціональних можливостях і API. Залежно від того, наскільки багата функціональність, контейнерні середовища виконання можна розділити на дві категорії: високорівневі середовища виконання та низькорівневі середовища виконання [29]. Високорівневі середовища виконання надають універсальні, версійні API для управління всім життєвим циклом контейнера та побудови образу, в той час як низькорівневі середовища виконання, такі як runc, kata контейнери, відповідають лише за створення контейнера. Низькорівневі контейнери повинні бути прозорими для кінцевих користувачів.

Docker Engine2, широко відомий як Docker, є повноцінним високорівневим середовищем виконання контейнеризованого програмного забезпечення. Він використовує архітектуру клієнт-сервер з багатьма модульними та замінними компонентами, такими як бібліотечна мережа та драйвери для зберігання даних. Рисунок 3 ілюструє Docker Engine з точки зору перспективи. Коли кінцеві користувачі вводять команди для створення контейнера, клієнти Docker переводять їх у відповідний виклик API і пересилають демону Docker. Демон Docker діє як сервер, який отримує запит і робить gRPC виклик до базового containerd. Containerd використовує runc для створення контейнерів та управління ними за допомогою процесу shim. Перевагою дизайну на основі C/S є відокремлення обслуговування контейнерів від демона Docker, що дозволяє оновлювати Docker, не торкаючись

існуючих контейнерів [47]. На рисунку 3 представлено Docker Engine зі змінним часом роботи контейнерів OCI.

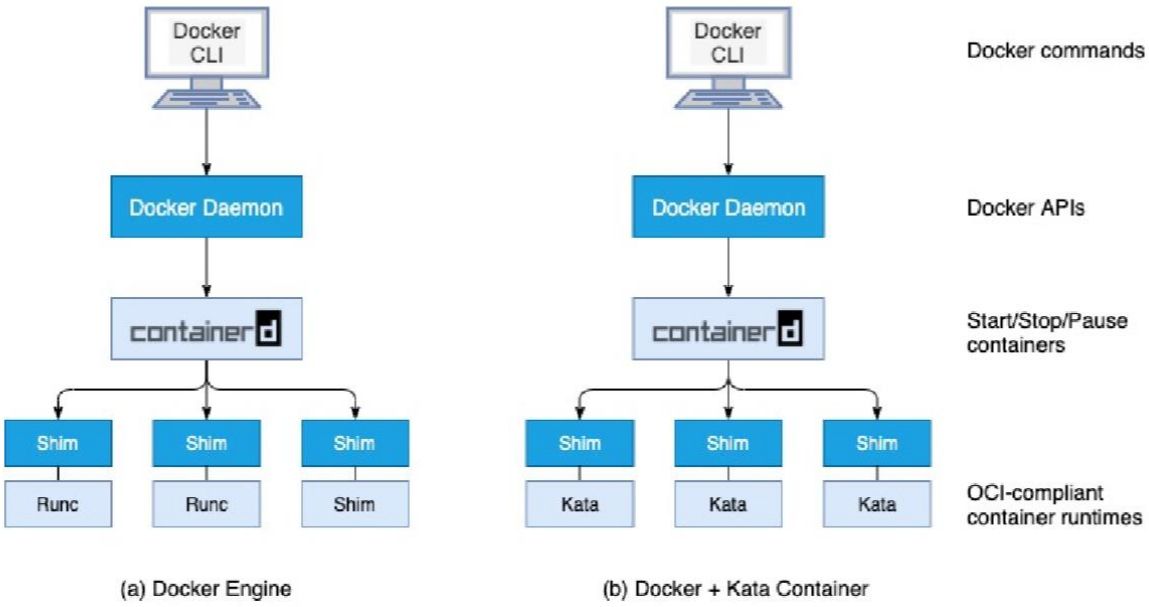


Рисунок 3 – Docker Engine зі змінним часом роботи контейнерів OCI.

Containerd3 є основою Docker, хоча він також може працювати як окреме середовище виконання контейнерів. Containerd є результатом прогресу спільноти Docker у напрямку компонентності та відокремлення раннього монолітного походження. Движок Docker переносить більшу частину управління життєвим циклом контейнерів в containerd і робить останній легким, портативним середовищем виконання контейнерів, яке може використовуватися зовнішніми проектами, такими як Kubernetes.

CRI-O4 – це рішення для виконання контейнерів, що забезпечує безперебійну інтеграцію з Kubernetes, платформою оркестрування контейнерів. Оскільки Docker не призначений для Kubernetes, спільнота Kubernetes шукає заміну Docker; саме тут з'являється CRI-O. CRI-O реалізує як специфікації OCI, так і CRI. CRI розшифровується як Container Runtime Interface, специфікація зв'язку між агентом вузла Kubernetes, kubelet, і базовими середовищами виконання контейнерів. До CRI-O Kubernetes взаємодіє з Docker через окремий dockershim,

який забезпечує відповідність CRI. Containerd також є CRI-сумісним, але він вимагає додаткових cri-плагінів, реалізованих крім containerd; тому він є менш природним, ніж CRI-O.

Контейнер Kata. Всі згадані вище контейнери працюють на runc під капотом. Інше сімейство низькорівневих контейнерів намагається позбутися runc і перебудувати фундамент, щоб додати додаткові можливості, такі як довірене середовище виконання. Контейнери Kata5 є одним з них, що має на меті підвищену безпеку. Філософія kata container – це швидкість контейнерів, безпека віртуальних машин, як стверджує його слоган.

1.5 Kubernetes

Kubernets розроблена компанією Google під назвою Borg для підтримки масштабованих виробничих навантажень. У 2014 році Google розрекламував систему Borg і провів її ребрендинг в Kubernetes. Зі зростанням контейнерної індустрії все більше уваги приділяється Kuberbentes, що робить його фактичним рішенням для управління контейнерами на тлі інших конкурентів, таких як Docker Swarm і Mesos.

Kubernetes вирішує задачі завантаження, оновлення та координацію контейнерних робочих навантажень в середовищі з високим рівнем розподілу для масштабування. Він має багато функцій, які прискорюють і забезпечують безпеку доставки продуктів. Як платформа оркестрування контейнерів, Kubernetes призначає кожному контейнерному робочому навантаженню (Pod мовою Kubernetes) унікальну IP-адресу на рівні кластера та DNS-ім'я, що забезпечує автоматизоване виявлення сервісів та балансування навантаження. Kubernetes проводить періодичні перевірки працездатності та замінює будь-які несанкціоновані робочі навантаження у відповідь на збої. У деяких ситуаціях оператори кластерів Kubernetes можуть вибрати відкат до попереднього стабільного стану.

Кластерна архітектура. Kubernetes розділяє вузли кластера на дві частини: майстер-вузли та робочі вузли. Майстер-вузли представляють площину управління, завданням якої є прийняття глобальних рішень і моніторинг продуктивності системи. Він відстежує поточні стани запущених робочих навантажень і звіряє їх із заздалегідь узгодженими станами. Будь-яка невідповідність запускає процес узгодження, що приводить систему до бажаного стану. Потім робочі вузли розміщують фактичні робочі навантаження, які призначаються площиною управління.

Площина керування. Площина керування складається з набору головних вузлів, на яких розміщуються необхідні компоненти, перераховані нижче. Вони взаємодіють і забезпечують безпеку кластера високої доступності. За винятком наведених нижче елементів, деякі доповнення рекомендуються офіційними документами Kubernetes для виробничих середовищ, наприклад, DNS-сервер для реєстрації служб Kubernetes і Prometheus для моніторингу:

– kube-apiserver: Фасад площини управління Kubernetes, що відкриває RESTful API-інтерфейси Kubernetes. Сервер Kubernetes API за замовчуванням прослуховує CRUD операції над об'єктами Kubernetes на хост-порті 6443. Він обробляє процес аутентифікації та авторизації, а також оновлює стан об'єкта, що зберігається в базі даних кластера, тощо;

– etcd – сховище ключів-значень для запису кластерних об'єктів Kubernetes [23]. etcd слугує єдиним джерелом істини у всьому кластері. Консенсус підтримується протоколом Raft. Нерідко у виробничих середовищах створюється багатовузловий кластер etcd, при цьому кластер etcd і кластер Kubernetes можуть співіснувати на одних і тих же машинах;

– kube-планувальник: Планувальник Kubernetes відповідає за прийняття рішень по плануванню роботи модулів. Kube-планувальник стежить за новоствореними об'єктами pod, консультуючись з ectd через kube-apiserver. У разі створення под-об'єкта, kubescheduler вивчає доступні вузли і визначає найбільш підходящий для хостингу. Прийняття рішення – це двоетапний процес, що включає

в себе фільтрацію вузлів та їх оцінку. В кінці цього процесу kube-планувальник перевизначить поле nodeName для пакета;

– kube-controller-manager: Менеджер контролерів Kubernetes - це сукупність безперервних контурів керування. Кожен цикл контролера відстежує щонайменше один тип ресурсу Kubernetes. Бажаний стан екземпляру ресурсу зберігається у його полі специфікації, а поточний стан - у полі стану. Kube-контролер-менеджер постійно відстежує та коригує ці ресурси, щоб гарантувати, що вони завжди знаходяться у визначеному стані;

– cloud-controller-manager Хмарний контролер-менеджер: містить контури управління, заблоковані постачальником. Деякі ресурси Kubernetes надаються постачальниками хмарних послуг, наприклад, вузли та маршрутизатори. Для узгодження станів цих ресурсів Kubernetes звертається до cloud-controller-manager, а останній делегує завдання хмарним постачальникам.

Робочі вузли. У Kubernetes робочі вузли виконують фактичні робочі навантаження додатків. Компоненти на робочому вузлі включають kubelet, kube-proxy та контейнерні середовища виконання. Хоча більшість компонентів Kubernetes були контейнеризовані, встановлення kubelet все ще відбувається двійковим способом:

– kubelet управляє контейнерами, розгорнутими операторами кластера. Це агент, що працює на кожному вузлі. В його обов'язки входить підтримка життєвого циклу контейнера, перевірка стану контейнера та звітування про його роботу на площину управління;

– kube-proxy: kube-proxy – це мережевий проксі-сервер, що забезпечує виконання правил маршрутизації трафіку на вузлах. Він фільтрує і перенаправляє пакети даних на цільові контейнери;

– середовище запуску контейнера: Kubernetes вимагає використання контейнерів, сумісних з CRI. Середовища виконання контейнерів є замінними компонентами в Kubernetes. Вони виконують фактичні завдання зі створення контейнерів.

Тип ресурсу Kubernetes. Kubernetes визначає набір абстрактних типів ресурсів для управління контейнеризованими застосунками. Об'єкти Kubernetes є екземплярами цих абстрактних типів ресурсів. Вони зберігаються в резервному сховищі etcd, що описує стан робочих навантажень в системі Kubernetes. Більшість об'єктів Kubernetes мають поля spec та status. Перше відноситься до бажаних станів ресурсів, таких як кількість реплік; друге вказує на їх поточні стани. Kubernetes постачається з пакетом вбудованих типів ресурсів.

Деякі ресурси, що мають відношення до проведеного дослідження, наведені нижче. Розробники також можуть привносити власні ресурси та реалізовувати контролер клієнтських ресурсів для розширення функціональності Kubernetes та використання декларативного API-рушія Kubernetes.

Kubernetes запускає контейнерні робочі навантаження в єдиному блоці Pod. Pod - це атомарний робочий набір у світі Kubernetes. Усередині Pod є прихований контейнер паузи, який виконує роль заповнювача, і принаймні один контейнер додатків. У той час як користувачі визначають робочі навантаження в контейнерах додатків, контейнер паузи управляється робочими навантаженнями передбачуваним чином.

Більшу частину часу оператори кластерів Kubernetes взаємодіють з ресурсами розгортання, а не безпосередньо з Pod.Kubernetes, приступаючи до налаштування просторів імен ядра раніше за інших. Контейнери мають бути ефемерними. Його життєвим циклом керують високорівневі ресурси, такі як Deployment або StatefulSet.

Розгортання містить шаблон для Pods. Контролер розгортання спостерігає за об'єктами Deployment API, генерує і управляє Pods, визначеними в полі шаблону. Властивість декларативного управління дає можливість розгортати та відкочувати

StatefulSet схожий на розгортання, але для додатків зі збереженням стану. StatefulSet визначає поведінку Pod в полі .spec.template, як і Deployment. Різниця полягає в тому, що керовані StatefulSet блоки претендують на пов'язані з ними неподільні томи зберігання і мають порядкові індекси для впорядкованого масштабування.

Kubernetes використовує службу Service для розкриття набору внутрішніх блоків (Backend Pods). Оскільки Pods є ефемерними, механізм виявлення служби Kubernetes надає набору Pods унікальне DNS-ім'я. Таким чином, видалення і приєднання базових блоків буде агностичним для зовнішнього світу. Kubernetes також може виконувати балансування навантаження між цими блоками через службу.

1.6. Висновки

В цьому розділі було проаналізовано предмету область дослідження. Наведено описи ключових понять, що входять до мікросервісної архітектури розробки програмного забезпечення а також її характеристики. Також було переглянуто середовища запуску і підтримки контейнеризованих застосунків, описано ключові компоненти і поняття екосистеми Kubernetes, які надалі будуть використовуватись в дослідженні.

2. ПРОБЛЕМА ХОЛОДНОГО СТАРТУ

Холодний старт в автомасштабуванні мікросервісів відноситься до затримки запуску, яка витрачається на те, щоб мати достатню кількість внутрішніх серверів, достатньої для обслуговування переважної кількості запитів. Довга затримка викликає занепокоєння для підприємств, які цінують швидкість реагування та якість обслуговування клієнтів.

Що стосується проблеми холодного запуску як такої, то вона має незначні варіації в різних контекстах. Наприклад, в безсерверній парадигмі проблема холодного старту чітко визначається як час, що витрачається на підготовку середовища виконання коду при першому обслуговуванні виклику функції. У порівнянні з мікросервісами, безсерверна архітектура є більш модульним архітектурним стилем, який в основі і має чіткі цілі відносно скорочення витрат з нульовою вартістю ресурсів, коли немає трафіку з сторони користувачів веб-додатку. Як результат, проблема холодного старту в безсерверній архітектурі є менш складною і широко дослідженою.

Незважаючи на те, що об'єкти дослідження – мікросервіси та функції – дещо відрізняються, проблема холодного старту в цих двох парадигмах має спільну характеристику і проявляється в тому, що під час виклику програми її код ще не завантажений в пам'ять. Висновки, зроблені в рамках концепції Serverless, можуть бути застосовані до контексту мікросервісів. У наступному розділі ми розглянемо холодний старт в парадигмі Serverless: як працює холодний старт, фактори, що впливають на нього, та рішення для оптимізації

2.1 Огляд етапів холодного старту

У безсерверній парадигмі виклик функції починається з того, що платформа перевіряє наявність вільного середовища для виконання коду. Якщо вона є, то виклик буде оброблятися цими екземплярами негайно, що є так званим "теплим" стартом. старт. В іншому випадку ми потрапляємо в процедуру холодного старту,

коли жодне середовище виконання не готова до виконання функції; таким чином, нове середовище виконання (наприклад, контейнер) повинна бути ініціалізована в першу чергу, як показано на рисунку 4.

Після створення контейнера постачальники безсерверних технологій повинні встановити функціональні коди. Вихідний код вихідний код може бути як з локального кешу, так і з віддаленого репозиторію. Наступним етапом є приєднання мережі до функції, що працює у віртуальній приватній хмарі (VPC), щоб захищений, ізольований час виконання коду міг взаємодіяти з зовнішнім світом. Загалом, проходження всіх етапів холодного старту може зайняти кілька секунд. холодного старту. Середовище додатків може бути багаторазово використане для обслуговування майбутніх запитів і підтримуватися в робочому стані з розумним таймаутом.

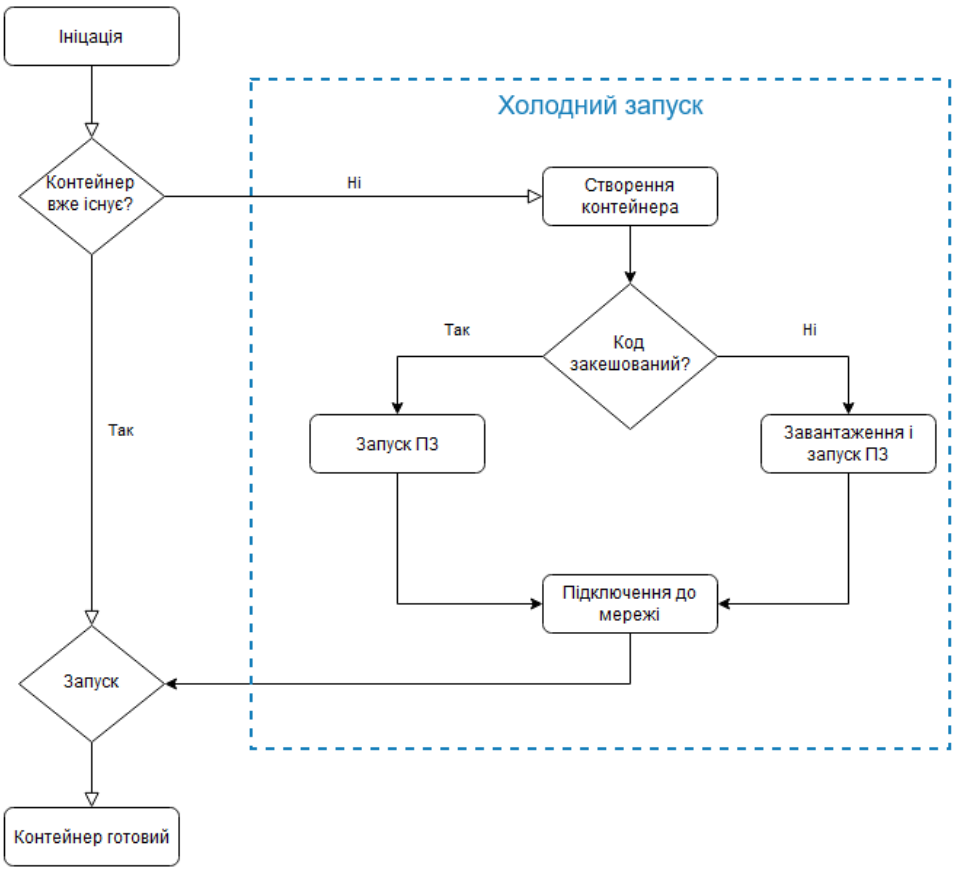


Рисунок 4 – Огляд етапів холодного старту.

Завантаження, контейнеризація та експонування функції є необхідною умовою для того, щоб обслуговувати виклик до середовища виконання. Підводячи підсумок, типовий холодний запуск в безсерверній системі складається з трьох етапів: створення контейнера, завантаження коду функції і розгортання в мережі. Створення контейнера та мережеве розгортання вносять свій внесок у затримку холодного старту на рівні секунд, в той час як час завантаження функціонального коду залежить від способів кешування коду.

2.2 Фактори, які впливають на холодний старт

Холодний запуск у безсерверних системах є скоріше технічною проблемою для постачальників безсерверних систем, ніж для розробників додатків. З одного боку, постачальники безсерверних систем повинні підтримувати мінімальну кількість теплих контейнерів живими, щоб зменшити операційні витрати; з іншого боку, вони повинні знайти спосіб скоротити затримку холодного старту, щоб відповідати вимогам масштабування з нуля.

Для визначення факторів, що впливають на затримку холодного запуску, було проведено широкий спектр досліджень, присвячених оптимізації холодного запуску. Наведено перелік семи факторів впливу, що охоплюють три основні етапи.

Фактор 1. Розгортання коду. Функції повинні бути скомпільовані і завантажені в пам'ять перед виконанням. Залежно від того, де зберігається код і як він побудований, затримка отримання програмного коду може змінюватися. Існує дві проблеми з розміщенням коду. По-перше, зберігання кодів у пам'яті є надто дорогим.

Завантаження кодів з віддаленого реєстру відбувається набагато повільніше, ніж з локального або вторинного сховища. Постачальники безсерверних систем повинні добре продумати кешування коду. По-друге, функція будується з використанням сторонніх бібліотек та кодів, написаних користувачем. Якщо сторонній пакет використовується вперше, його завантаження та інсталяція з мережі на локальний комп'ютер може зайняти кілька секунд.

Oakes et al. проаналізували 20 найпоширеніших пакетів Python за 876 тис. Python-проектів на GitHub і виявили, що витрати на ініціалізацію пакета при першому запуску варіюються від 1 до 13 секунд. На основі отриманих результатів вони пропонують трирівневу систему кешування [43]. Функції, що простоюють, залишатимуться в контейнері паузи до моменту заміни на основі LRU. Для подальшого зменшення споживання пам'яті в пам'яті зберігається пул контейнерів "Zygote" з попередньо імпортованими пакетами для швидкого холодного старту. Пул Zygote використовує політику співвідношення вигод та витрат для вибору контейнера, який потрібно виселити.

Ще однією новою стратегією скорочення часу компіляції та завантаження коду є використання технології CRIU (checkpoint/restore in user namespace – контрольна точка/відновлення в просторі імен користувача) [56]. Замість того, щоб проходити через всі процеси розгортання коду, функція ініціалізується та відновлюється зі знімка, створеного в контрольній точці. Дослідники демонструють дієвий, практичний підхід, який може бути інтегрований з існуючими безсерверними платформами, такими як OpenFaas.

Фактор 2. Функціональне середовище. Середовище функцій – це середовище виконання, що забезпечує ізольоване та захищене виконання функцій. Безсерверний режим ізолює виконання функцій між кількома орендарями в середовищах. Вибір продуктивних середовищ може вплинути на холодний старт. На початку розвитку безсерверної індустрії час виконання забезпечувався за допомогою віртуальних машин. З появою парадигми Cloud Native переважають контейнерні середовища виконання. Перевагу надають ощадливому контейнерному середовищу виконання.

Oakes et al. реалізують спеціалізовану, ощадливу контейнерну систему SOCK, оптимізовану для безсерверних робочих навантажень. Дослідники виявляють, що створення мережевих просторів імен та просторів імен монтування сприяє виникненню вузького місця при холодному старті. Тому SOCK відключає IPv6 і видаляє важкий широкомовний код, щоб прискорити створення мережевого простору імен. Крім того, SOCK використовує для зберігання даних прив'язані

снапшоти замість файлових систем AUFS. З нашої точки зору, ця ощадлива контейнерна система є прийнятною для використання в безсерверних середовищах, хоча вона накладає багато обмежень на функціональність і застосування.

Фактор 3. Вибір мови компіляції програмного забезпечення. Час виконання мови відноситься до проміжного, специфічного для мови середовища, що працює між функціональними кодами та середовищами виконання. Відповідальність включає ретрансляцію тригерних подій, взаємодію функцій користувача з сервісами хмарного постачальника та надання контекстної інформації [4].

Різні хмарні провайдери підтримують різні мовні середовища виконання з різним рівнем оптимізації. Дослідники тестують і показують, що деякі мовні середовища виконання мають більшу затримку ініціалізації, ніж інші, таким чином подовжуючи холодний старт. Джексон і Клінч порівнюють час виконання функцій в NodeJS, Golang, Python, Java, .NET Core 2 на AWS Lambda і NodeJS, .NET C# на Azure Functions. Результат показує, що в сценарії холодного старту Java та .Net Core відстають від інших доступних мовних середовищ виконання [22]. Manner et al. також підтверджують, що затримка холодного запуску для функцій на основі Java займає в 2-3 рази більше часу, ніж для їх аналогів на JavaScript [35]. Варто зазначити, що хмарні вендори постійно докладають зусиль для оптимізації швидкості запуску для мовних середовищ, які вони пропонують[34].

Фактор 4. Налаштування мережі середовища контейнера. Дослідники виявили, що налаштування віртуальної мережі для контейнерів викликає значну затримку холодного запуску. Thomas et al. досліджують оверлейні мережі на основі VXLAN і приходять до висновку, що затримка налаштування мережевого простору імен лінійно зростає зі збільшенням кількості контейнерів у мережі. Автор застосовує тристоронній підхід, щоб зробити налаштування мережі операцією постійного часу для паралельних завдань [57].

Перша конструкція – це консолідація простору імен. Контейнери розгалуження для одного і того ж орендаря поміщаються в один мережевий простір імен. По-друге, ряд IP-адрес буде створено заздалегідь для контейнерів, а не чекати, поки контейнер буде запущений. Нарешті, відображення "один-до-багатьох" між

інтерфейсами VETH та IP-адресами зменшує кількість пристроїв VETH, які необхідно створити.

Мохан та ін. також досліджують питання ініціалізації мережі. Запозичивши ідеї у Kubernetes, вони пропонують рішення для попереднього прогріву мережі, яке досягається шляхом створення контейнерів пауз заздалегідь. Контейнер пауз виконує дві основні функції:

– попереднє створення кінцевих точок мережі (включаючи відображення IP-адреси);

– прив'язка цих кінцевих точок до функціональних середовищ, коли це необхідно.

Щоб максимізувати повторне використання, диспетчер пулу контейнерів пауз буде відновлювати і об'єднувати їх для подальшого використання.

Фактор 5. Розподіл процесора і операційної пам'яті. Розподіл процесора/пам'яті відноситься до ресурсів процесора та пам'яті, зарезервованих та виділених для функціонального середовища. Маннер та ін. досліджують AWS Lambda у 2018 році та спостерігають тенденцію, що накладні витрати на холодний запуск зменшуються, оскільки виділяється більше місця в пам'яті [35]. Дослідники пояснюють це тим, що при надлишку ресурсу завантаження та налаштування контейнерів займає менше часу. Малішев також спостерігав подібні результати у своєму експерименті в 2019 році [34].

Фактор 6. Послідовний виклик функцій. Деякі програми мають послідовний виклик функцій, що означає, що одна функція може викликати іншу, утворюючи конвеєр, який виконує послідовні завдання. Прикладом може слугувати обробка зображень у машинному навчанні, яка може містити кілька кроків до завершення потоку даних. Спільне розміщення цих функцій дозволяє досягти хорошої продуктивності, тоді як неврахування ланцюжка функцій може призвести до значних накладних витрат і затримок. Аккус та ін. повідомляють, що безсерверні платформи, такі як AWS Step Functions та IBM Cloud Functions, однаково ставляться до внутрішнього та зовнішнього виклику функцій [3]. Ці платформи завжди призначають нове середовище для вхідних запитів, незалежно від того,

внутрішні вони чи ні. В результаті більше половини часу обслуговування витрачається даремно.

Розуміння того, як функції поводяться і взаємодіють, відкриває простір для потенційного вдосконалення середовища і відповідно покращення швидкодії. Для підвищення продуктивності послідовних функцій Аккус та ін. пропонують систему SAND, високопродуктивну безсерверну платформу, із застосуванням середовища дворівневої ізоляції відмов та ієрархічної черги повідомлень для прискорення розміщення корельованих функцій [3]. Ідея дворівневої ізоляції відмов полягає в тому, що SAND розміщує функції одного додатку в одному загальному контейнері, призначаючи нові контейнери для функцій різних додатків. У першому випадку використовується розгалуження процесів через менш жорсткі вимоги до ізоляції функцій у середовищі. SAND перевершує Apache OpenWhisk, безсерверний движок з відкритим вихідним кодом.

Дав Ет Аль в своїй роботі розглядає можливості запуску холодного запуску в каскадних ланцюжках функцій з іншої точки зору. Дослідники намагаються визначити найбільш вірогідний шлях (МВШ) виконання функції в DAG. З динамічно налаштованим робочим процесом DGA, функції будуть попередньо розгорнуті непередбачувано [12]. Для того, щоб виявити неявні ланцюжки функцій, Daw та ін. додали унікальний ідентифікатор в заголовки запитів, щоб зафіксувати, звідки походить виклик і для чого він призначений.

Фактор 7. Шаблон виклику функцій. Спостереження в галузі безсерверних технологій показують, що трафік виклику функцій може впливати на холодний запуск. Дослідники Microsoft Azure проаналізували реальні робочі навантаження функцій з їхньої платформи і дійшли висновку, що для 40% додатків час між прибуттями виклику функцій дуже варіативний, що робить його важко передбачуваним. Крім того, спостережувана закономірність помітно відрізняється для різних додатків [54]. Ця інформація може допомогти постачальникам безсерверних технологій ефективно визначати стратегії надання та утримання ресурсів.

Стратегія поділяється на два типи: прогнозоване надання ресурсів (стратегія прогріву) та адаптивне збереження (стратегія підтримання працездатності). Обидві стратегії мають на меті покращити показник потрапляння запитів в систему. Що стосується першої, то Xu та ін. пропонують стратегію адаптивного розігріву (Adaptive Warm-Up, AWU) шляхом прогнозування майбутнього часу виклику функцій для попереднього резервування ресурсів [61].

Дослідники використовують мережевий алгоритм довготривалої короткочасної пам'яті (LSTM) для виявлення залежностей між даними часових рядів викликів функцій. Шахрад та ін. дотримуються останнього напрямку і вказують на те, що поточна політика фіксованого таймауту, прийнята на безсерверних платформах, погано реагує на непостійні, високо варіабельні шаблони викликів. Тоді пропонується адаптивна політика "keep-alive" [54]. Ця політика визначає два ковзаючі вікна: вікно попереднього прогріву та вікно підтримання працездатності. Одразу після закінчення виконання функції вікно попереднього прогріву встановлює інтервал перед повторним завантаженням функції.

Функція буде знаходитися в пам'яті протягом періоду, визначеного вікном збереження активності. Обидва вікна мають адаптивні розміри, визначені на основі шаблонів виклику функцій.

Спостереження в галузі безсерверних технологій показують, що трафік виклику функцій може впливати на холодний запуск. Дослідники Microsoft Azure проаналізували реальні робочі навантаження функцій з їхньої платформи і дійшли висновку, що для 40% додатків час між прибуттями виклику функцій дуже варіативний, що робить його важко передбачуваним. Крім того, спостережувана закономірність помітно відрізняється для різних додатків [54]. Ця інформація може допомогти постачальникам безсерверних технологій ефективно визначати стратегії надання та утримання ресурсів.

Стратегія поділяється на два типи: прогнозоване надання ресурсів (стратегія прогріву) та адаптивне збереження (стратегія підтримання працездатності). Обидві стратегії мають на меті покращити показник потрапляння запитів в систему. Що

стосується першої, то Xu та ін. пропонують стратегію адаптивного розігріву (Adaptive Warm-Up, AWU) шляхом прогнозування майбутнього часу виклику функцій для попереднього резервування ресурсів [61].

Дослідники використовують мережевий алгоритм довготривалої короткочасної пам'яті (LSTM) для виявлення залежностей між даними часових рядів викликів функцій.

Шахрад та ін. дотримуються останнього напрямку і вказують на те, що поточна політика фіксованого таймауту, прийнята на безсерверних платформах, погано реагує на непостійні, високо варіабельні шаблони викликів. Тоді пропонується адаптивна політика "keep-alive" [54]. Ця політика визначає два підхода для оперуванням запуску: підхід попереднього прогріву та підхід підтримання працездатності. Одразу після закінчення виконання функції вікно підхід попереднього прогріву встановлює інтервал перед повторним завантаженням функції. Функція буде знаходитися в пам'яті протягом періоду, визначеного статусом збереження активності. Обидва підходи мають адаптивні розміри, визначені на основі шаблонів виклику функцій.

2.3 Огляд сучасних автоматичних масштабувальників

У цьому розділі ми обговоримо сучасні пропозиції автоскалерів. Автоскалер є центральним компонентом, який виконує всю магію надання ресурсів від імені додатків. Ми порівняємо чотири високопродуктивних автоскалера зі спільноти Kubernetes. Чотири автомасштабувальники можна розділити на два класи: низькорівневий інструментарій автомасштабування та високорівневий фреймворк автомасштабування.

Низькорівневий автомасштабувальник відповідає лише за автоматичне масштабування мікросервісу по горизонталі або вертикалі. До цього класу відносяться HPA, VPA та KEDA.

Високорівневий фреймворк автомасштабування бере активну участь в оркестровці сервісів. Необхідні допоміжні коди можуть знадобитися для вбудовування в код програми. Одним з типових прикладів є Knative.

2.3.1 Огляд автомасштабувальника HPA

HPA (Horizontal Pod Autoscaler) – це вбудований автомасштабувальник, що постачається з Kubernetes. HPA автоматично збільшує або зменшує кількість реплік робочого навантаження на основі заданих правил, наприклад, цільового рівня використання CPU/MEM. Внутрішньо HPA складається з ресурсу Kubernetes та процесу циклу управління. Ресурс HPA – це об'єкт API, який визначає поведінку автомасштабування та бажаний стан цілі масштабування.

Контролер спостерігає за об'єктами ресурсу, зчитує з них стратегію масштабування та динамічно налаштовує цільові робочі навантаження, як показано на рисунку 5.

HPA періодично отримує метрики через API метрик Kubernetes, що підтримуються додатковими модулями моніторингу, такими як метричний сервер. Інтервал вибірки за замовчуванням становить 15 секунд і може бути змінений за допомогою параметра kube-controller-manager -horizontal-podautoscaler-sync-period -fb02fag. Рішення про масштабування приймається на основі співвідношення поточної метрики до цільової, як показано у рівнянні 1. Поточне значення метрики є середнім значенням для всіх блоків.

Якщо застосовуються мультиплікативні метрики, то найбільший результат розрахунку буде числом, до якого слід масштабувати робоче навантаження.

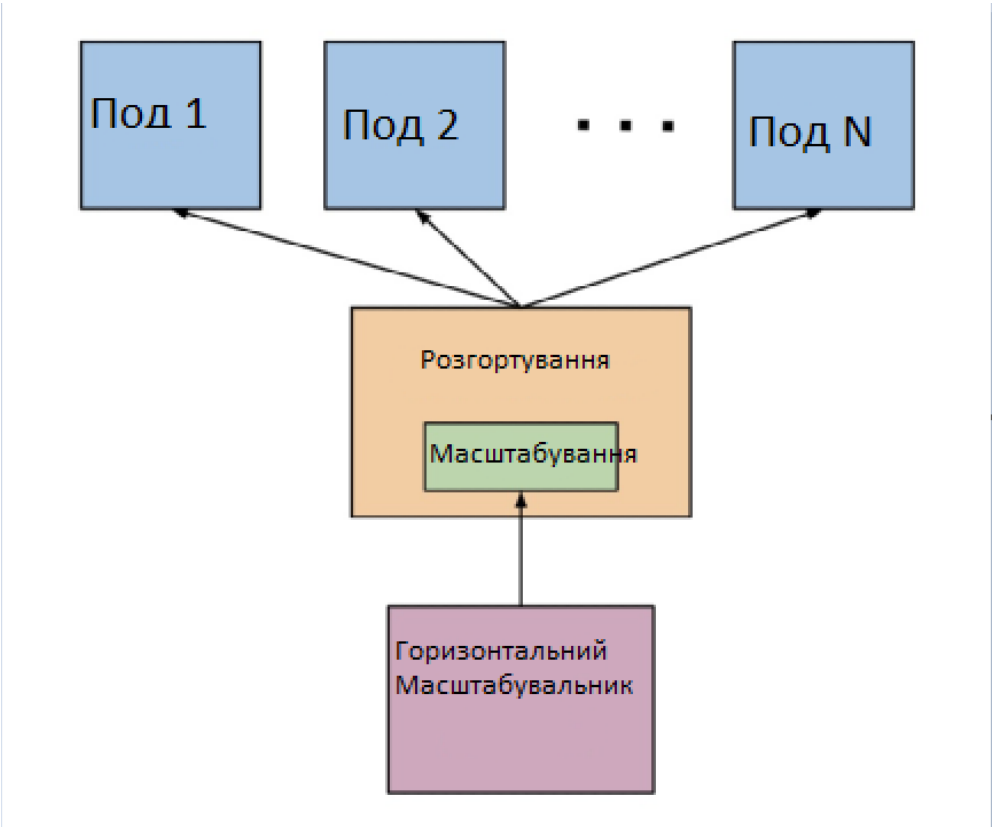


Рисунок 5 – Робота горизонтального масштабувальника

Починаючи з версії 2, HPA додає підтримку для інтеграції з користувацькими метриками, що визначаються користувачем, окрім використання CPU/MEM.

$$targetPodCount = \left\lceil currentPodCount \cdot \frac{currentMetricValue}{targetMetricValue} \right\rceil \qquad 1$$

2.3.2 Огляд автомасштабувальника VPA

На відміну від HPA, VPA (Vertical Pod Autoscaler) масштабує робоче навантаження по вертикалі, що здійснюється шляхом перепризначення квоти ресурсів (наприклад, процесор, пам'ять) для контейнерів в под. VPA менш поширений, ніж HPA; таким чином, він не постачається з Kubernetes за замовчуванням.

Однак, VPA все ще має нішеве застосування. Розробники додатків часто намагаються визначити "золоту середину" для розподілу ресурсів. Надмірно

обережний запит на ресурси може призвести до збоїв, коли трафік нестабільний. VPA здатний вирішити цю дилему. Модуль рекомендацій VPA вивчає історичне споживання CPU/MEM і рекомендує відповідне значення. VPA Updater автоматично замінить запит на ресурси та поля обмежень для блоків.

2.3.3 Огляд автомасштабувальника Keda

KEDA2 (Kubernetes Event-driven Autoscallor) – це проект в рамках Cloud Native Computing Foundation (CNCF). KEDA розширює функціональність HPA, роблячи можливим і легким масштабування на основі інформації з будь-якого джерела подій. KEDA в основному складається з трьох компонентів: скалер, узгоджувач, адаптер метрик:

– масштабувальник: визначає правило відображення між масштабованими робочими навантаженнями (наприклад, Deployment, StatefulSet) та джерелами подій (наприклад, Redis, Kafka). Кожен масштабувальник визначає, як робоче навантаження буде масштабуватися і за яких умов;

– контролер: відстежує джерела подій і виконує власне процес масштабування. При надходженні подій, що відслідковуються, контролер приймає рішення про масштабування та перезаписує цільову кількість реплік робочого навантаження;

– адаптер метрик: перетворює події в числові метрики і виставляє їх в користувацьких кінцевих точках API метрик.

2.3.4 Огляд автомасштабувальника Knative

Knative3 – це високорівневий безсерверний фреймворк, побудований на Kubernetes. Він відповідає за розгортання функцій та автоматичне масштабування, оскільки розробникам потрібно піклуватися лише про фактичну бізнес-логіку. Однією з головних особливостей Knative є можливість автоматичного масштабування від/до нуля.

Холодний старт є давньою проблемою разом з Knative. Функціональність автоматичного масштабування Knative підтримується трійцею Autoscaler, Activator та Queue-Proxy, як показано на рисунку 6. Вони працюють спільно, щоб забезпечити плавне та гнучке автоматичне масштабування.

Автомасштабувальник відіграє аналогічну роль, як і згадані вище автомасштабувальники, але також піклується про проблеми холодного старту. Щоб пом'якшити холодний старт, Knative реалізує дворівневий механізм черги та буферизації. Коли маршрут отримує нові запити, але немає жодного працюючого екземпляра, Activator буде буферизувати вхідні запити і чекати, поки хоча б один екземпляр не буде масштабований. Існує різниця між масштабуванням від 0 до 1 і від 1 до декількох. Як тільки автомасштабувальник помітить, що один екземпляр pod готовий до обслуговування, Activator перенаправить запити на цей екземпляр pod і відключить трафік від його шляху передачі даних. Постійно зростаючий попит буде оброблятися компонентом Queue-Proxy всередині модуля програми.

Контейнер Queue-Proxy працює як додатковий модуль, що знаходиться разом з контейнерами додатків. Knative автоматично вставляє візок Queue-Proxy в контейнери додатків. Шаблон візка відокремлює управління трафіком від бізнес-логіки. Queue-Proxy - це невеликий, легкий буфер. Автомасштабування динамічно регулює розмір пулу екземплярів додатків, щоб уникнути занадто швидкого заповнення черги.

Рисунок 6 – Масштабувальник в Knative

2.4 Висновки

Приведений огляд літератури показав, що за холодний старт в безсерверній системі спільно відповідають декілька факторів. Щоб застосувати уроки безсерверних технологій до наших дослідницьких проблем, ми повинні прояснити розрив між цими двома сценаріями. На відміну від безсерверної архітектури , мікросервісна архітектура має наступну відмінність, на яку варто звернути увагу.

Автоматичне масштабування мікросервісів має різні етапи холодного запуску. Наприклад, проблеми холодного розгортання не існує, оскільки розробники мікросервісів можуть кешувати образи контейнерів на вузлах

локально. У Розділі 2 наочно показано, як відбувається холодний запуск мікросервісів на Kubernetes.

Мікросервіси не обмежуються постачальниками мовних середовищ виконання. Безсерверні технології мають ризик прив'язки до постачальника за рахунок переваг відсутності управління сервером [60]. Однак, користувачі мікросервісів можуть вибирати та адаптувати свої технологічні стеки, створювати інфраструктуру та налаштовувати продуктивність, щоб найкращим чином відповідати їхнім потребам.

Холодний старт у безсерверних системах є більшою проблемою для постачальників безсерверних систем. Небажана затримка холодного старту може відштовхнути клієнтів. У мікросервісах команда розробників та команда інфраструктури зазвичай повинні розглядати цю проблему рука об руку в рамках культури DevOps.

3- РОЗРОБКА РІШЕНЬ ТА МЕТОДІВ ДЛЯ ВИРІШЕННЯ ПРОБЛЕМИ ХОЛОДНОГО СТАРТУ

У цьому розділі представлено пояснення етапів холодного запуску при автоматичному масштабуванні мікросервісів на Kubernetes та запропоновано гіпотетичні фактори, які можуть впливати на затримку запуску. У розділі 3.1 надано приклад мікросервісу, який був використаний в проведеному дослідженні, і пояснюється його налаштування на Kubernetes. У розділі 3.2 виконано розбиття процесу автоматичного масштабування на чотири етапи. Процес масштабування починається з виявлення зростаючого трафіку і завершується новоствореними контейнерами, готовими до обслуговування. У Розділі 3.3 також представлено п'ять гіпотетичних факторів, що базуються, як на дослідженні етапів холодного старту, так і на аналізі літературних джерел. У Розділі 3.4 запропоновано та впроваджено скоординований автоматичний масштабувальник, як доповнення до існуючих автоскалерів.

3.1 Тематичний проект

Прикладом мікросервісного додатку, який було використано для оцінки автомасштабування, є проект Acme Air1 – система бронювання авіаквитків на основі мікросервісів для бенчмаркінгу мікросервісної архітектури. Додаток спочатку був розроблений дослідником IBM Ендрю Спайкером (Andrew Spyker) з використанням WebSphere для сервера додатків Java, MicroProfile для середовища розробки мікросервісів і WebSphere eXtreme Scale для зберігання даних. Він широко цитується для тестування продуктивності програмного забезпечення для порівняння монолітної та мікросервісної архітектури [2, 63, 58]. У 2019 році інший дослідник IBM, Джо МакКлюр (Joe McClure), доопрацював проект, використовуючи фреймворк Spring Boot та MongoDB в якості резервного сховища за замовчуванням 2.

В проведеному дослідженні прийнято рішення спиратися на роботу Джо протягом усього нашого дослідження. Для того, щоб вписати кейс-проект у дослідження, були вбудовані бібліотеки метрик Prometheus в код додатку, щоб показати метрики запитів в секунду (QPS). Крім того, також проведено видалення Open Liberty (наступник WebSphere з відкритим вихідним кодом) з базового образу і використано вбудований сервер Tomcat, який постачається з Spring Boot за замовчуванням, щоб полегшити проведення подальших експериментів цього дослідження. Ревізію для ознайомлення надається в репозиторії Github3.

3.1.2 Представлення проекту Acme Air

Система Acme Air складається з чотирьох мікросервісів, як показано на рисунку 7. Основний сервіс – це інтерфейс, який містить статичні веб-сторінки та здійснює навігацію клієнтів до інших веб-сервісів. Служба польотів надає запити щодо інформації про виліт, прибуття рейсу та нарахування миль винагороди. Служба обслуговування клієнтів повертає інформацію про подорожі клієнтів, включаючи загальну кількість миль - винагород та зарезервовані рейси. Клієнти можуть забронювати авіаквитки, звернувшись до Служби бронювання. Останні три служби володіють виключно своїм інформаційним простором. Зв'язок із зовнішнім світом здійснюється через шлюз API, а внутрішньо сервіси можуть безпосередньо взаємодіяти один з одним.

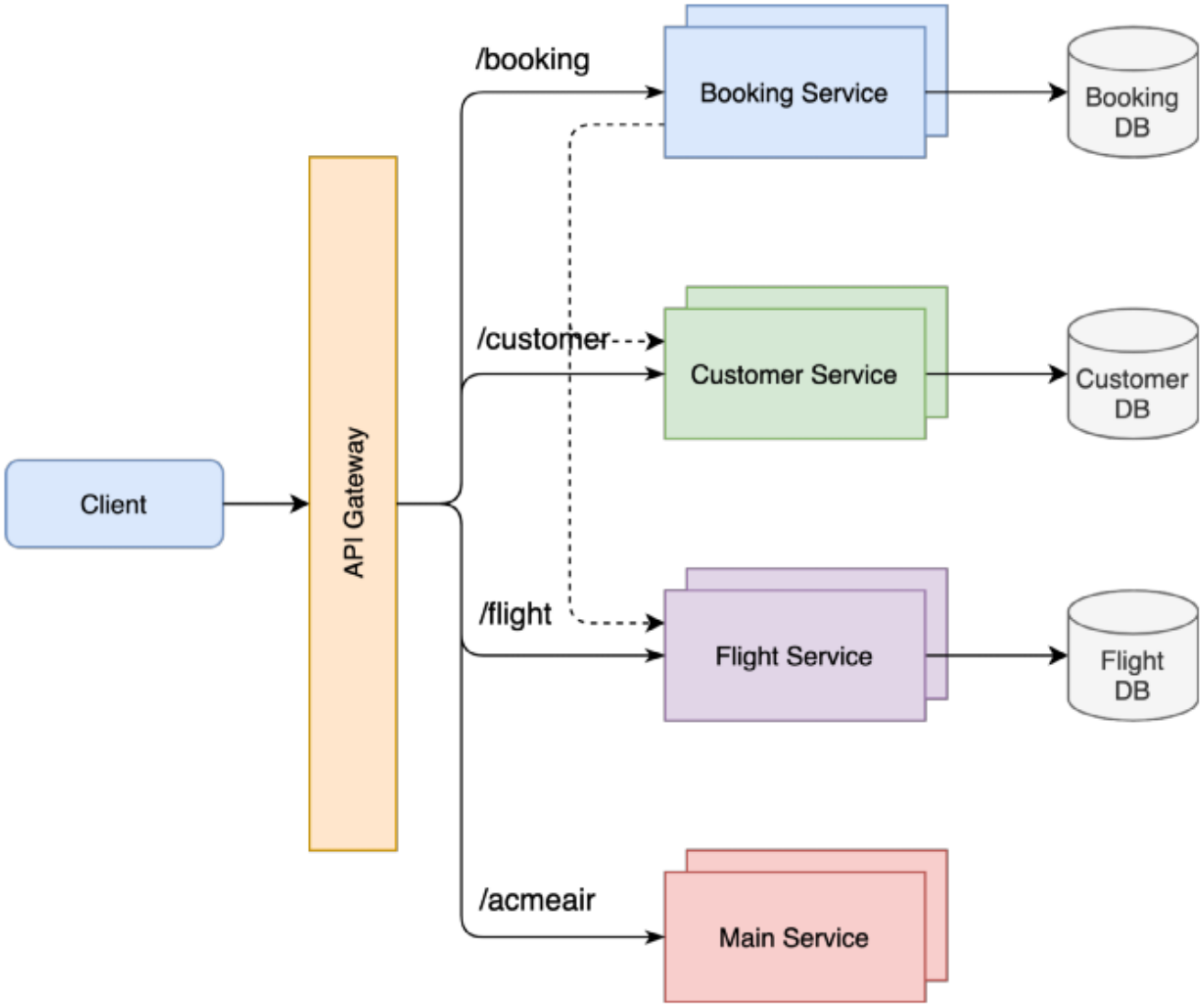


Рисунок 7 – Архітектура тестувального програмного забезпечення Acme Air

Було впроваджено бібліотеку Prometheus Java в систему Acme Air, щоб служби могли виставляти метрики QPS на кінцевій точці /prometheus в текстовому форматі Prometheus, як показано на рисунку 8. Ці показники будуть оцінюватися автомасштабувальниками при прийнятті рішень про масштабування. Наприклад, HPA масштабує модулі на QPS, як тільки бачить, що існуючі запущені екземпляри перевантажені.

```
Method: GET
Endpoint: /prometheus
Content Type: text/plain
Response Code: 200
Response Body:
# HELP http_server_requests_seconds
# TYPE http_server_requests_seconds summary
http_server_requests_seconds_count{uri="/health"} 372.0
http_server_requests_seconds_count{uri="/queryflights"} 12.0
http_server_requests_seconds_count{uri="/prometheus"} 129.0
```

Рисунок 8 – API сервіса Prometheus

3.1.2. Налаштування Kubernetes

У цьому розділі надається опис загального налаштування працюючого кластера Kubernetes для тестування. Деталі повністю представлені у розділі 5. Спочатку виконується контейнеризація додатку Acme Air на кластері Kubernetes. У наданій версії проекту Acme Air вибирається openjdk:11.0.10-jre-slim, як базовий образ для створення образів контейнерів. Кластер Kubernetes встановлено на AWS EC2 за допомогою kops4, одного з офіційно рекомендованих інструментів встановлення. Інсталяційні скрипти та YAML-файли можна знайти в репозиторії Github5.

Розгортання та обслуговування. У розділі 2.3.2 виконується пояснення, що Kubernetes використовує об'єкти ресурсів розгортання і обслуговування для абстрагування фактичних контейнерних робочих навантажень. Розгортання показує, як Kubernetes розгортає визначає, як Kubernetes повинен завантажувати контейнери, а сервіс розкриває веб-сервіс назовні.

Визначення розгортання зазвичай має п'ять обов'язкових полів: *apiVersion, вид, метадані, специфікація, статус*. Kubernetes використовує перші три поля для ідентифікації об'єктів розгортання. Поле spec надає написаний користувачем опис бажаної поведінки та атрибутів, які повинно мати робоче навантаження, в той час, як поле status повідомляє про поточний стан робочого навантаження. Розрив між

цими двома полями вказує на невідповідність стану, і менеджер контролерів Kubernetes працює над зменшенням розриву. Розгортання має контейнери, так що масштабування так само просто, як модифікація поля репліки розгортання вручну або автоматично за допомогою автомасштабування. Лістинг представлений нижче, являє собою представлення об'єкту розгортання та об'єкту Pod для додатку Flight Service.

Об'єкти сервісу виставляють керовані розгортанням модулі (Pods) і балансують трафік навантаження між ними. Kubernetes пов'язує службу з набором модулів через об'єкти кінцевих точок. Кінцева точка синхронізується зі змінами статусу Pod (наприклад, контейнери виходять з ладу). До списку Endpoint потрапляють тільки працюючі боди, які пройшли перевірку на готовність. Крім того, чотири сервіси Acme Air маршрутизуються через API-шлюз, який визначений в іншому об'єкті Kubernetes – Ingress, див. рисунок 9 (а, б).

```
---
# Kubernetes Deployment object definition
apiVersion: apps/v1
kind: Deployment
metadata:
  name: flightservice
spec:
  # desired number of Pod replicas
  # increasing the value causes Deployment to scale out
  replicas: 1
  # template for generating Pods.
  template:
    spec:
      containers:
      - image: flightservice:latest
        resources:
          limits:
            memory: 800Mi
            cpu: 800m
        # kubelet probes pod readiness as code is loading.
        readinessProbe:
          httpGet:
            path: /health
            port: 9080
```

Рисунок 9 (а) – Налаштування Ingress для тестового середовища

```
---
# Kubernetes Pod object definition
apiVersion: v1
kind: Pod
metadata:
  name: flightservice-5b99dffb56-qwvws
spec:
  # this pod is scheduled to node2.
  nodeName: node2
  # identical content as Deployment's podTemplate field.
  containers:
  - .....
status:
  podIP: 172.17.0.5
```

Рисунок 9 (б) – Налаштування Ingress для тестового середовища

Моніторинг. Для збору метрик QPS на рівні подів розгортаються Prometheus, як рішення для моніторингу, щоб замінити метричний сервер, надбудову для моніторингу за замовчуванням, що постачається з Kubernetes, яка надає лише базові метрики CPU/Mem. Prometheus вважається де-факто рішенням для моніторингу в епоху Cloud Native. Формат даних Prometheus був широко прийнятий спільнотою Kubernetes [11]. Prometheus – це система моніторингу на основі витягування, де Prometheus проактивно виявляє та вилучає кінцеві точки метрик додатків та записує їх у свою базу даних часових рядів.

Для того, щоб додаток Flight Service міг масштабуватися на QPS, ми інтегрували бібліотеку метрик Prometheus у вихідний код. Це реалізовано шляхом додавання Java-клієнтської залежності Prometheus та включення підтримки Prometheus в конфігурації управління кінцевими точками. Для автоматизації інсталяції компонентів Prometheus використовується Prometheus operator. У вихідному коді реєструється метрика типу Counter з назвою http server requests seconds count, кумулятивну кількість запитів. Таким чином, QPS рівня pod можна обчислити у виразі PromQL (2) як

$$sum\ by(pod)\big(irate(http\ server\ request\{uri = /queryflights\}[3m])\big) \quad 2.$$

а саме секундну частоту HTTP-запитів для двох останніх вибірок часових рядів з періодом до 3 хвилин.

Крім того, на стороні Kubernetes, автоскалери, як і HPA, отримують користувацькі метрики через єдиний рівень API метрик Kubernetes. Це пов'язано з тим, що Kubernetes за своєю конструкцією здатний працювати з різними рішеннями для моніторингу. Таким чином, виконується розгортання адаптеру Prometheus для конвертації результатів запитів Prometheus в API метрик Kubernetes. Якщо автоскалер має викликати користувацький API метрик Kubernetes, адаптер Prometheus буде робити запити до сервера Prometheus від імені автоскалера та повертатиме агреговані результати.

Автоматичний масштабувальник. У Розділі 3.2 представлене узагальнення сучасних автомасштабувальників. Серед них HPA є автоскалером за замовчуванням, що постачається з Kubernetes, а KEDA розширює функціональність HPA і працює поверх нього. У запропонованому дослідженні було зосередження на HPA через його широке розповсюдження і розгортання його для виконання завдань масштабування додатку Flight Service для бенчмаркінгу в Розділі 4. Виконана конфігурація поведінки масштабування через об'єкт ресурсу HPA, за яким спостерігає контролер HPA, фактичний процес управління циклом виконує завдання масштабування. Тип ресурсу HPA визначає декілька полів для конфігурації стратегій масштабування та умов запуску 8. Нижче детально на рисунку 10 розглянуті поля, які були використані, оскільки HPA є чудовим довідником для проектування нових автомасштабувальників.

```yaml
# Kubernetes HPA object definition
apiVersion: autoscaling/v2beta2
kind: HorizontalPodAutoscaler
metadata:
  name: flightservice-hpa
spec:
  # specifies the target workload to scale
  scaleTargetRef:
    apiVersion: apps/v1
    kind: Deployment
    name: flightservice
  minReplicas: 1
  maxReplicas: 15
  # points which metrics to use
  # to calculate the desired replica number
  metrics:
  - type: Pods
    pods:
      metric:
        name: query_per_second
      target:
        type: AverageValue
        averageValue: 200
```

Рисунок 10 – Конфігурація горизонтального масштабувальника

3.3 Аналіз стадій холодного старту

У Розділі 2.1 наведено вивчення етапів холодного запуску в парадигмі Serverless і підкреслено різницю між двома програмними архітектурами. Знання з Serverless можуть бути застосовані до запропонованого дослідження. З цього розділу відбулося глибоке занурення в робочий процес автоматичного масштабування мікросервісів на Kubernetes.

Під час ретельного аналізу процесу автоматичного масштабування, відбулось розбиття його на чотири етапи, засновані на життєвому циклі контейнера: (1) прийняття рішення про автоматичне масштабування, (2) прив'язка контейнера до вузла, (3) ініціалізація контейнера і (4) запуск додатку. Кожен етап тісно пов'язаний з конкретними компонентами Kubernetes для виконання одного завдання та створення подій для запуску наступного етапу. Щоб допомогти розкрити секрет процедури автомасштабування, проаналізовано відповідний вихідний код, документи, офіційні блоги Kubernetes та HPA, а потім візуалізовано робочий процес та критичні шляхи кожного етапу. Перевагою цієї роботи є те, що є

можливість виміряти затримку запуску кожного етапу окремо та виявити потенційні вузькі місця.

3.3.1 Етап 1: Прийняття рішення про масштабування

Для менш мінливої інтенсивності запитів достатньо фіксованої кількості екземплярів сервісів, а споживання ресурсів може бути обмежене стабільним діапазоном. Однак, в реальності, нестабільний трафік є нормою. При збільшенні трафіку, що надходить на веб-систему, починає працювати HPA, який доводить кількість екземплярів сервісу до достатньої кількості. Таким чином, вся процедура автоматичного масштабування починається з виявлення трафіку і прийняття рішення про масштабування, як показано на рисунку 11.

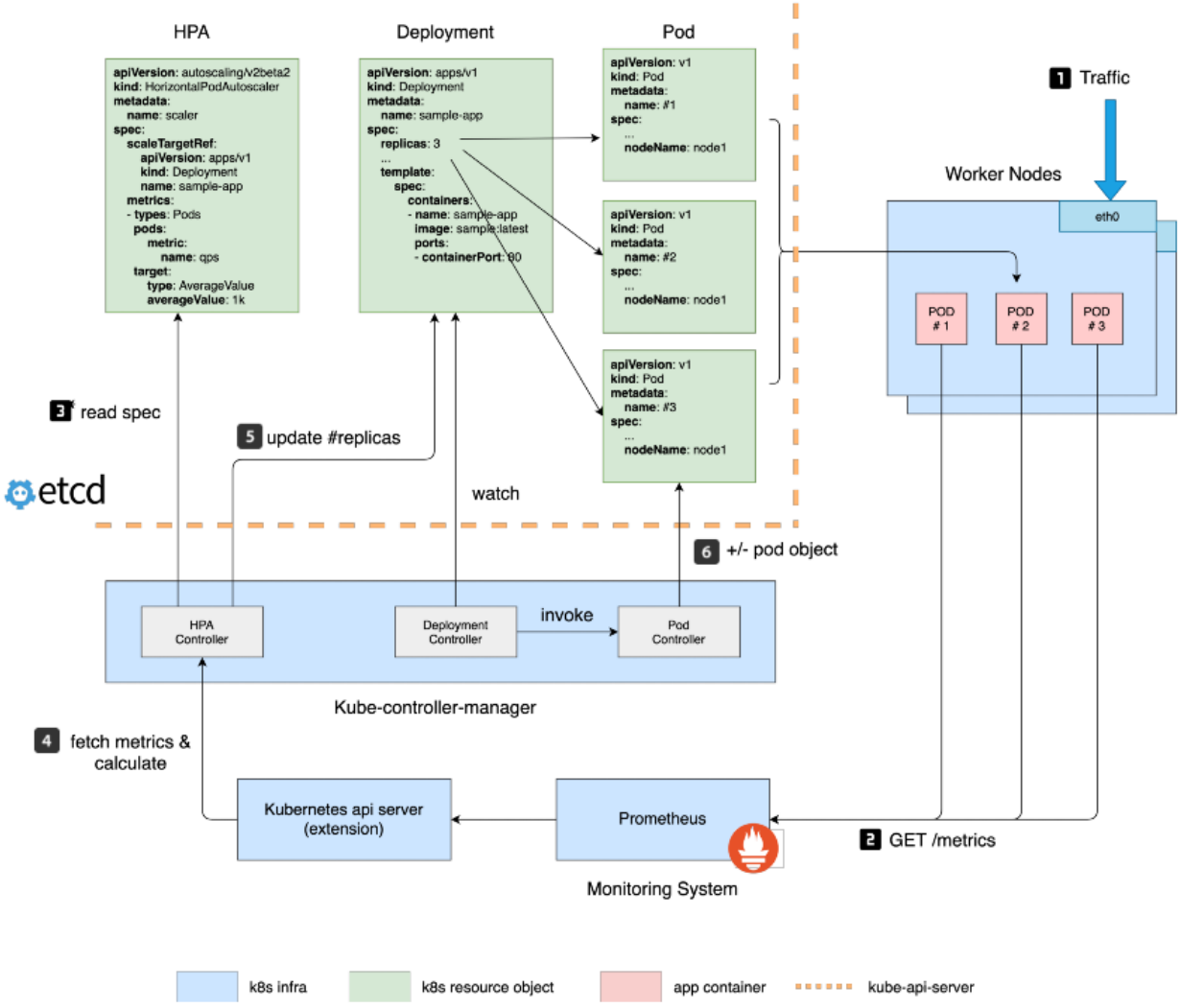


Рисунок 11 – Процес прийняття рішення про масштабування

Під час обслуговування запитів на робочих вузлах система моніторингу безперервно відстежує та фіксує кількість HTTP-запитів, що надходять до них. Система моніторингу Prometheus Система моніторингу Prometheus використовує метод витягування для отримання метрик з кінцевих точок HTTP, що піддаються впливу під впливом ботів. Інтервал вилучення за замовчуванням становить одну хвилину, що означає, що існує системна затримка в одну хвилину у виявленні сплеску трафіку.

На головних вузлах Kubernetes контролер HPA, що входить до складу kube-controller-manager, є мозком операції автоматичного масштабування. Він працює з об'єктами ресурсів HPA, що зберігаються в etcd. Об'єкт HPA регулює поведінку автомасштабування за метрикою QPS, щоб середнє значення QPS для всіх вузлів було нижче заданого значення. У об'єкті, структура якого показанана рисунку 10, контролер HPA періодично отримує метрики QPS протягом трьох хвилин (див. вираз PromQL в розділі 3.1.2) з бази даних часових рядів Prometheus і обчислює цільову кількість реплік для масштабування.

Інтервал вибірки налаштовується за допомогою параметра flag -horizontal-pod-autoscalersync-period kube-controller-manager (за замовчуванням: 15s). Метрики будуть переведені у формат API метрик Kubernetes за допомогою адаптера перед тим, як вони будуть подані в контролер HPA. Випадки в точках - це відсутність достатньої кількості готових сервісних екземплярів, щоб контролер HPA розкручував додаткові pods.

Після визначення кількості реплік для масштабування контролер HPA змінює специфікацію робочого навантаження додатка, наприклад, поле spec.replicas в Deployment. Контролер розгортання, в свою чергу, створює відкладені об'єкти Pods в etcd для планування. Нарешті, перший етап завершується створенням додаткових об'єктів Pod.

Новостворені блоки використовують той самий шаблон налаштування, що й інші запущені брати і сестри, а різниця полягає в тому, що їх поле nodeName є

порожнім, яке буде призначено на наступному етапі. Таким чином, прийняття рішення про масштабування складається з 6 наступних кроків:

1. Зростаючий трафік досягає контейнерів, і очікується, що Кубернейт вживатиме заходів для згладжування трафіку.

2. Kubernetes балансує трафік між блоками. Кожен блок підраховує вхідні запити та виставляє метрики кількості запитів на кінцевих точках /prometheus.

Ці показники періодично зчитуються та агрегуються Prometheus:

3. Контролер HPA звертається до ectd для об'єктів ресурсів HPA, визначених оператором кластера, щоб отримати інформацію про політику масштабування.

4. Контролер HPA отримує метрики і обчислює новий номер репліки для масштабування на основі заданих специфікацій.

на основі заданої специфікації.

5. Після прийняття рішення контролер HPA перезаписує поле репліки робочого навантаження.

6. Контролер розгортання та контролер модулів співпрацюють для створення нових об'єктів модулів.

3.3.2 Етап 2: Прив'язка вузла до вузла

На цьому етапі контейнерний блок призначається робочій машині. Після того, як нові об'єкти контейнерів були створені, настав час вибрати робочі вузли для розміщення цих контейнерів. Контейнери, яким не був призначений вузол, називаються відкладеними контейнерами. Куб-планувальник, що є частиною площини керування, відповідає за планування відкладених подів на найкраще відповідний робочий вузол із запропонованих. На рисунку 12 відображено, як працює прив'язка між вузлами в системі Kubernetes.

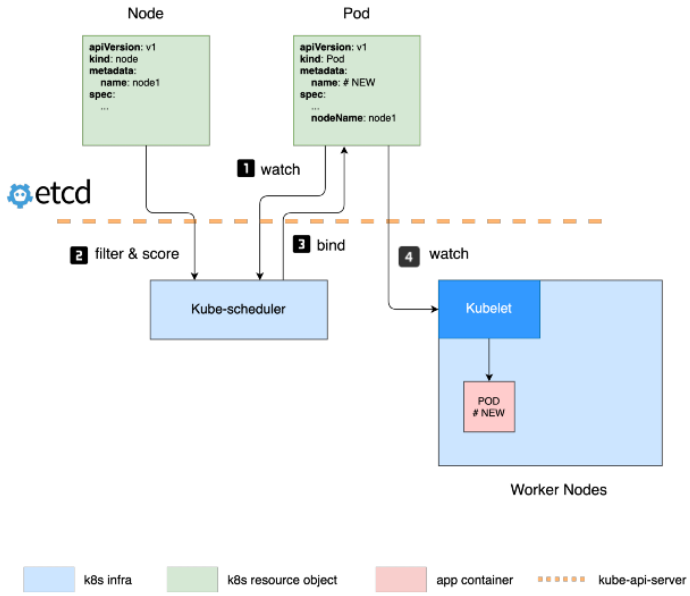


Рисунок 12 – Робочий процес прив'язки вузла контейнера

Кубернейт-планувальник відстежує новостворені об'єкти pod шляхом запиту до etcd через kube-apiserver. Кубернейт-планувальник виконує 2-етапний процес вибору кваліфікованого вузла, який складається з фільтрації та оцінки контейнера.

На етапі заповнення куб-планувальник шукає та оцінює вузли-кандидати, щоб перевірити, які вузли задовольняють специфічним вимогам розміщення пакета. Різні робочі навантаження можуть мати різні вимоги до їх розміщення. Наприклад, під запитують процесор/пам'ять, мають перевагу щодо приналежності під або віддають перевагу вузлам з певними мітками. Ці вимоги повинні бути виконані в першу чергу. Вузли, які проходять фазу заповнення, називаються можливими вузлами. На етапі оцінки кубічний планувальник оцінює та ранжує всі можливі вершини. Деякі вузли можуть бути кращими за інші, оскільки вони виконують менше робочих навантажень або локально кешують зображення контейнерів. Вузол з найвищим рейтингом буде обраний для розміщення відкладених капсул.

Для кластера великого розміру цей процес вибору може зайняти деякий час. Після 2-етапної операції, планувальник куба змінює поле nodeName контейнера таким чином, щоб відповідні робочі вузли могли помітити цю інформацію і

запустити ініціалізацію контейнера на наступному етапі. Таким чином, прив'язка контейнера до вузла складається з чотирьох кроків:

1. Куб-планувальник слідкує за нерозподіленими подами.

2. Kube-планувальник запускає процес фільтрації та оцінки, щоб вибрати оптимальні вузли для хостингу подів.

3. Kube-планувальник оновлює поле nodeName (ім'я вузла) для подів.

4. Kubelet, агент робочого вузла, помічає нові поди, призначені для вузла, яким він керує.

3.3.3 Етап 3: Прив'язка вузла до контейнера

Контейнер – це максимально мінімізоване середовище для запуску додатків, яке відмінно підходить для розгортання мікросервісів. У Kubernetes вузловий агент, kubelet, відповідає за ініціалізацію контейнера. Kubelet дає вказівки середовищу виконання контейнера налаштувати простори імен Linux і розділити ресурси ядра.

Після Етапу 2, kubelet періодично звертається до etcd щодо запланованих контейнерів та оновлює статус розміщених контейнерів від імені робочого вузла. Якщо контейнер призначено вузлу, але ще не створено, kubelet викликає середовище виконання контейнера через інтерфейс виконання контейнерів (CRI), щоб підготувати середовище виконання коду. Контейнерне середовище виконання є замінним у Kubernetes і має бути попередньо встановлене у двійковому форматі. Деякі контейнерні середовища виконання за замовчуванням призначені для взаємодії з CRI, в той час як інші, такі як Docker, вимагають процесу shim для подолання розриву.

Контейнерне середовище виконання спочатку перевіряє, чи має хост локальний кеш зображень. В іншому випадку, середовищу виконання контейнера необхідно отримати копію з реєстру образів через мережу, що займає багато часу і затримує готовність модулів. Образ містить вихідний код програми, бібліотеки та конфігурацію. Вузол не завжди зберігає файли образів через механізм збору сміття для зменшення використання диска. За допомогою образу середовище виконання

контейнера створює простори імен ядра Linux для ізольованих середовищ додатків. До цього моменту прикладні процеси в контейнері отримують незалежний мережевий стек, точки монтування, управління процесами далеко від хоста.

Тим часом, kubelet викликає CNI-плагіни для приєднання та підключення мережевих кінцевих точок як всередині контейнера, так і на хості. Відповідальність плагінів CNI включає в себе призначення IP-адрес подів і оновлення таблиць маршрутів, щоб піди могли спілкуватися із зовнішнім світом. З цього моменту контейнер готовий до роботи, а його статус зміниться на "Працює", хоча потрібен додатковий час для проходження перевірки працездатності, перш ніж він зможе серйозно обслуговувати трафік, який буде представлене на заключній стадії. Коротко представлено операцію ініціалізації контейнера наступним чином:

1. Kubelet помічає призначені йому нові створені под, потім викликає CRI-сумісні середовища виконання контейнерів, щоб підготувати середовища для под.
2. Контейнерне середовище виконання перевіряє наявність образів контейнерів і витягує копію з реєстру образів у разі відсутності в кеші.
3. Виконавче середовище контейнера налаштовує простори імен ядра Linux та запускає контейнер.
4. Kubelet викликає плагіни CNI для підключення мережі до контейнерів.
5. Kubelet видає подію запуску контейнера.

3.3.4. Етап 4: Запуск програми

Контейнер ще не готовий до обслуговування трафіку. На останньому етапі Kubernetes чекає, поки буде завантажено код програми і контейнер пройде перевірку готовності. Під час завантаження коду додатку модуль менеджера зондів в kubelet періодично перевіряє стан готовності контейнера за допомогою визначеного користувачем зонду готовності. Це може бути реалізовано додатком, що розкриває свій стан здоров'я через кінцеві точки HTTP API.

Після того, як под пройде перевірку готовності, kubelet оновить ресурсний об'єкт Endpoint із зазначенням IP-адреси, порту та протоколу зв'язку. В результаті,

всі kubeproxy помітять цю зміну і відповідно оновлять правила фільтрації та переадресації трафіку в iptables. Нарешті, модуль може почати обслуговувати трафік. Рисунок 13 ілюструє весь робочий процес на цьому етапі.

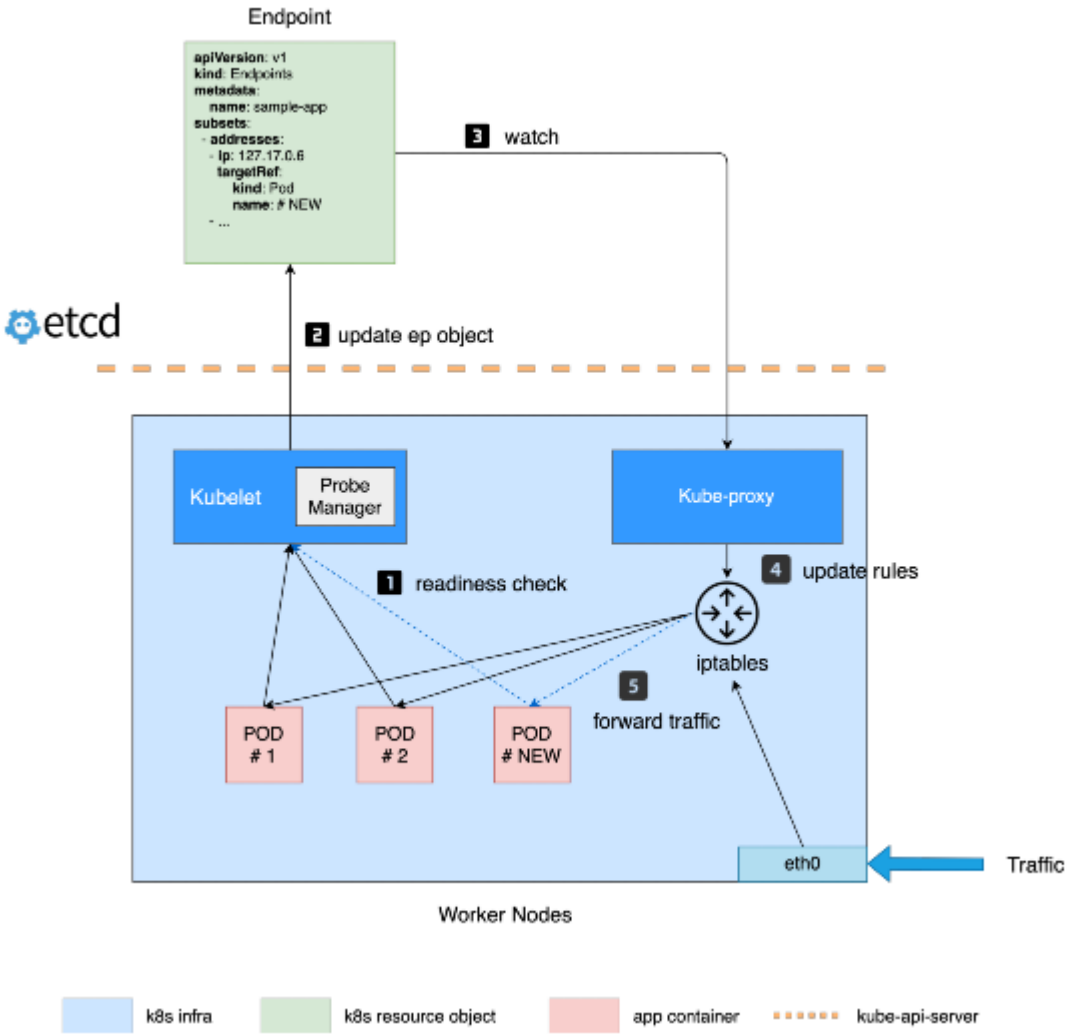


Рисунок 13 – Процедура запуску програми

Від часу, витраченого на цей етап, значною мірою залежить продуктивність запуску додатків. Підсумовуючи, останній етап можна розбити на чотири кроки:

1. Probe Manager періодично перевіряє стан готовності капсули по мірі завантаження коду додатку.
2. Як тільки готовність пройдена, kubelet оновлює об'єкт Endpoint і додає IP-адресу для новоствореного бода.
3. Куб-проксі помічає зміни в Endpoint.

4. Kube-proxy на всіх вузлах оновлює правила iptables на основі інформації про Endpoint.

5. Вузли починають приймати трафік. Процес автомасштабування завершується.

3.4 Гіпотетичні фактори

На основі аналізу етапів холодного старту було визначено п'ять гіпотетичних факторів, які можуть вплинути на ефективність автоматичного масштабування. Ці п'ять факторів охоплюють вищезгадані чотири етапи холодного старту. Для кожного фактору була аргументована причина та обговорено вибір технічного рішення, далі ще раз будуть розглянуті ці фактори, оцінено вплив різних рішень на продуктивність і порекомендовано вибір технології для розгортання додатків на налаштованих кластерах Kubernetes з метою гнучкого автоматичного масштабування для фахівців з мікросервісу.

Фактор 1. Сервісна залежність. Мікросервісна архітектура допомагає розбити програмне забезпечення на слабко пов'язані між собою компоненти. Все ще може існувати взаємозв'язок між висхідним та низхідним потоками, коли низхідний сервіс залежить від віддачі від висхідного [51]. Якщо на висхідний сервіс надходить перевантажений трафік, додатковий трафік може генеруватися всередині висхідного сервісу і швидко спричиняти тиск навантаження на нього. Тому, як і функціональна залежність в безсерверних системах, необхідно бути особливо уважним, щоб полегшити координацію мікросервісів при автомасштабуванні.

Існуючий автомасштабувальник, HPA, масштабується тільки на одну цільове робоче середовище. Потрібно розглянути можливість координованого масштабування сервісів і впровадити вдосконалений автомасштабірувальник, який виконує автомасштабування декількох робочих навантажень паралельно.

Фактор 2. Розмір кластера. На етапі зв'язування вузлів між собою витрати часу на планування вузлів залежать від розміру кластера відповідно до алгоритму планування Kubernetes. Поточне рішення для налаштування продуктивності

планування полягає у встановленні порогу оцінки вузла [28], щоб куб-планувальник міг припинити пошук можливих вузлів раніше, як тільки він побачить достатню кількість кандидатів. Візьмемо для прикладу кластер з 5000 вузлів. Якщо Kubernetes знайде 10% вузлів, які задовольняють вимогам до розміщення, він припинить перевірку інших вузлів і негайно перейде до етапу оцінювання.

Фактор 3. Середовище запуску контейнера. Час виконання контейнера відповідає за створення середовища запуску для додатків на етапі 3. Kubernetes розроблено для підтримки декількох середовищ виконання контейнерів за допомогою інтерфейсу середовища виконання контейнерів. CNCF інкубує різні CRI-сумісні середовища виконання контейнерів для різних цілей (див. Рисунок 14). Найбільш відомими середовищами виконання контейнерів є Docker, оскільки він є першопрохідцем в епоху Cloud Native.

Kubernetes інтегрується з Docker за допомогою процесу shim, який вводить додаткову затримку запуску. На основі CRI-O, власної реалізації для роботи з Kubernetes, з'являється ощадливе середовище виконання контейнерів, containerd, яке слідує за CRI-O. Крім того, було запропоновано контейнер kata з покращеною безпекою контейнерів. Необхідно подбати про те, щоб вибрати правильний час виконання контейнера для різних діапазонів додатків, особливо коли проводиться обговорення проблеми холодного старту.

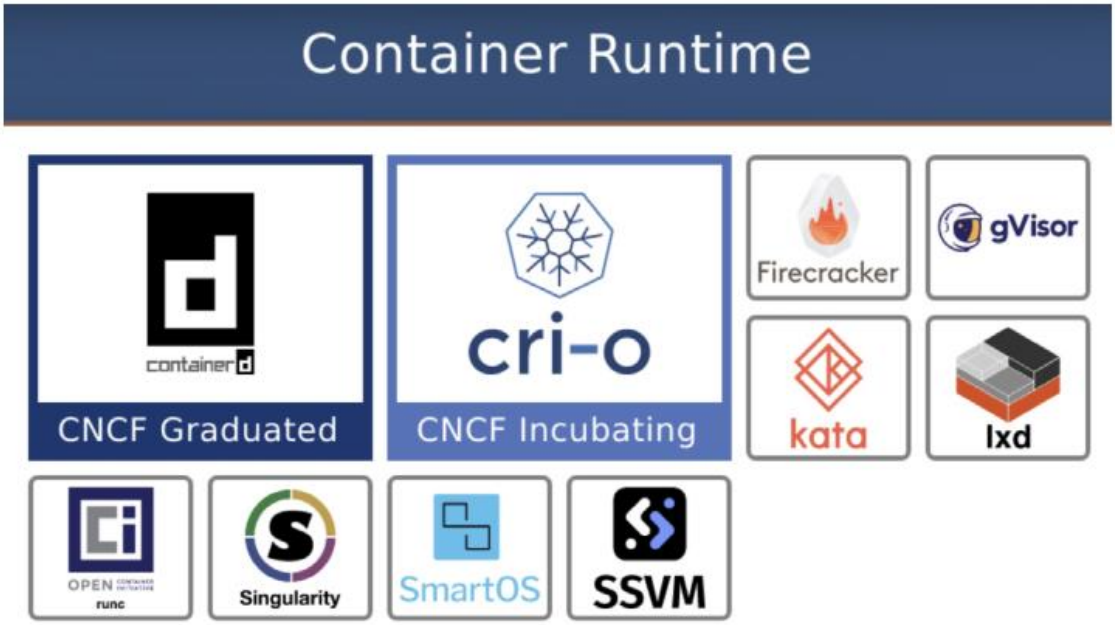


Рисунок 14 – Середовища для запуску контейнерів

Фактор 4. Налаштування мережі контейнера. Налаштування мережі є ще одним фактором, який, може бути спільно відповідальним за затримку холодного старту на етапі ініціалізації контейнера. Kubernetes делегує завдання розгортання мережі плагінам CNI. Загальні обов'язки CNI включають приєднання мережевого інтерфейсу до контейнерів, підключення контейнера до стеку хост-мережі через віртуальну ethernet (veth) пару, управління IP-адресами та підтримку маршрутів. Як і контейнерні середовища, CNCF інкубує низку мережевих рішень CNI. Різні плагіни використовують різні технології мережевої віртуалізації і підтримують різні функції. На рисунку 14 показано знімок з ландшафту CNCF, що відображає активні рішення CNI в рамках CNCF. Виконана оцінка кілька широко прийнятих плагінів CNI, щоб з'ясувати, як вибір рішення CNI може вплинути на ініціалізацію контейнера.

Фактор 4. Мова програмування. Останнім фактором, який було розглянуто, є мова програмування. Оцінено, як різні варіанти мови програмування впливають на завантаження коду. Довга затримка ініціалізації може потенційно затримати проходження перевірки готовності додатку. Зразок додатку, який було

протестовано, написаний на Java з використанням фреймворку Spring Boot. Java має розвинену екосистему та отримує активну підтримку від спільноти.

Нові мови програмування, що з'являються, привносять свіжі принципи проектування. Наприклад, мова Go, вперше випущена в 2012 році, вводить концепцію goroutine, яка за функціями є аналогом потоку ОС на рівні ядра, але має зростаючий стек пам'яті і містить власний планувальник в користувацькому просторі, отже, зменшує накладні витрати на паралелізм [13]. Go також є мовою для Docker, Kubernetes та багатьох інших проектів з відкритим вихідним кодом в епоху Cloud Native. Варто порівняти поширені мови бекенд-розробки в контейнерних середовищах.

Можна підсумувати п'ять гіпотетичних факторів наступним чином.

Таблиця 3.1 – Підсумок гіпотетичних факторів

Гіпотетичні фактори	Задіяна стадія	Технічне рішення
Сервісна залежність	Масштабування прийняття рішень	узгоджене автомасштабування
Розмір кластера	Зв'язування між вузлами	малий, середній, великий
Середовище запуску контейнера	Ініціалізація контейнера	Docker, containerd, cri-o
Налаштування мережі	Ініціалізація контейнера	Calico, Flannel, Weave
Мова програмування	Запуск додатку	Java, Go, JavaScript

3.5 Конструкція координованого масштабувальника

У цьому розділі запропоновано координований горизонтальний автомасштабувальник (CHPA) для розширення функціональності HPA. Мета полягає в тому, щоб координувати прийняття рішень щодо автоматичного масштабування на наборі мікросервісів, які мають взаємозв'язок висхідного і

низхідного потоків. CHPA – це рішення даного дослідження для гіпотетичного фактору "залежність від сервісу" (Service Dependency).

Як точка відліку, HPA представляє собою рольову модель для проектування автомасштабування. Горизонтальний автомасштабувальник складається з об'єктів ресурсів HPA і контролера HPA. Об'єкт ресурсів визначає бажаний стан цілей масштабування, наприклад, очікувані метричні значення і поведінку масштабу. Контролер є контуром управління, який зчитує об'єкти HPA та реалізує логіку автоскалера, включаючи періодичне отримання метрики, визначення напрямків шкали та пропонування нових номерів реплік. Поєднання ресурсів Kubernetes та контролера добре відоме як патерн Operator [20]. Спільнота надає kubebuilder для створення проекту оператора для користувацьких ресурсів, визначених користувачем.

Філософія проектування Оператора застосовується для CHPA. У наступних розділах спочатку детально розглядається визначення користувацького ресурсу CHPA, потім пояснюється логіка роботи контролера CHPA та обговорюється застосування CHPA. Виконана оцінка та порівняння його продуктивності з HPA в Розділі 4.

3.5.1 Структура CHPA

Ресурс CHPA визначає умови та поведінку для виконання автомасштабування на цільових робочих навантаженнях. Рисунок 16 ілюструє приклад об'єктів ресурсу CHPA. Ресурси CHPA використовують розширюваність Kubernetes, надаючи декларативні API для користувачів для визначення бажаного стану масштабованих робочих навантажень. Об'єкти API зберігатимуться у сховищі даних etcd. Крім того, власне процес автоматичного масштабування буде завантажений на контролер CHPA і буде агностичним до кінцевих користувачів.

По суті, ресурси CHPA нічим не відрізняються від інших вбудованих ресурсів Kubernetes, таких як Deployment або Pod. Контролер CHPA може використовувати розширені API Kubernetes для отримання об'єктів CHPA.

Об'єкт ресурсу CHPA складається з трьох частин: метадані, специфікація і статус. Поле метаданих ідентифікує унікальний об'єкт CHPA по імені та області простору імен. У полі специфікації перелічуються визначені користувачем масштабні цілі, які потребують синхронного масштабування вгору та вниз, а саме групи автомасштабування. Структура координованого горизонтального масштабувальника показана на Рисунку 15.

Для кожного елемента в масиві структура даних подібна до тієї, що ми бачили в HPA, при цьому вводиться додаткове підполе weight, щоб описати вагу робочого навантаження в потоці трафіку. Наприклад, зразок об'єкта CHPA відображає випадок, коли два навантаження A та B повинні масштабуватися в одному напрямку та з однаковим кроком, оскільки вони мають однакову вагу.

Коли трафік досягає робочого навантаження, перше робоче навантаження, яке повідомляє про сплеск попиту, запускає процес автоматичного масштабування і об'єднує свої однорангові навантаження для спільного масштабування, навіть якщо відстаюче навантаження ще не досягло свого піку. CHPA використовує відображення ваги для координації автоматичного масштабування.

```yaml
---
apiVersion: chpa.my.domain/v1
kind: CHPA
metadata:
  name: chpa-sample
spec:
  # defines auto-scaling group
  scaleTargets:
  - scaleTargetRef:
      apiVersion: apps/v1
      kind: Deployment
      name: workload-a
    # metric name and desired value
    metric:
      name: qps-a
      targetValue: 1000
    # weight mapping in the data flow
    weight: 1
  - scaleTargetRef:
      apiVersion: apps/v1
      kind: Deployment
      name: workload-b
    metric:
      name: qps-b
      targetValue: 1000
    weight: 1
status:
  lastScaleTime: 2021-06-23T09:45:26Z
  scaleTargets:
  - scaleTargetRef:
      apiVersion: apps/v1
      kind: Deployment
      name: workload-a
      # replica count since last scale
      replicas: 2
  - scaleTargetRef:
      apiVersion: apps/v1
      kind: Deployment
      name: workload-b
      replicas: 2
```

Рисунок 15 – Приклад структури CHPA

Поле стану призначене для запису контролером CHPA історії масштабування та останнього рішення про масштабування. У разі прийняття рішення про масштабування, контролер CHPA виконає масштабування та оновить поле статусу. Якщо пропозиція репліки еквівалентна поточному номеру репліки, модифікація

буде пропущена. Користувачам не дозволяється змінювати поле статусу. Кінцеві користувачі повинні в основному взаємодіяти з полем специфікації.

3.5.2 CHPA контролер

Контролер CHPA реалізує логіку автомасштабування над об'єктами CHPA. Контролер CHPA консультується зі сховищем etcd для об'єктів CHPA, щоб визначити, яке робоче навантаження масштабувати і які метрики оцінювати. У разі масштабування, CHPA коригує репліки робочих навантажень шляхом модифікації поля репліки. Було взято кілька уроків у HPA. Окрім залучення механізму координації, контролер CHPA реалізує подібні стратегії зменшення масштабу-затримки для згладжування метричних коливань. Крім того, встановлено інтервал послідовного оцінювання на 15 с за замовчуванням, як це робить HPA. У наступних підрозділах виконаний опис трьох основних кроків логіки координованого автоматичного масштабування.

Отримання метрик. У кожному періоді оцінки контролер CHPA отримує доступ до об'єктів CHPA для отримання списку останніх цільових масштабованих робочих навантажень. На основі ідентифікації робочого навантаження (комбінація Kind, Version, Name) та визначених метрик, контролер CHPA надсилає запити до API метрик Kubernetes для отримання актуальних значень метрик, таких як QPS.

В результаті було отримано список метрик подів для кожного робочого навантаження, в якому детально описується ім'я запущених подів, керованих робочим навантаженням, і значення метрики, яке спостерігає под. Матриця метрик буде оцінена для створення нової пропозиції щодо реплікації в кінці цього раунду оцінювання.

Прогнозування реплік. Як і HPA, кожне робоче навантаження пропонує нову кількість реплік для себе відповідно до отриманої метрики. Крім того, CHPA також дозволяє робочому навантаженню пропонувати кількість реплік для своїх аналогів на основі попередньо встановленого відображення ваги. Зачатки цього розрахунку відображені в проілюстровані в рівняннях 3.2 і 3.3. ReplicaP позначає кількість

реплік, запропоновану і-м робочим навантаженням для себе. Коефіцієнт масштабу отримується шляхом ділення загального значення метрики на цільове значення метрики.

Наприклад, розглядаючи і-те робоче навантаження, яке в даний час має два запущені модулі, що звітують про 2000 qps в цілому, і ми очікуємо, що середнє значення qps для кожного модуля становить 1000. Таким чином, коефіцієнт масштабу буде дорівнювати 2, а пропозиція щодо реплікації буде дорівнювати 4. Для блоків, які ще не почали звітувати про показники, ми припускаємо, що в розрахунках значення буде дорівнювати 0.

Оскільки всі робочі навантаження пропонують репліки, ми використовуємо ці пропозиції для розрахунку репліки для інших рівних. Як ілюструє рівняння 3 і 4, це здійснюється за допомогою попередньо встановленого відображення ваги. NewReplicaj позначає остаточний номер репліки для j-го робочого навантаження. Максимальна пропозиція буде обрана як остаточне рішення щодо масштабування. Якщо нова репліка більша за поточну репліку робочого навантаження, це означає зменшення масштабу, інакше – збільшення.

$$ReplicaProposal_i = ReadyPodCount_i * \frac{TotalMetricValue_i}{TargetMetricValue_i} \qquad (3)$$

$$NewReplica_j = ReplicaProposal_i * \frac{Weight_j}{Weight_i} \qquad (4)$$

Затримка при зменшенні масштабу. При роботі з масштабуванням треба бути обережним, оскільки метрики є мінливими за своєю природою. Щоб згладити варіації, було застосовано стратегію затримки зі зменшенням масштабу, подібно до того, як це робить HPA. Для зменшення масштабу завжди бажано вживати заходів негайно, в той час, як цілеспрямовано затримується зменшення масштабу, щоб уникнути різких стрибків масштабу. За задумом цього дослідження, якщо нова пропозиція репліки полягає в зміні напрямку масштабування з верхнього на

нижній, оцінюється останній час масштабування, що зберігається в полі стану. Якщо різниця менше 5 хвилин, то ігнорується це зменшення масштабу. Лише перше зменшення масштабу через 5 хвилин призведе до поступового опускання капсул вниз. У разі повторного піку трафіку, зменшення масштабу відбудеться в найближчому раунді оцінювання і поверне модулі до роботи.

3.5.3 Застосування CHPA

CHPA вимагає, щоб користувачі добре розуміли потік даних у своєму додатку мікросервісу, щоб отримати бажане вагове відображення. Це може бути отримано з історичних даних про трафік або добре вивченого потоку викликів API. Визначення гарячої точки і активного потоку даних може допомогти згрупувати робочі навантаження, які повинні синхронно масштабуватися. Перевага використання CHPA проти HPA полягає в тому, що масштабування робочого навантаження не буде відкладатися на останню хвилину через часовий лаг між піком трафіку та метричною звітністю. Відображення ваги не обов'язково повинно бути один на один, але також може бути застосоване до більш складних висхідних і низхідних зв'язків, таких як віялоподібні, ланцюгові та графічні зв'язки. На рис. 16 показані моделі залежності послуг (послуга A знаходиться на чолі потоку даних) та відповідні об'єкти АПВП. Щоб точно налаштувати вагове відображення, кінцеві користувачі можуть просто переглянути числа в об'єктах CHPA.

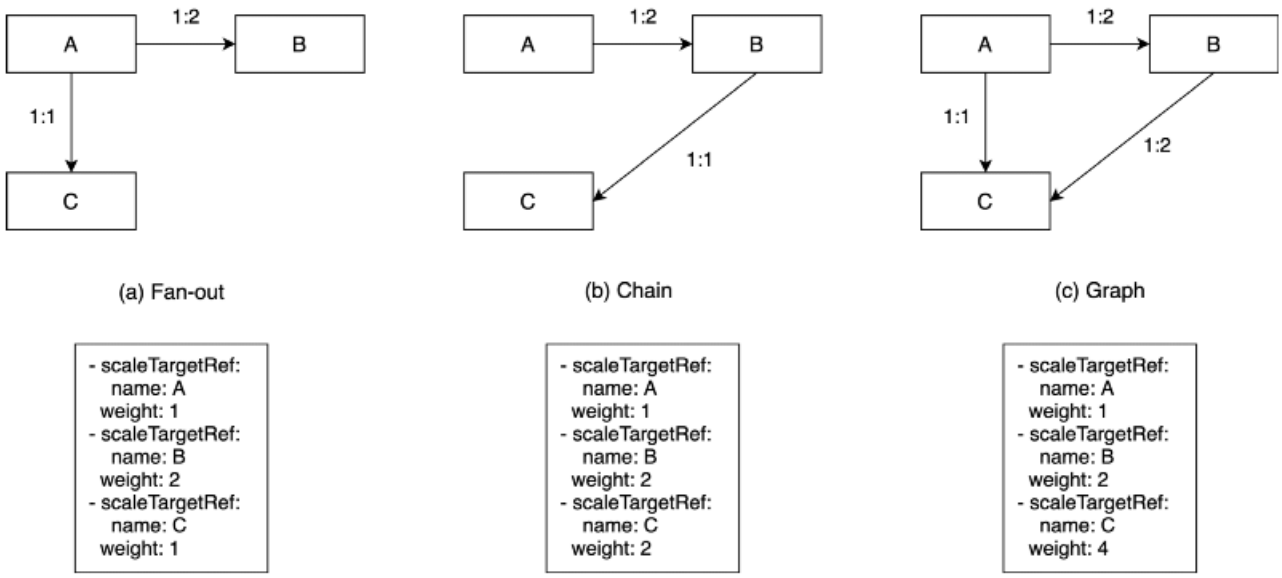


Рисунок 16 – Структура додатків CHPA

3.6 Висновки

У цьому розділі проведено детальний опис етапів холодного запуску програмного забезпечення при автоматичному масштабуванні мікросервісів у Kubernetes. Сформовано ряд гіпотетичних факторів, які мають вплив на швидкодію та час запуску контейнерів під час холодного старту. Проаналізувавши ці фактори, були розглянуті способи для пом’якшення холодного старту, та запроновано скоординований автоматичний горизонтальний масштабувальник.

4 ОФОРМЛЕННЯ СПОСОБІВ ПОКРАЩЕННЯ ШВИДКОДІЇ

У цьому розділі оцінюється продуктивність автомасштабування при холодному старті та досліджується, як різні гіпотетичні фактори та варіанти рішень, представлені в Розділі 3, можуть вплинути на продуктивність та пом'якшити ефект холодного старту. Розділі 4.1 представлено два тести навантаження на додаток Acme Air, щоб отримати уявлення про панораму продуктивності автоматичного масштабування. У Розділі 4.2 розглядається гіпотетичні фактори, щоб визначити їх вплив на продуктивність автоматичного масштабування за допомогою мікробенчмаркінгу та надається рекомендації, яких можуть дотримуватися фахівці з Kubernetes. Як результат, виконується порівняння запропонованого у дослідженні автомасштабувальника, CHPA, з вбудованою пропозицією, HPA, в розділі 4.3.

4.1 Огляд продуктивності автоматичного масштабування

Перш ніж буде заглиблення в оцінку гіпотетичних факторів та запропонованого автомасштабування, дуже важливо мати уявлення про те, як відбувається автомасштабування за допомогою HPA. Тому попередні експерименти в цьому розділі розглядають два сценарії: автомасштабування одного мікросервісу та декількох мікросервісів, які мають взаємозв'язок "висхідний-низхідний" [51].

Для двох експериментів було створено чотиривузловий кластер Kubernetes на AWS з одним вузлом, призначеним головним (див. табл. 4.1). Установчий маніфест містить Kubernetes v1.20.6. Docker v19.03.15 в якості середовища виконання контейнерів, Cilium v1.9.4 для рішення CNI, додаток Acme Air. Кластер встановлено за допомогою kops v1.20.1, інсталятор Kubernetes ентерпрайз класу.

Додатковими компонентами є Prometheus для моніторингу та стек EFK для логування. Попередньо завантажено образи додатків на робочі вузли, щоб

уникнути мережевих затримок, пов'язаних з підтягуванням образів з віддаленого реєстру. Крім того, налаштовано клієнтську машину, розташовану в тій же підмережі, для передачі трафіку на додаток Acme Air, щоб імітувати, як кінцеві користувачі отримують доступ до веб-сервісів. Навантажувальний тест використовує поетапний підхід з використанням JMeter.

Спочатку було створено низьку пропускну здатність протягом п'яти хвилин, а потім миттєво збільшено її до високого рівня, щоб побачити, як HPA і Kubernetes відреагують на сплеск трафіку.

Таблиця 4.1 – Налаштування середовища Kubernetes

Задача вузла	Тип контейнера	CPU/ОЗУ Налаштування
1 Мастер вузл	T3.medium	2 vCPU, 4 GB
3 Робочі вузли	T3.xlarge	4 vCPU, 16 GB

4.1.1 Експеримент 1: Масштабування одного мікросервіса

У першому експерименті досліджувалося автоматичне масштабування на одному мікросервісі, Flight Service, шляхом безперервного виклику API через JMeter. У першій частині тесту на збільшення навантаження дається вказівка JMeter видавати запити зі швидкістю 200 qps і дозволяється додатку працювати з одним блоком протягом 5 хвилин, а потім завантажується трафік до 1200 qps. Оскільки було налаштовано HPA на утримання кожного пакета в середньому нижче 200 qps, HPA повинен ініціювати створення додаткових пакетів, щоб збалансувати попит на трафік.

Аналізуючи дані видно, що за одну хвилину було створено п'ять нових модулів, які успішно запустили службовий трафік. Рисунки 17 (а) та 17(б) ілюструють використання процесора та пам'яті кожного з них. Крім того, слід зазначити, що один з них (з ім'ям suffix chfcn) зазнав різкого збільшення

використання процесора і перевищив встановлений ліміт процесора у 800 міліядер, що призвело до того, що Kubernetes завершив роботу і перезапустив його, як показано на Рисунку 17 (в). Під час перезапуску модуля JMeter зафіксував один запис про збій HTTP-з'єднання.

Рисунок 17 – (а) Швидкість надходження запитів до сервера (qps)

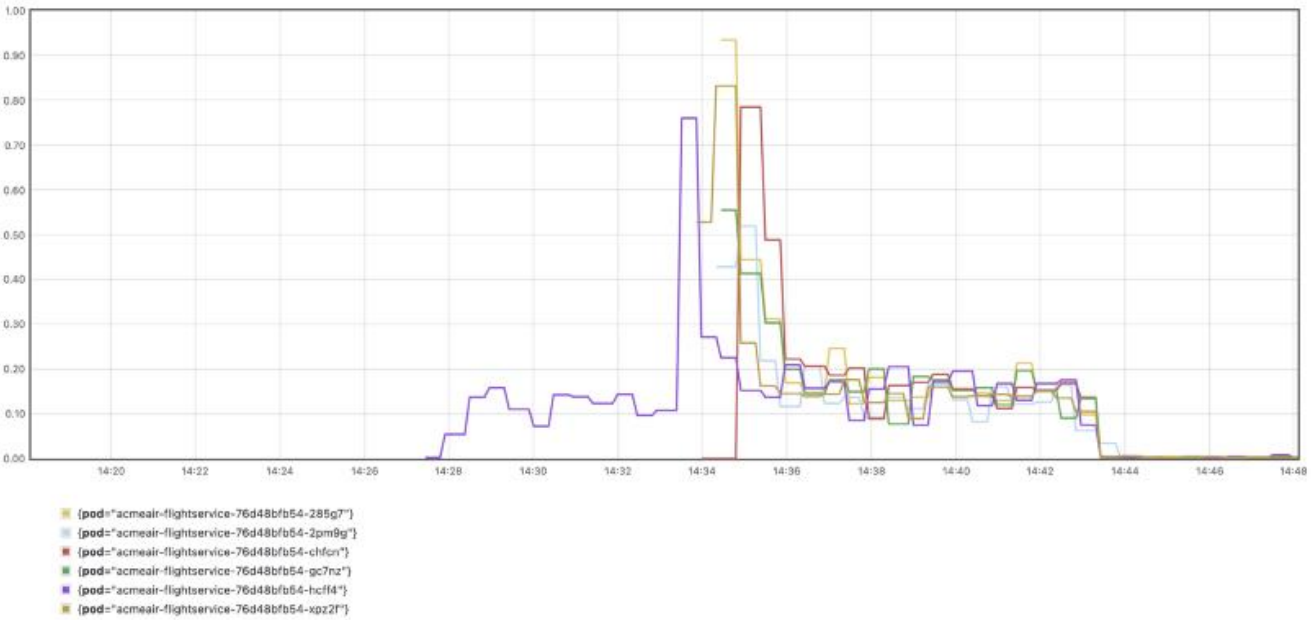


Рисунок 17 – (б) навантаження на процесор

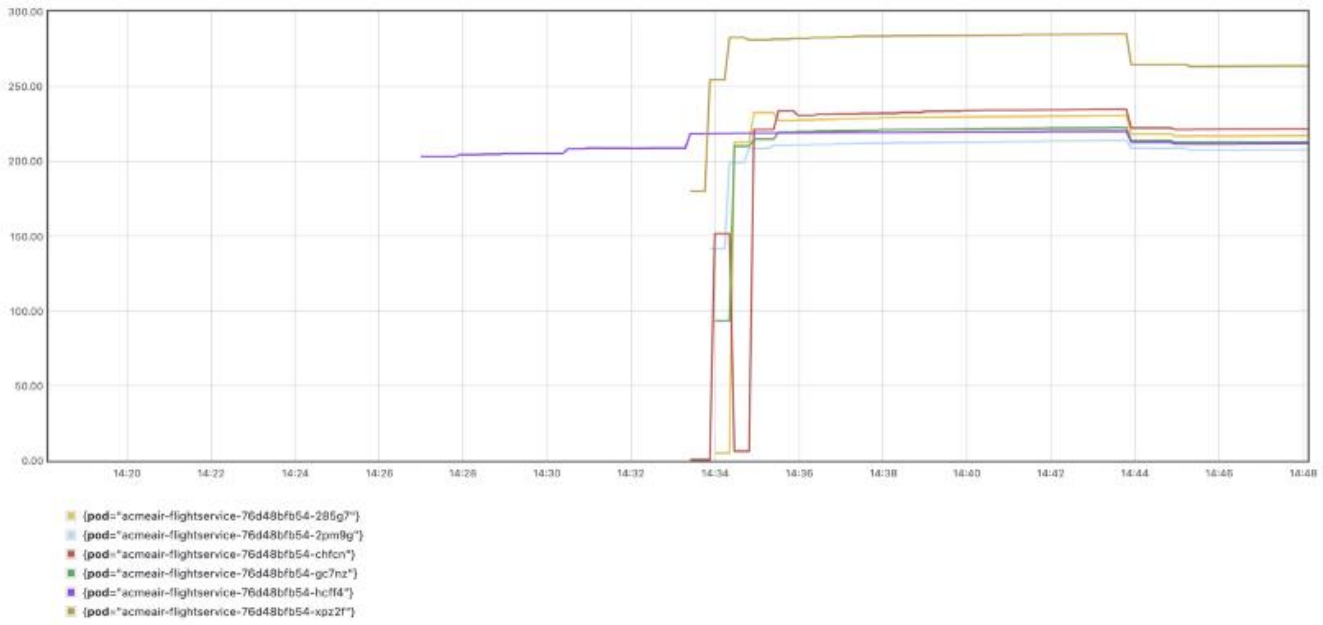


Рисунок 17 – (в) навантаження на оперативну пам’ять

Таблиця 4.2 відображає часовий проміжок для кожного новоствореного підрозділу та структуру цього часу. Процес автоматичного масштабування було розділено на чотири етапи. Шляхом збору та аналізу журналів аудиту визначили, що етап 1 прийняття рішення про масштабування починається з моменту виникнення піку трафіку (зміщення тесту навантаження) і закінчується, коли kube-apiserver успішно перевіряє запити на створення подів. Етап 2, прив'язка pod до вузла, має свою чергу і закінчується, коли kubelet на обраному робочому вузлі помічає рішення про планування за допомогою GET-запиту на створення pod. Етап 3, ініціалізація контейнера, конкурує, коли відбувається подія Started контейнера. Нарешті, стадія 4 запуску додатку охоплює перехід готовності контейнера від false до true за допомогою patch-запиту від kubelet.

З розрахунків видно, що HPA виконує 3-етапне масштабування, спочатку створюючи один новий блок за майже 14 секунд, а потім доводячи загальну кількість блоків до бажаного числа 6 (включаючи вже існуючий блок) за одну хвилину. Це також може бути підтверджено журналами аудиту, пов'язаними з HPA. Крім того, Етапи 1 і 4 складають більшу частину процесу масштабування і працюють у часовому масштабі десятків секунд, в той час як планування

контейнерів та ініціалізація контейнерів працюють у значно коротших часових масштабах, як правило, кілька мілісекунд і одна секунда, відповідно.

Таблиця 4.2: Загальний час, який Kubernetes витрачає на масштабування Flight мікросервіса, з розподілом на блоки та етапи (секунди).

Назва вузла	Етап 1	Етап 2	Етап 3	Етап 4
xpz2f	14.699	0.037	1.125	15.953
chfcn	43.553	0.015	1.139	23.654*
2pm9g	43.435	0.022	1.243	26.236
gc7nz	61.272	0.027	1.137	25.513
285g7	61.311	0.028	1.149	24.154
Середнє	44.7002	0.0256	1.1634	24.348

Тому, щоб пом'якшити проблему холодного запуску, необхідно докласти більше зусиль для покращення першого та четвертого етапів. Ще одним висновком з цього результату є те, що при масштабуванні у Kubernetes може виникнути нестабільний стан, який може призвести до аварійного завершення роботи через періодичне перевищення лімітів ресурсів. Наприклад, обхід відмов для chfcn займає ще 49,88 секунди. Статус подів в моніторі Kubernetes проілюстрований на рисунку 18.

```
ubuntu@ip-172-31-29-48:~$ kubectl get po
NAME                                    READY   STATUS    RESTARTS   AGE
acmeair-flight-db-69df8f87b4-tg7h4      1/1     Running   0          111m
acmeair-flightservice-76d48bfb54-285g7  1/1     Running   0          66s
acmeair-flightservice-76d48bfb54-2pm9g  1/1     Running   0          82s
acmeair-flightservice-76d48bfb54-chfcn  1/1     Running   1          82s
acmeair-flightservice-76d48bfb54-gc7nz  1/1     Running   0          66s
acmeair-flightservice-76d48bfb54-hcff4  1/1     Running   0          25m
acmeair-flightservice-76d48bfb54-xpz2f  1/1     Running   0          112s
```

Рисунок 18 – Статуси вузлів під час масштабування

4.1.2 Експеримент 2: Масштабування декількох мікросервісів

Далі відтворено експеримент з масштабуванням декількох мікросервісів, оскільки за течією – проти течії патерн є більш типовим для потоку даних мікросервісу. В експерименті викликано API бронювання рейсу для резервування рейсу для одного клієнта, який буде запитувати персональну інформацію за рейс та оновлювати загальну кількість милі клієнта. Таким чином, запит на бронювання рейсу буде послідовно викликати сервіс бронювання, сервіс польотів та сервіс обслуговування клієнтів.

Кожна служба виставляє QPS в метричній кінцевій точці Prometheus. Розгорнуто три об'єкти HPA для кожного сервісу відповідно, щоб виконати автоматичне масштабування декількох сервісів. Це пов'язано з тим, що Kubernetes HPA може працювати тільки на одній цілі масштабування. Встановлюється поріг масштабування на рівні 100 qps і збільшується трафікове навантаження від 100 qps протягом перших 5 хвилин до 400 qps протягом решти часу, так що для кожного сервісу потрібно підняти три додаткові pods.

У Таблиці 4.3 представлено загальний час, необхідний Kubernetes для масштабування сервісів у відповідь на високий трафік. Цього разу було зосереджено на Етапі 1. Згідно з результатами, три сервіси не масштабуються синхронно. Сервіс польотів досягає очікуваних реплік раніше за інших, незважаючи на те, що служба бронювання стоїть на чолі потоку даних. Пояснюється це тим, що HPA не підтримує скоординоване масштабування. В результаті, різні робочі навантаження мають непослідовні перспективи автоматичного масштабування, навіть якщо деякі робочі навантаження мають явну залежність. Відсутність скоординованого масштабування залишає простір для вдосконалення.

4.2 Гіпотетичні фактори та мікробенчмаркінг

У Розділі 4.1 було визначено, що час, який пройшов, є неймовірно різноманітним на різних етапах. Щоб підтвердити наші висновки, розглядається оцінка гіпотетичних факторів, представлених в попередньому розділі, і досліджується, чи можуть різні варіанти рішень вплинути на продуктивність масштабування або поліпшити її. Для завершення наступних експериментів треба запобігати тому, щоб представлене середовище з Kubernetes не забезпечувало "повне покриття". В дослідженні дотримується тих же налаштувань AWS EC2 і Kubernetes, що і раніше, за винятком експериментів з розміром кластера, де для робочих вузлів використовується тип EC2 t2.small.

Таблиця 4.3 – Загальний час, який Kubernetes витрачає на масштабування цільових сервісів, з розподілом на блоки та етапи (секунди).

Мікросервіс	Назва вузла	Етап 1	Етап 2	Етап 3	Етап 4
Сервіс Замовлення	kmzg6	31.633	0.043	1.204	22.534
	ncch5	31.646	0.047	1.434	23.65
	gwlqt	45.47	0.021	1.644	20.054
Сервіс Польотів	7hdpg	31.941	0.023	1.355	23.032
	9jvrj	31.828	0.014	1.562	23.743
	jc4gg	31.789	0.016	1.879	20.374
Сервіс Користувачів	h8q5s	31.879	0.022	1.285	17.437
	hpcxb	31.891	0.033	1.997	23.231
	mxgws	59.674	0.026	2.111	21.439

4.2.1 Розмір кластера

У попередніх експериментах було побудовано кластер з 1 вузлом майстром і трьома робочими вузлами і стверджується, що планування вузлів найменше сприяє процесу автомасштабування. Однак наше спостереження базується на крихітному кластері. Щоб завершити наш висновок, оцінюється, як куб-планувальник поводиться на більшому кластері на Етапі 2. Було створено кластери від 10 до 50

вузлів (з одним головним вузлом включно) і відтворено масштабування розгортання Flight Service 100 разів вручну для кожної установки.

Виконано вимірювання тривалості планування вузлів, точно визначаючи незаплановані події подів та заплановані події з журналів куб-шайдерів. За нашими спостереженнями, цей час має тенденцію до зростання, але не є лінійною залежністю від розміру кластеру, як показано на Рисунку 19 Крім того, планувальник Kubernetes має механізм дострокового припинення оцінки можливих вузлів, щоб уникнути перевірки кожного кандидата, коли кластер великий [27].

Однак, незалежно від розміру кластеру, планування роботи вузлів працює в масштабі мілісекунд; таким чином, вплив розміру кластеру на продуктивність масштабування вузлів повинен бути незначним.

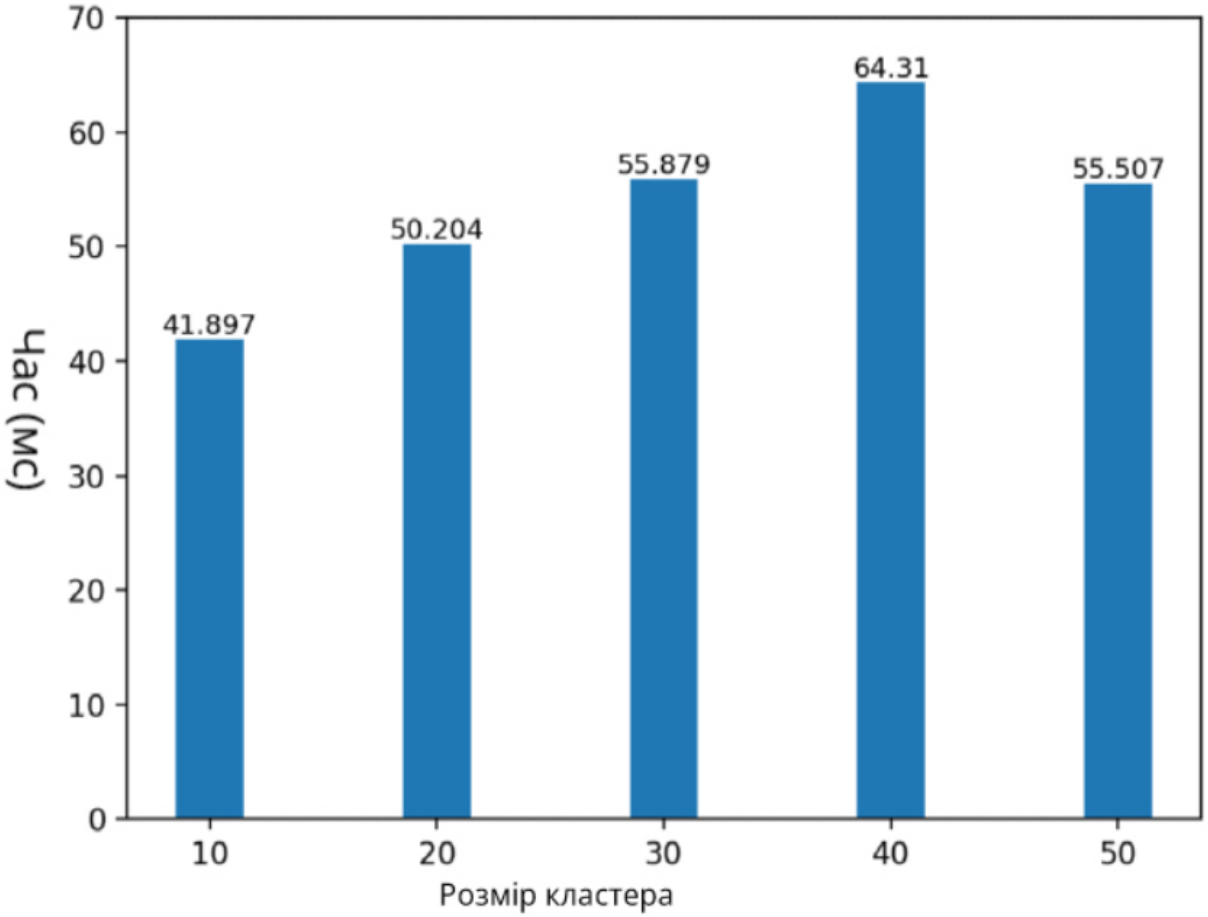


Рисунок 19 – Час на планування вузлів на різних розмірах кластера

4.2.2. Середовище запуску контейнерів

Наступним фактором, який ми оцінюємо, є те, як середовище контейнерів може вплинути на швидкість ініціалізації контейнера. Ми порівнюємо три доступні середовища виконання контейнерів: Docker, containerd, CRI-O. Вони відрізняються реалізацією інтерфейсу виконання контейнерів Kubernetes Container Runtime Interface (CRI). Docker вимагає окремого процесу dockershim для склеювання демона Docker і kubelet, а containerd включає в свій код ту ж функціональність dockershim, що і плагін CRI. CRI-O реалізує CRI за дизайном і може взаємодіяти з kubelet безпосередньо без додаткових накладних витрат. Щоб порівняти їх продуктивність, ми виміряли час запуску контейнера при різних режимах роботи контейнера. Час запуску контейнера – це тривалість від моменту отримання кубелетом новоствореного, призначеного йому контейнера до моменту трансляції події "Початок роботи контейнера" (що охоплює Етап 3). Журнали кубелетів повідомляють про деталі та часові мітки подій. Запуск контейнера в різних середовищах представлено на рис.20

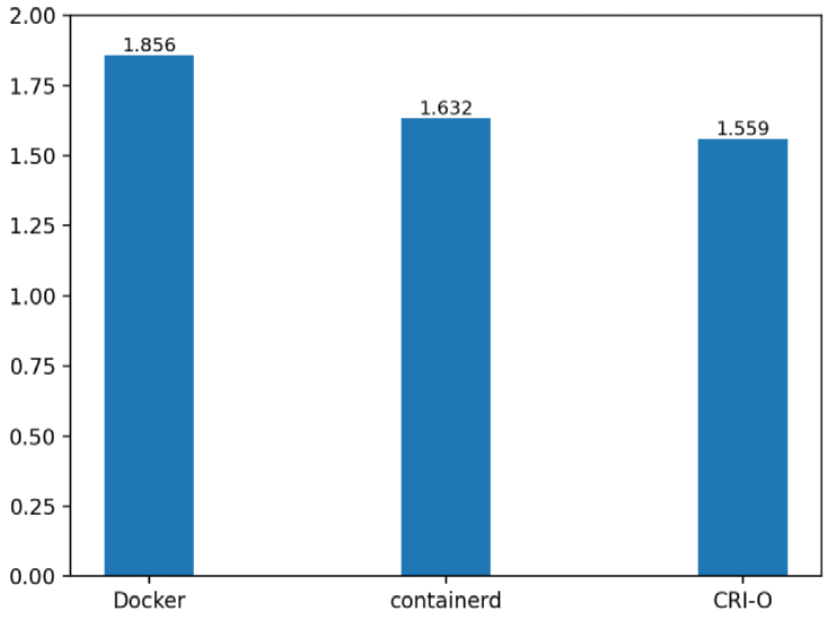


Рисунок 20 – Запуск контейнера в різних середовищах

Проводиться створення контейнерів шляхом ручного масштабування Flight Service в 100 разів. Як видно з діаграми (див. рис. 20), CRI-O має найкращі показники з точки зору затримки запуску контейнера. Однак, покращення є незначним, оскільки всі контейнери можуть запускатися менш ніж за дві секунди. Для цілей гнучкого масштабування різний час роботи контейнерів, як правило, є прийнятним на етапі 3, оскільки він має менше значення для поліпшення холодного запуску в порівнянні з факторами на етапах 1 і 4. Варто зазначити, що Kubernetes застаріла Dockershim починаючи з версії 1.20 і планує незабаром її видалити [26]. Таким чином, за замовчуванням контейнерне середовище виконання для програми встановлення kops є контейнерним для всіх налаштувань Kubernetes v1.20. Крім того, постійно з'являються нові контейнери для виконання. Наприклад, контейнер kata орієнтований на довірене середовище, яке ми не розглядали через її апаратні вимоги до віртуальної технології, а машини AWS не мають підтримки вкладеної віртуалізації [24].

4.2.3 Налаштування мережі контейнера

Третім гіпотетичним фактором, який було розглянуто, є вибір рішень CNI. Плагіни CNI відповідають за приєднання мережі до контейнерів під час ініціалізації контейнера на етапі 3. У попередніх експериментах використовували Cilium для створення мережі. У цьому розділі розширено оцінку запропонованого дослідження і порівнівняно п'ять поширених рішень, перелічених у Таблиці 4.4.

Різні плагіни CNI відрізняються за складністю проекту, мережевою моделлю та підтримуваними функціями. Наприклад, Flannel розглядається як рішення без наповнення, що зосереджується на роботі з мережею без підтримки мережевої політики. Для контейнерної комунікації на одному хості, Flannel, Weave Net і kube-router

Таблиця 4.4 – Варіанти мережевого інтерфейсу

Назва плагіна	Тип мережі		Протокол тунулювання	Версія
	Intra-host	Inter-host		
Cillium	3 рівень	Накладення	VXLAN	1.9.4
Flannel	2 рівень	Накладення	VXLAN	0.13
Calico	3 рівень	Накладення	IP in IP	3.18.3
Weave net	2 рівень	Накладення	VXLAN	2.8.1
Kube-router	2 рівень	Накладення	IP in IP	1.1.1

Для кожного раунду експерименту було замінено плагіни CNI і вручну масштабовано розгортання Flight Service, щоб розрахувати час запуску контейнера і час налаштування мережі. Для вимірювання часу запуску контейнера використовується той самий метод, що і в попередньому розділі. Час налаштування мережі – це тривалість від виклику кубелетом плагінів CNI до моменту визначення IP-адреси контейнера, який можна визначити за допомогою відстеження журналів кубелету. Час налаштування мережі є частиною часу запуску контейнера.

На рисунку 21 показано середній результат 100 повторень для кожного плагіна CNI. Верхній графік показує час налаштування мережі, а нижній – час запуску контейнера. Згідно з верхнім графіком, Flannel та kube-router мають найменші витрати часу, тоді як Cilium є відносно повільнішим за інші. Однак, як видно з нижньої діаграми, загальна різниця між CNI є незначною за часом запуску контейнера. В результаті, зроблено висновок, що різні рішення CNI вносять менший внесок у час запуску контейнера. Основна відмінність між рішеннями CNI в значній мірі полягає в їх обіцяній пропускній здатності вводу/виводу [48, 18]

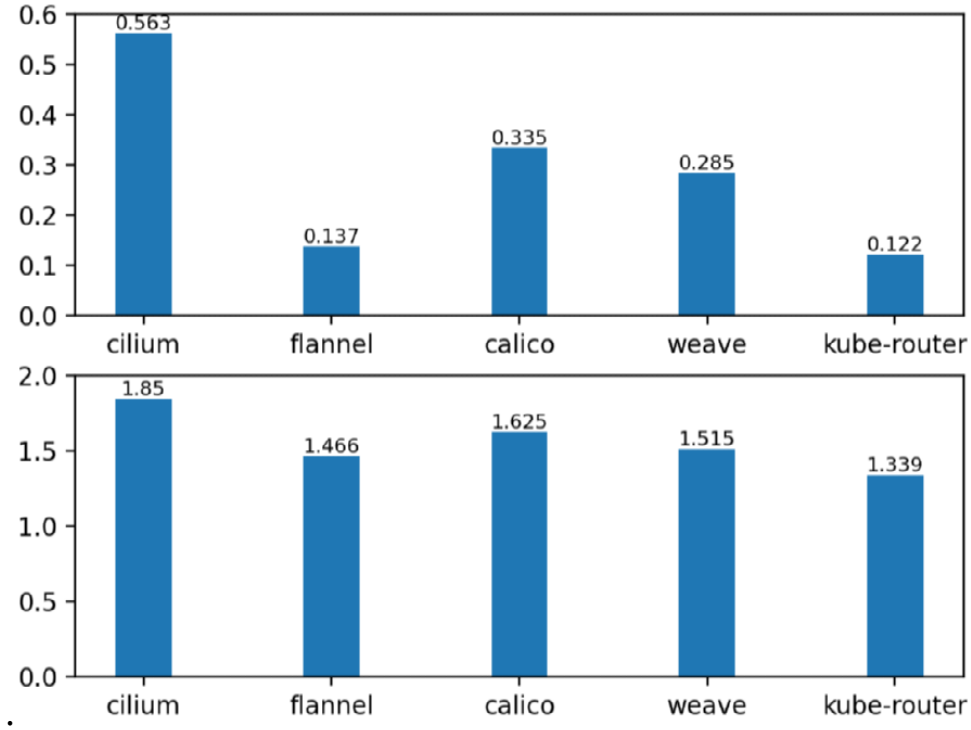


Рисунок 21 – Час запуску контейнера і час налаштування мережі CNI

4.2.4 Мова програмування

Оскільки в розділі 4.1 дійшли до висновку, що команди розробників повинні докладати спільних зусиль з командами інфраструктури для скорочення часу запуску додатків на етапі 4, варто оцінити, як різний вибір стеку мов програмування може вплинути на поставлену в дослідженні мету. Таким чином, порівнюється час запуску програм "Hello, World!", написаних на шести широко розповсюджених мовах програмування.

Було обрано такі мови: Go, Java, C#, JavaScript, PHP та Python. Програма не робить нічого, крім того, що слухає і повертає "Hello, World!". Потім розгортається додаток на Kubernetes і вимірюється час запуску, копаючись в журналах kubelet і обчислюючи різницю в часі між виникненням події Started контейнера і успішним патчем, випущеним kubelet, щоб змінити готовність капсули на true (охоплює Етап 4).

На рисунку 22 представлено середній час запуску на 100 повторень для кожної мови. Відповідно до результату, Java займає найбільше часу для

ініціалізації і приблизно в 2,6 рази повільніше, ніж Go, мова з найкращою продуктивністю в нашому експерименті. Java витрачає значний час на завантаження JVM перед виконанням, чого немає у таких компільованих мовах, як Go. Іншим спостереженням є те, що Python є повільним після Java. Це в першу чергу пов'язано з інтерпретованою та динамічно типізованою природою мови Python [59]. Однак інші інтерпретовані мови, такі як JavaScript та PHP, є швидшими за Python під час завантаження, частково тому, що різні інтерпретовані мови мають різні рівні оптимізації [55]. Крім того, хоча мова Go випереджає всі інші обрані мови, вона трохи швидша за мову C#, яка посідає друге місце.

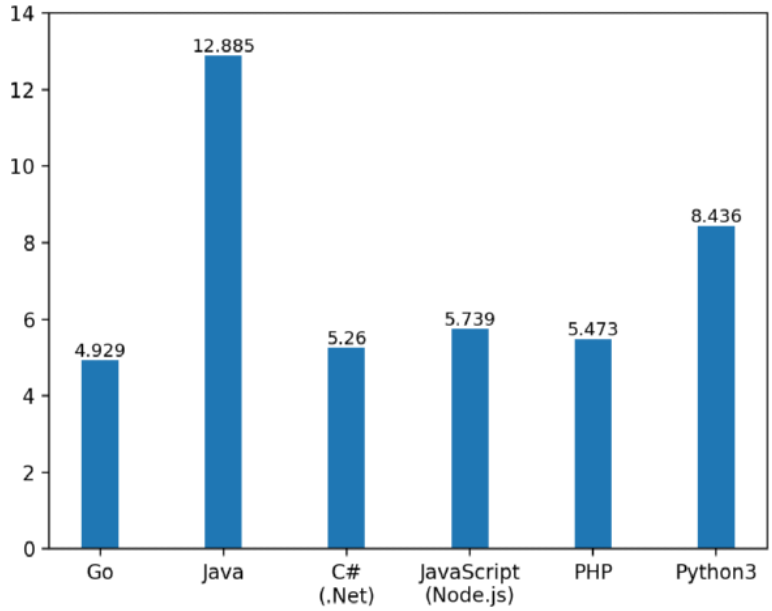


Рисунок 22 – Часові затрати при запуску ПЗ

Таким чином, рекомендовано Go, як варіант для розробки мікросервісних компонентів не тільки тому, що приведений в дослідженні результат демонструє низьку затримку запуску, але й тому, що вона є основною мовою для проекту Kubernetes та отримує широку підтримку з боку спільноти Kubernetes. Дійсно, дебати щодо вибору мови ніколи не закінчуються. Тим не менш, мікросервісна архітектура має перевагу в можливості незалежної розробки та розгортання. Команди можуть поступово мігрувати на потрібну мову, таку як Go, і користуватися перевагами швидкого запуску

4.3 Аналіз координованого автомасштабувальника

Для оцінки запропонованого координованого горизонтального автоскалера (CHPA) було відтворено експеримент 2 в розділі 4.1.2. Проведена заміна об'єкту ресурсу HPA на об'єкт ресурсу CHPA, в якому присвоюється трьом службам однакову вагу 1, оскільки дія бронювання рейсу викликає сервіс бронювання, сервіс польотів та сервіс обслуговування клієнтів один раз послідовно. На відміну від HPA, що є частиною kube-controller-manager в коді, контролер CHPA потрібно розгорнути як Deployment на кластері вручну.

У Таблиці 4.5 показано результат автомасштабування трьох сервісів в рамках CHPA. Порівнюючи стовпчик Етап-1 з аналогічним у Таблиці 4.3, видно, що тепер три сервіси масштабуються синхронно. Процес автоматичного масштабування в HPA займає три послідовних оціночних періоди, щоб мати всі рішення щодо масштабування на місці, тоді як CHPA коштує лише два періоди.

Хоча час створення найпершого блоку залежить від часу отримання метрик системою моніторингу, значно скорочується час створення останніх блоків, оскільки немає відстаючих блоків. Покращення демонструє, що запропонований в дослідженні CHPA може допомогти мікросервісу швидко підготуватися до обслуговування трафіку. Рисунки 23 (а), (б), (в) показують зміну QPS трьох сервісів в цьому експерименті.

На закінчення, залежність від сервісу є фактором, який фахівці з мікросервісу повинні брати до уваги при розробці заявок на масштабування. Використання CHPA може значно скоротити час холодного старту на етапі 1, прийняття рішення про масштабування.

Таблиця 4.5 – Загальний час, який Kubernetes витрачає на масштабування цільових послуг, з розподілом на блоки та етапи (секунди).

Мікросервіс	Назва вузла	Етап 1	Етап 2	Етап 3	Етап 4
Сервіс Замовлення	kmzg6	20.633	0.053	1.704	24.534
	ncch5	31.646	0.037	2.334	21.65
	gwlqt	20.47	0.015	1.444	26.054
Сервіс Польотів	7hdpg	25.941	0.018	3.355	22.032
	9jvrj	30.828	0.017	1.562	21.743
	jc4gg	36.789	0.014	1.179	22.374
Сервіс Користувачів	h8q5s	21.879	0.016	2.285	23.437
	hpcxb	33.891	0.024	1.397	33.231
	mxgws	34.674	0.023	1.511	23.439

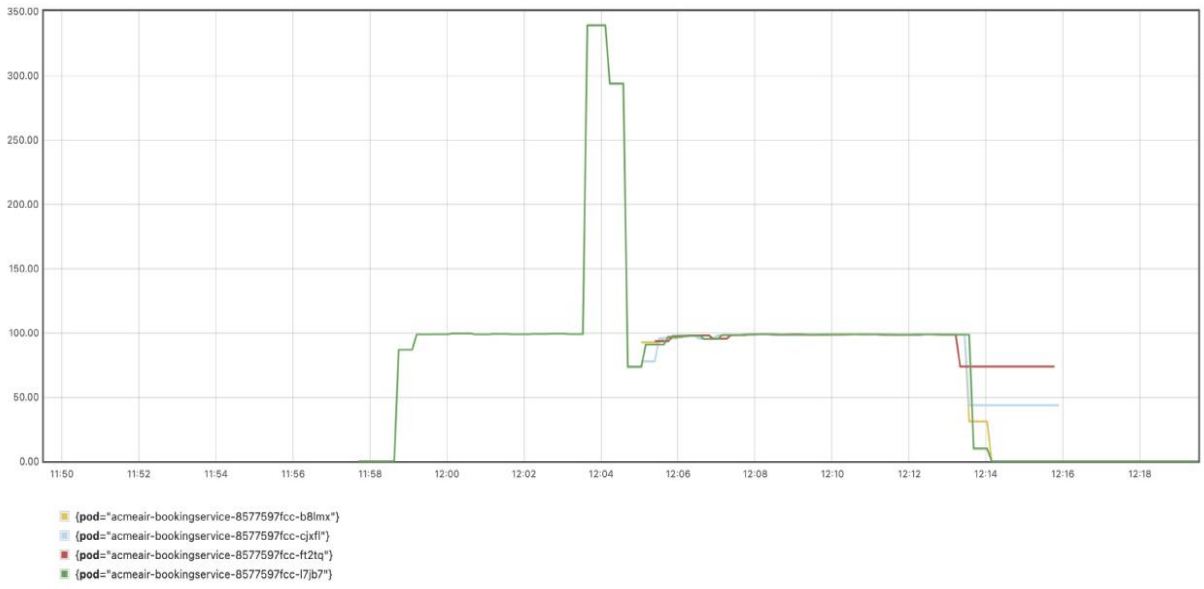


Рисунок 23 (а) – Навантаження на сервіс замовлень

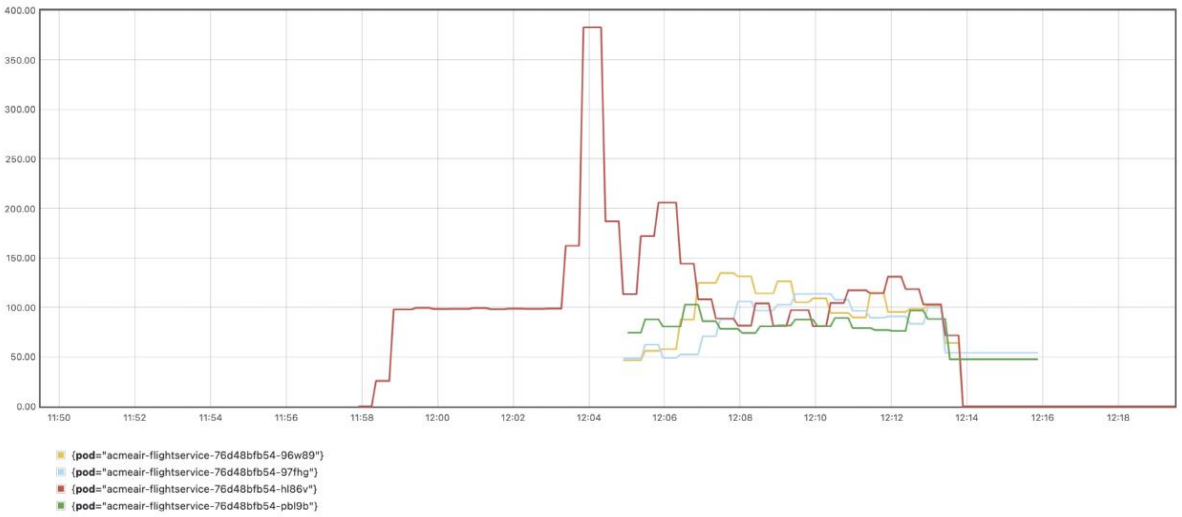


Рисунок 23 (б) – Навантаження на сервіс польотів

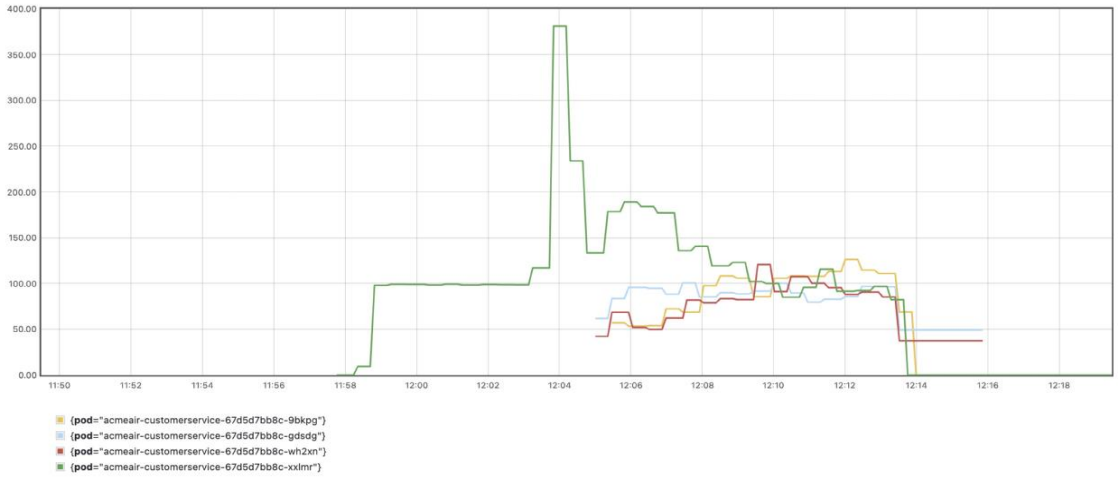


Рисунок 23 (в) – Навантаження на користувацький сервіс

4.4 Висновки

В цьому розділі було описано порівняння використання координованого горизонтального масштабувальника на відміну від звичайного горизонтального масштабувальника експерементальним шляхом. Результати дослідження показують, що гіпотетичні фактори, описані в 1 Розділу мають вплив на швидкодію і на час запуску програмного забезпечення. Використання CHPA ефективно вирішує проблему холодного старту, що дозволить зменшити витрати на ресурси сервера і краще опрацьовувати користувацький трафік.

ВИСНОВКИ

Kubernetes відкриває еру хмарних технологій. Вона приносить кілька нових ідей в розробку і розгортання мікросервісних додатків. Однією з переваг впровадження Kubernetes у комерцію є його здатність до горизонтального автоматичного масштабування поверх контейнерних робочих навантажень. Додатки, чутливі до затримок, очікують гнучкого масштабування.

У разі недостатньої кількості екземплярів контейнерів, коли до системи надходить велика кількість запитів, може відбутися процес автоматичного масштабування з холодного старту, що затримує реакцію та погіршує якість обслуговування користувачів. Щоб з'ясувати, які фактори впливають на холодний старт, ретельно було досліджено механізм та принцип роботи горизонтального автомасштабування. Отримані висновки демонструють, що для досягнення гнучкого автоматичного масштабування є можливості для вдосконалення. Оскільки все більше підприємств переносять додатки мікросервісів на кластери Kubernetes, представлене дослідження може дати розробникам мікросервісів, які цінують гнучкість, уявлення про те, як пом'якшити холодний старт.

За визначенням, "холодний старт" означає ситуацію, коли служба отримує величезну кількість запитів ззовні, але не має достатньої кількості внутрішніх екземплярів, щоб їх обслужити. Оскільки автоматичне масштабування запускається повільно, проблема холодного старту може збільшити затримку відповіді або навіть призвести до недоступності сервісу в більш серйозному випадку. При оцінці масштабування Служби польотів на основі HPA спостерігається, що найдовший час – 119,242 секунди – потрібен для підготовки та стабільної роботи під після першої хвилі трафіку з високим навантаженням (див. chfcn під в Таблиці 4.2). Навіть найшвидший вузл все ще потребує 32,915 секунди (див. pod xpz2f в Таблиці 4.2).

Щоб з'ясувати, які фактори найбільше впливають на затримку холодного старту при автоматичному масштабуванні мікросервісів на Kubernetes, проведено

ретельний огляд літератури, уважно розглянуто механізм автоматичного масштабування на основі HPA поетапно та оцінено п'ять гіпотетичних факторів.

Згідно з висновками, найбільші витрати часу припадають на два етапи: прийняття рішення про масштабування та запуск додатку. Час, витрачений на прийняття рішення про масштабування, варіюється від 12,499 до 60,302 секунд в експерименті 1. Через природу реактивних автомасштабувальників, HPA розтягує час при визначенні того, коли і як запускати масштабування, оскільки попит на трафік зростає. З одного боку, на продуктивність HPA впливає інтервал оцінки та період вибірки базової системи моніторингу, які за замовчуванням становлять 15 секунд та 1 хвилину. З іншого боку, в HPA відсутній механізм координації автомасштабування декількох сервісів. Кожен сервіс вимагає окремого процесу HPA та приймає незалежні, індивідуальні рішення щодо масштабування; таким чином, сервіси масштабуються несинхронно, хоча деякі сервіси демонструють взаємозв'язок висхідного та низхідного потоків і могли б масштабуватися синхронно.

Крім того, запуск додатку є ще одним трудомістким етапом після прийняття рішення про масштабування. Після того, як середовище виконання контейнера та мережа були встановлені та виділені для використання, додаток може завантажувати свій код, виконуватися та очікувати на обслуговування запитів, доки не пройде перевірку готовності кубелетом. Попередні експерименти в розділі 4.1 показують, що компонент Flight Service, написаний на Java та фреймворку Spring, виконується за час близько двадцяти секунд. Подальші дослідження показують, що вибір мови програмування може суттєво впливати на швидкість запуску додатку. Було порівняно час запуску програми "Hello, World!", реалізованої на шести мовах внутрішнього програмування, і виявили, що Java є найповільнішою, в той час як Go є перспективним вибором.

Отже, проблема холодного старту полягає у значній кількості часу, необхідного для визначення масштабних подій та завантаження коду. Серед гіпотетичних факторів вважається, що варто звернути увагу на залежність від сервісу та мову програмування. Розробники мікросервісів повинні розглянути

можливість використання висхідних та низхідних зв'язків для проведення скоординованого автоматичного масштабування. Крім того, оскільки мікросервісний стиль розробки робить можливим гетерогенну архітектуру програмного забезпечення, слід безперешкодно переходити на стек хмарних мов програмування, таких як Go, для швидкого запуску додатків. Крім того, необхідно подбати про те, щоб знайти "золоту середину" для розподілу ресурсів та обмеження робочих навантажень; в іншому випадку може статися перезапуск модулів, що ще більше затримає готовність модулів.

Крім того, запропоновано скоординований горизонтальний автомасштабувальник під назвою CHPA. CHPA є доповненням до HPA, що дозволяє групі ідентифікованих, корельованих цілей масштабування масштабуватися синхронно, замість того, щоб кожна з них приймала незалежне рішення про масштабування. В результаті, перше робоче навантаження, яке прийме правильне і бажане рішення, залучить своїх однорангових колег до спільного масштабування, навіть якщо ці однорангові робочі навантаження ще не досягли піку трафіку.

Щоб порівняти запропонований автомасштабувальник, CHPA, з HPA, проведено тести навантаження на API бронювання рейсів додатку Acme Air в розділі 4.1.2 та розділі 4.3, і дозволили двом автомасштабувальникам масштабувати робочі навантаження, щоб відповідати встановленому метричному показнику якості обслуговування (QPS). Результати автоматичного масштабування на основі HPA в розділі 4.1.2 показують, що три сервіси не масштабувалися з фіксованими кроками. Для останнього пакету mxgws знадобилося три послідовних раунди оцінювання, щоб прийняти рішення про масштабування (див. Таблицю 4.3). Хоча Служба польотів успішно масштабувалася до потрібної кількості модулів вже через 30 секунд, два відстаючих модулі від Служби бронювання та Служби обслуговування клієнтів були розтягнуті через їх неузгодженість поглядів на метрики QPS.

Навпаки, CHPA дозволяє координувати процес автоматичного масштабування корельованих навантажень. Результати повторного експерименту

показують, що за допомогою CHPA три сервіси масштабувалися синхронно. Для цього знадобилося лише два послідовних раунди оцінювання для всіх модулів.

Хоча точний час початку масштабування залежить від метрик, отриманих системою моніторингу, спостерігається, що відстаючі блоки зникають, демонструючи, що CHPA отримує корельовані робочі навантаження з висхідними та низхідними зв'язками швидше, ніж HPA, тим самим комплексно пом'якшуючи проблему "холодного старту".

Майбутні дослідження

Представлене дослідження розглядає декілька факторів, що сприяють холодному старту при автомасштабуванні, та пропонує можливі рішення для розробників мікросервісів. Однак, деякі обмеження все ще існують в цій тезі. Тому вказується на три перспективні напрямки для подальшої роботи.

Протягом усього дослідження базою виступав зразок мікросервісного додатку, Acme Air, для тестування та оцінки. Хоча Acme Air відображає багато важливих особливостей архітектури мікросервісів, насправді він не є готовим до виробництва додатком. Для подальшої роботи будуть застосовувати інші нові знання в реальних мікросервісних додатках для вивчення кожного конкретного випадку.

Реактивні автоскалери, включаючи HPA і запропонований в дослідженні CHPA, за своєю природою є нечутливими до сплесків. Не була розглянута можливість проектування прогнозованого горизонтального автомасштабувальника з урахуванням перевантажень, який відсутній в існуючих пропозиціях автомасштабувальників. Очікується, що предиктивний автомасштабувальник надасть обчислювальні ресурси перед піком трафіку. Однак поки не вистачає даних реального трафіку, тому це питання залишається для подальшої роботи.

Перелік джерел посиланнь:

1. Cunningham Ward. The wycash portfolio management system. ACMSIGPLANOOPS Messenger, 4(2):29–30, 1992.

2. Li Zengyang, Avgeriou Paris, Liang Peng. A systematic mapping study on technical debt and its management. Journal of Systems and Software, 101:193–220, 2015.

3. Buschmann Frank. To pay or not to pay technical debt. IEEE software, 28(6):29–31, 2011.

4. Fowler Martin. Refactoring: improving the design of existing code. Addison-Wesley Professional, 2018. Rattan Dhavleesh, Bhati

5. M. Abdullah, W. Iqbal, J. L. Berral, J. Polo, and D. Carrera. Прогнозне автомасштабування з урахуванням пакетів для контейнерних мікросервісів. IEEE Transactionson Services Computing, 2020

6. A. Bernstein and E. Newcomer. Principles of transaction processing. Morgan Kaufmann, July 2009

7. . W.Yi and L. Ozdamar, Cold Start problem optimization, vol. 179, no. 3, pp.. 1177–1193, 2007.

8. KEDA. Документація keda. https://keda.sh/docs/2.3/concepts/#architecture, травень 2021 р.

9. B.Bershad, R. P. Draves, and A. Forin. Using microbenchmarks to evaluate system performance. In [1992] Proceedings Third Workshop on Workstation Operating Systems, pages 148–153. IEEE, 1992.

10. Thakur and K. A.-M. Donnelly, Оркестрація мікросервісної архітектури, J. Food Eng., vol. 99, no. 2, pp. 98–105, 2010

11. B. Josh Rosso, Rich Lander and J. Harris. Кубернейт в комерції. O'Reilly Media, March 2021.

12. A. Khan and K. Salah, IoT security: Review, blockchain solutions, and open challenges, Future Gener. Comput. Syst., vol. 82, pp. 395–411, May 2018.

13. J. Carnell. Spring microservices in action. Manning Publications Company, July 2017.

14. D. Mao, Z. Hao, F. Wang, and H. Li, State-of-the-practice Autoscalers: A case study in Shandong Province, China, Sustainability, vol. 10, no. 2, p. 3149, 2018.

15. H. Hasan and K. Salah, Autoscalers in action, IEEE Access, vol. 6, no. 1, pp. 46781–46793, Dec. 2018.

16. Amir Shevat. Designing Bots: Creating Conversational Experiences [Online] / A. Shevat // O'Reilly Media. – Available: https://www.oreilly.com/library/view/designing-bots/9781491974810/

17. Playing with Java Microservices on Kubernetes and OpenShift - Nebrass Lamouchi (Leanpub account or valid email requested)

18. Software Design Using Kubernetes - Br. David Carlson, Br. Isidore Minerd

19. Linux Shell Scripting Tutorial - A Beginner's Handbook (2002) - Vivek G. Gite (HTML)

20. Introduction to the Command Line - Launch School (HTML)

21. From Containers to Kubernetes with Node.js (https://www.digitalocean.com/community/books/from-containers-to-kubernetes-with-node-js-ebook) - Kathleen Juell (PDF, EPUB)

22. Saini Vaibhav, Farmahinifarahani Farima, Lu Yadong, Baldi Pierre, VLopes Cristina. Oreo: Detection of clones in the twilight zone. In Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, pages 354–365, 2018.

23. Göde Nils, Koschke Rainer. Incremental clone detection. In 2009 13th European Conference on Software Maintenance and Reengineering, pages 219–228. IEEE, 2009.

24. Nguyen Tung Thanh, Nguyen Hoan Anh, Al-Kofahi Jafar M, Pham Nam H, Nguyen Tien N. Scalable and incremental clone detection for evolving software. In 2009IEEEInternationalConferenceonSoftwareMaintenance, pages491–494.IEEE, 2009.

25. Higo Yoshiki, Yasushi Ueda, Nishino Minoru, Kusumoto Shinji. Incremental code clone detection: A pdg-based approach. In 2011 18th Working Conference on Reverse Engineering, pages 3–12. IEEE, 2011.

26. Ragkhitwetsagul Chaiyong, Krinke Jens. Siamese: scalable and incremental code clone search via multiple code representations. Empirical Software Engineering, pages 1–49, 2019.

27. Krinke Jens. Identifying similar code with program dependence graphs. In Proceedings Eighth Working Conferenceon Reverse Engineering, pages301–309.IEEE,2001.

28. Komondoor Raghavan, Horwitz Susan. Using slicing to identify duplication in source code. In International static analysis symposium, pages 40–56. Springer, 2001.

29. Liu Chao, Chen Chen, Han Jiawei, Yu Philip S. Gplag: detection of software plagiarism by program dependence graph analysis. In Proceedings of the 12th ACM SIGKDD international conference on Knowledge discovery and data mining, pages 872–881. ACM, 2006.

30. Marcus Andrian, Maletic Jonathan I. Identification of high-level concept clones in source code. In Proceedings 16th Annual International Conference on Automated Software Engineering (ASE 2001), pages 107–114. IEEE, 2001.

31. Kamiya Toshihiro, Kusumoto Shinji, Inoue Katsuro. Ccfinder: a multilinguistic token-based code clone detection system for large scale source code. IEEE Transactions on Software Engineering, 28(7):654–670, 2002.

32. Baker Brenda S. On finding duplication and near-duplication in large software systems. In Proceedings of 2nd Working Conference on Reverse Engineering, pages 86–95. IEEE, 1995.

33. Jiang L., Misherghi G., Su Z., Glondu S. «DECKARD : Scalable and accurate tree-based detection of code clones,» в Proceedings of the 29th International Conference on Software Engineering, 2007.

34. Sargsyan S., Kurmangaleev S., Belevantsev A., Aslanyan H., Baloian A. «Scalable tool for code clone detection based on semantic analysis of program,» Trudy. ISP RAS, т. 1, pp. 39-50, 2015.

35. Schulman A. «Finding binary clones with opstrings function digests: Part III» Dr. Dobb’s Journal, 2005.

36. X. Zhou, X. Peng, T. Xie, J. Sun, C. Xu, C. Ji, and W. Zhao. Poster: Benchmarking microservice systems for software engineering research. In 2018 IEEE/ACM 40th International Conference on Software Engineering: Companion (ICSE-Companion), pages 323–324. IEEE, 2018.

37. Y. L. Scott Zhou. Go faaster: Cold start optimization in a serverless platform. KubeCon China, July 2019.

38. S. Qi, S. G. Kulkarni, and K. Ramakrishnan. Assessing container network interface plugins: Functionality, performance, and scalability. IEEE Transactions on Network and Service Management, 2020.

39. J. Nickoloff. Docker in action. Manning Publications Co., 2016.

40. I. Nadareishvili, R. Mitra, M. McLarty, and M. Amundsen. Microservice architecture: aligning principles, practices, and culture. ” O’Reilly Media, Inc.”, 2016.

41. A. Mohan, H. Sane, K. Doshi, S. Edupuganti, N. Nayak, and V. Sukhomlinov. Agile cold starts for scalable serverless. In 11th {USENIX} Workshop on Hot Topics in Cloud Computing (HotCloud 19), 2019.

42. L. Mishra, T. Gupta, and A. Shree. Online teaching-learning in higher education during lockdown period of covid-19 pandemic. International Journal of Educational Research Open, 1:100012, 2020.

43. K. Matthias and S. P. Kane. Docker: Up & Running: Shipping Reliable Containers in Production. ” O’Reilly Media, Inc.”, 2015.

44. J. Manner, M. Endreß, T. Heckel, and G. Wirtz. Cold start influencing factors in function as a service. In 2018 IEEE/ACM International Conference on Utility and Cloud Computing Companion (UCC Companion), pages 181– 188. IEEE, 2018.

45. J. Lewis. Microservices: a definition of this new architectural term. https://martinfowler.com/articles/microservices.html, March 2014.

46. Kubernetes. Kubernetes documentation. https://kubernetes.io/, 2021.

47. Kubernetes. Don’t panic: Kubernetes and docker. https://kubernetes.io/blog/2020/12/02/dont-panic-kubernetes-and-docker/, December 2020.

48. A. B. Josh Rosso, Rich Lander and J. Harris. Production Kubernetes. O'Reilly Media, March 2021.

49. D. Jackson and G. Clynch. An investigation of the impact of language runtime on the performance and cost of serverless functions. In 2018 IEEE/ACM International Conference on Utility and Cloud Computing Companion (UCC Companion), сторінки 154–160. IEEE, 2018.

50. J. Chester. Knative in Action. " O'Reilly Media, Inc.", March 2021

51. P. A. Bernstein and E. Newcomer. Principles of transaction processing. Morgan Kaufmann, July 2009.

52. M. Abdullah, W. Iqbal, J. L. Berral, J. Polo, and D. Carrera. Burst-aware predictive autoscaling for containerized microservices. IEEE Transactions on Services Computing, 2020.

53. Sen, K., Kalasapur, S., Brutch, T., Gibbs, S.: Jalangi: a selective record-replay and dynamic analysis framework for Javascript. In: Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering, pp. 488–498. ACM 2013

54. Atzei N, Bartoletti M, Cimoli T. A survey of attacks on Ethereum smart contracts (SoK). In: Maffei M, Ryan M., eds. 6th Conf. on Principles of Security and Trust (POST). 10204 of LNCS. Springer; 2017; Uppsala, Sweden: 164-186.

55. R. Kamath, ''Food traceability on blockchain: Walmart's pork and mango pilots with IBM,'' J. Brit. Blockchain Assoc., vol. 1, no. 2, p. 3712, 2018.

56. N. Nizamuddin, H. Hasan, and K. Salah, ''IPFS-blockchain-based authenticity of online publications,'' in Proc. Int. Conf. Blockchain (ICBC) (Lecture Notes in Computer Science). Seattle, WA, USA: Springer, 2018.

57. Q. Zhu and P. Zhou, The System Architecture for the Basic Information of Science and Technology Experts Based on Distributed Storage and Web Mining, pp. 527-530, 2012.

58. Sen, K., Kalasapur, S., Brutch, T., Gibbs, S.: Jalangi: a selective record-replay and dynamic analysis framework for Javascript. In: Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering, pp. 488–498. ACM 2013

59. S.M. Nabavinejad and M. Goudarzi, "Faster MapReduce Computation on Clouds through Better Performance Estimation", IEEE Transactions on Cloud Computing, no. 99, pp. 1-1, 2017.

ДОДАТОК А
(Обов’язковий)

КОПІЇ НАУКОВИХ ПУБЛІКАЦІЙ

ISSN 2307-5732
DOI 10.31891/2307-5732

НАУКОВИЙ ЖУРНАЛ

5.2022

ВІСНИК

Хмельницького національного університету

Технічні науки

Technical sciences

SCIENTIFIC JOURNAL
HERALD OF KHMELNYTSKYI NATIONAL UNIVERSITY

2022, Issue 5, Volume 313

Хмельницький

ВІСНИК
ХМЕЛЬНИЦЬКОГО НАЦІОНАЛЬНОГО УНІВЕРСИТЕТУ
серія: Технічні науки

Затверджений як фахове видання категорії «Б»,
РІШЕННЯ АТЕСТАЦІЙНОЇ КОЛЕГІЇ № 1643 ВІД 28.12.2019 та №409 від 17.03.2020

Засновано в липні 1997 р. ***Виходить 6 разів на рік***

Хмельницький, 2022, № 5(313)

Засновник і видавець: Хмельницький національний університет
(до 2005 р. – Технологічний університет Поділля, м. Хмельницький)

Наукова бібліотека України ім. В.І. Вернадського http://nbuv.gov.ua/j-tit/Vchnu_tekh

Включено до науково-метричних баз:

Google Scholar	http://scholar.google.com.ua/citations?hl=uk&user=aIUP9OYAAAAJ
Index Copernicus	http://jml2012.indexcopernicus.com/passport.php?id=4538&id lang=3
Polish Scholarly Bibliography	https://pbn.nauka.gov.pl/journals/46221
CrossRef	http://doi.org/10.31891/2307-5732

Технічний редактор Горященко К. Л., к.т.н.
Редактор-коректор Броженко В. О.

Рекомендовано до друку рішенням вченої ради Хмельницького національного університету, протокол № 3 від 27.10.2022 р.

Адреса редакції: редакція журналу "Вісник Хмельницького національного університету"
Хмельницький національний університет
вул. Інститутська, 11, м. Хмельницький, Україна, 29016

☎ (038-2) 67-51-08 **web:** http://journals.khnu.km.ua/vestnik
http://lib.khnu.km.ua/visnyk_tup.htm

e-mail: visnyk.khnu@khmnu.edu.ua

Зареєстровано Міністерством України у справах преси та інформації.
Свідоцтво про державну реєстрацію друкованого засобу масової інформації
Серія КВ № 24922-14862ПР від 12 липня 2021 року

ЗМІСТ

DOI 10.31891/2307-5732-2022-313-5-260-264
УДК 004.056.5:621

МАЄВСЬКИЙ Ярослав
Хмельницький національний університет
ORCID ID: 0000-0002-5732-7093
e-mail: yarchik.mayevskij@gmail.com
ПРАВОРСЬКА Наталія
Хмельницький національний університет
ORCID ID: 0000-0001-6001-3311
e-mail: margana2000007@gmail.com

ПІДВИЩЕНЯ ЕФЕКТИВНОСТІ АВТОМАТИЗАЦІЇ МАСШТАБУВАННЯ МІКРОСЕРВІСІВ У СИСТЕМІ КЕРУВАННЯ КОНТЕЙНЕРИЗОВАНИМИ ЗАСТОСУНКАМИ KUBERNETES

Автоматичне масштабування контейнерів в системі Kubernetes відіграє важливу роль у опрацюванні вхідних запитів користувачів веб-застосунків. В цій статті проаналізовано етапи масштабування контейнерів, їхня ініціалізація і подальший запуск програмного забезпечення. Для досягнення низької затримки запитів користувача у випадку динамічного навантаження проводився аналіз процесу автоматичного масштабування контейнерів та факторів, які впливатимуть на процес масштабування. Маючи повне розуміння процесів та механізмів, за правилами яких відбувається масштабування, стало можливим розробка методу та стратегії для прибирання перепон, що сповільнюють сам процес автомасштабування і при цьому збереження необхідних властивостей від існуючого масштабування. Пришвидшення масштабування контейнерів, яке напряму буде впливати на швидкість веб-сервісів стає можливим саме через позбавлення затримки в автоматичному масштабуванні контейнерів.

З отриманих результатів дослідження сформовано метод оптимізації автоматичного масштабування контейнеризованих застосунків за рахунок позбавлення затримки під час холодного старту. Така затримка проявляється у випадку автомасштабуванням мікросервіса, де Kubernetes, як очікується, горизонтально масштабує контейнери шляхом створення додаткових реплік до необхідної кількості, щоб опрацювати необхідний трафік ззовні. Затримка, спричинена автомасштабувальником, впливає на час опрацювання запитів користувача веб-сервісу.

Ключові слова: мікросервіси, контейнери, автомасштабувальник, автоматичне масштабування контейнерів, Kubernetes, затримка, холодний старт.

MAYEVSKIJ Yaroslav, PRAVORSKA Natalya
Khmelnitskyi national university

IMPROVING THE EFFICIENCY OF AUTOMATION THE SCALING OF MICROSERVICES IN THE KUBERNETES CONTAINERIZED APPLICATION MANAGEMENT SYSTEM

Automatic container scaling in Kubernetes plays an important role in handling incoming requests from web application users. This article analyzes the stages of container scaling, their initialization and subsequent software launch. In order to achieve low latency of user requests in the case of dynamic load, the analysis of the process of automatic scaling of containers and the factors that will affect the scaling process was carried out. Having a full understanding of the processes and mechanisms by which scaling takes place, it became possible to develop a method and strategy for cleaning obstacles that slow down the autoscaling process itself and at the same time preserve the necessary properties of the existing scaling. Acceleration of scaling of containers, which will directly affect the speed of web services, becomes possible precisely because of the elimination of the delay in automatic scaling of containers.

The work considered scaling optimization using not only container pre-creation networks, but also the use of container images, which enable the sharing of linked libraries in memory and the extension of Kubernetes' declarative configuration management approach for parallel creation of multiple container instances.

Based on the obtained research results, a method for optimizing the automatic scaling of containerized applications by eliminating the delay during a cold start has been developed. This latency manifests itself in the case of microservice autoscaling, where Kubernetes is expected to scale containers horizontally by creating additional replicas to the required number to handle the required traffic from the outside. The delay caused by the autoscaler affects the processing time of the user's web service requests.

Keywords: microservices, containers, autoscaler, autoscaling of containers, Kubernetes, latency, cold start.

Постановка проблеми у загальному вигляді та її зв'язок із важливими науковими чи практичними завданнями

У контексті Kubernetes холодний старт – це затримка у процесі масштабування контейнеризованих застосунків, яка спричинена ініціалізацією контейнера з необхідним сервісом. Така затримка проявляється у випадку автомасштабування мікросервіса, де Kubernetes, як очікується, горизонтально масштабує контейнеризовану програму шляхом створення додаткових реплік потрібного контейнера до необхідної кількості, щоб опрацювати необхідний розмір трафіка ззовні.

Затримка автоматичного масштабування визначається багатьма факторами на різних етапах. В Kubernetes процес масштабування складається з кілька взаємопов'язаних фаз. Починається процес з того, що платформа визначає пік трафіку а також наступне планування набіру контейнерів аж до ініціалізації контейнера і виконання коду програмного забезпечення, після чого ми можемо констатувати успішну відповідь на запит

Щоб досягти низької затримки запитів користувача у випадку динамічного навантаження потрібно проаналізувати процес автоматичного масштабування контейнерів і визначити ключові фактори, які впливають на цей процес. Під час проведення дослідження детально вивчається проблема холодного запуску і аналізується процес масштабування. Тільки з повним розумінням процесів і механізмів масштабування є змога запропонувати метод або стратегії для усунення проблем, які сповільнюють процес автоматичного масштабування і в той же момент залишивши всі потрібні властивості від існуючого масштабування.

Позбавлення затримки в автоматичному масштабуванні контейнерів дозволяє пришвидшити масштабування контейнерів, що на пряму впливає на швидкодію веб-сервісів.

Аналіз останніх джерел

Дослідження засноване на офіційній документації платформи Kubernetes, яке детально описує етапи масштабування контейнеризованих застосунків а також описує в деталях горизонтальний і вертикальний автомасштабувальники. Для порівняння звичайних горизонтальних і вертикальних масштабувальників з вбудованим фреймворком від Kubernetes було проаналізована офіційна документація KNtive. Також в роботі авторами було розглянуто оптимізацію масштабування за рахунок використання нейронних мереж для попереднього створення контейнерів, використання образів контейнерів, які дозволяють спільно використовувати зв'язані бібліотеки в пам'яті і розширення декларативного підходу керування конфігурацією Kubernetes для паралельного створення кількох екземплярів контейнера. Ця проблема має різні варіанти и в різних контекстах і умовах. Наприклад, у безсерверній парадигмі проблема холодного запуску чітко визначається як час, витрачений на підготовку середовища виконання коду під час першого виклику функції. Порівняно з мікросервісами, безсерверна архітектура є більш модульним архітектурним стилем і має поставлені задачі щодо економії ресурсів, коли немає активного трафіку користувачів.

Виклад основного матеріалу

Холодний запуск у автоматичному масштабуванні контейнеризованих застосунків – це подія, яка описує момент значного збільшення навантаження на веб-застосунок, але в противагу програмному забезпеченні не достатньо ресурсів для опрацювання цих запитів. Тому, платформа керування контейнеризованими застосунками реагує на цю подію і розпочинає процес запуску додаткових контейнерів, які будуть опрацьовувати новий трафік. На цей процес витрачається багато часу, і якщо інші контейнери не звільнять свої ресурси – користувач буде очікувати час запуску додаткових контейнерів. Як результат – запити користувача довго опрацьовуються, або користувач взагалі не отримує відповіді на свої запити.

Процес холодного старту зображений на рисунку 1.

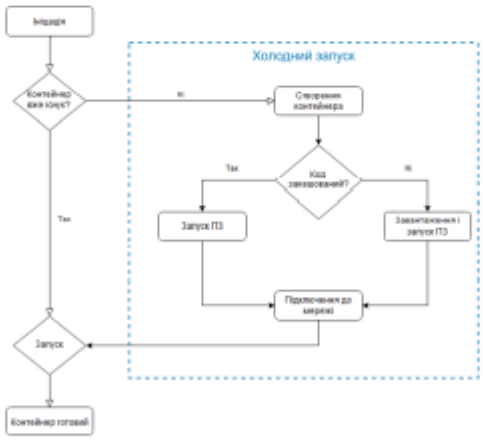


Рис. 1. Централізована система управління

Зазвичай, цей процес починається з того, що платформа перевіряє наявність середовища для запуску програмного забезпечення. Якщо таке існує – виклик буде опрацьований існуючими екземплярами – такий процес називається теплим стартом. В іншому випадку розпочинається процес холодного старту, де немає підготовлених контейнерів для обробки запита. Відповідно починається процес запуску контейнера, для подальшої роботи потребують код програмного забезпечення, який може бути закешований локально або завантажений. Після чого відбувається підключення мережі.

Загалом, для проходження усіх етапів холодного запуску може знадобитися кілька секунд – в залежності від програмного забезпечення або конфігурації контейнерів. Створений контейнер використовується багаторазово для обслуговування майбутніх запитів.

Підводячи підсумок, можна сказати, що типовий холодний запуск передбачає три етапи створення: створення контейнера, завантаження коду програмного забезпечення та розгортання мережі. Створення контейнера та розгортання мережі сприяють секундній затримці холодного запуску, тоді як час завантаження функціонального коду залежить від способів кешування коду.

Контейнери запуску програмного забезпечення

Для початку, проводиться порівняння трьох доступних середовищ для запуску контейнерів, а саме: Docker (v19.03.15), container (v1.4.4), CRI-O (v1.20.3). Вони відрізняються реалізацією Kubernetes Container Runtime Interface (CRI).

Для Docker потрібен окремий процес dockershim, щоб підтримувати разом Docker і kubelet, а container містить ту саму функціональність dockershim, що й плагін CRI, у своєму коді. CRI-O реалізує CRI за дизайном і може безпосередньо взаємодіяти з kubelet без додаткових витрат. Щоб порівняти їх продуктивність, вимірюється час запуску контейнера під час виконання різних контейнерів. Час запуску контейнера — це тривалість від того моменту, як kubelet отримав контейнеру щойно створений контейнер, призначений йому.

На рисунку 2 показана діаграма, на якій CRI-O має найкращу продуктивність з точки зору затримки

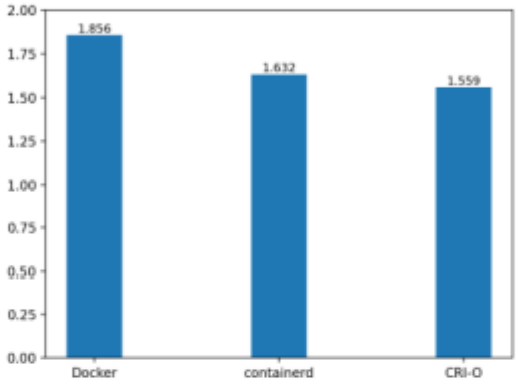


Рис. 2. Діаграма затраченого часу на запуск контейнера в різних умовах

Налаштування мережі

Плагіни мережевого інтерфейсу контейнера (CNI) використовуються в Kubernetes для підтримки роботи в кластерній мережі. CNI плагіни відповідають за підключення мережі до контейнера під час його ініціалізації. Такі плагіни відрізняються за собою механізмами реалізації, протоколами і мережевою моделлю. Порівняємо 5 поширених рішень, а саме: Cilium, Flannel, Calico, Weave Net, Kube-router. Діаграма затраченого часу в секундах на запуск контейнера з різними плагінами CNI представлена на рисунку 3.

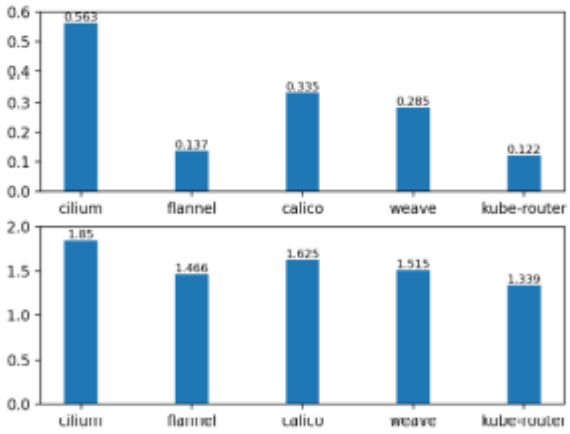


Рис. 3. Діаграма затраченого часу в секундах на запуск контейнера з різними плагінами CNI

Згідно з діаграмою, Flannel і kube-router витрачають найменше часу, тоді як Cillium відносно повільніший за інших. Однак, загальна різниця між плагінами не значна. У результаті можна зробити висновок, що різні CNI плагіни можуть збільшити швидкодію і зменшити час запуску контейнера. Але варто розуміти, що різні плагіни мають інші параметри як час на запуск, значною мірою вони відрізняються в пропускній здатності вводу-ввиводу запитів, а не часу на запуск.

Розмір кластера

Розмір кластера є однією з гіпотетичною проблем у швидкодії запуску контейнерів. Але, планувальник Kubernetes має механізм, який аналізує дані найменшого за розміром вузла в контейнері. Однак, незалежно від розміру кластера, планування модуля працює на шкалі часу в мілісекунди; таким чином, вплив розміру кластера на продуктивність масштабування модуля має бути незначним. На рисунку 4 представлена діаграма, яка відображує затрачений час на запуск контейнера з різними за розміром кластерами.

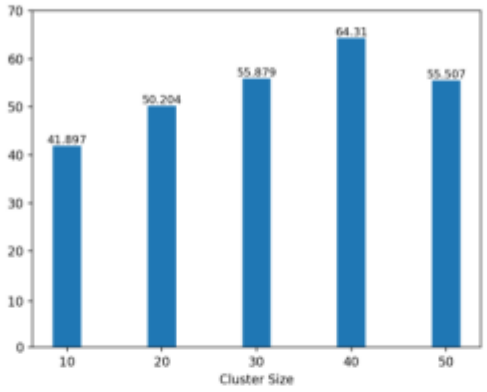


Рис. 4. Діаграма затраченого часу в мікросекундах на запуск контейнера з різними розмірами кластера

Мова програмування

Важливим фактором у запуску контейнерів є програмне забезпечення, а саме мова якою воно написане. Таким чином, ми порівнюємо час на запуск мінімального програмного забезпечення в умовах платформи Kubernete з контейнером Docker. Для прикладу, було реалізоване мінімальне програмне забезпечення на шістьох різних мовах, яке виводить в консоль інформацію, що воно запустилось. Вибрані мови: Go 1.16, Java 11.00, C# 3.1, JavaScript 14.17, PHP 7.3, Python 3.9. Результати запусків наведені на рисунку 5.

Результат дослідження показав, що мова програмування теж впливає на час запуску контейнерів.

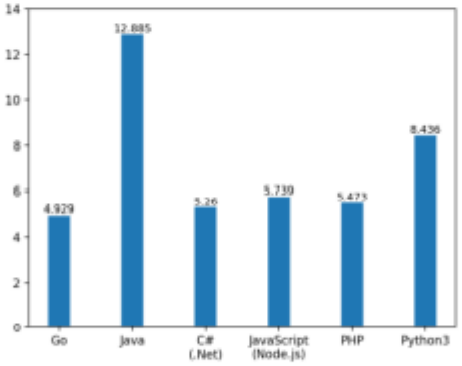


Рис. 5. Діаграма використаного часу для запуску мінімального ПЗ

З діаграми добре видно, що більший час було затрачено для запуску програмного забезпечення написаного мовою Java, а лідером швидкості з запуску ПЗ, стало написане мовою Go.

Висновки з даного дослідження і перспективи подальших розвідок у даному напрямі

Проведені в дослідженні експерименти надали деяку корисну інформацію, яка стане в пригоді при розробці методів, направлених на покращення автоматичного масштабування контейнерів. У проведеному дослідженні було проведено аналіз та порівняння різних факторів, що впливають на ефективність масштабування контейнеризованих застосунків в системі керування контейнеризованими застосунками. Було проаналізовано вплив інструментів контейнеризації на час запуску сервісів, в результаті чого, отримані дані, які вказують, що CRI-O є найшвидшим інструментом контейнеризації. Також детально проаналізовано вплив плагінів мережевого інтерфейсу контейнера, і доведено, що Flannel і kube-router витрачають найменше часу, тоді як Cillium, який використовується з коробки – відносно повільніший за інших. Однак, загальна різниця між плагінами не значна. Також корисною інформацією є те, що розмір кластера не має впливу на час запуску контейнера, а мови програмування значно впливають на час запуску програмного забезпечення в контейнерах. Всі отримані дані будуть використані в наступній розробці як самого методу,

так і стратегії для прибирання перепон, що сповільнюють сам процес автомасштабування і при цьому збереження необхідних властивостей від існуючого масштабування.

Література:

1. D. Mao, Z. Hao, F. Wang, and H. Li, State-of-the-practice Autoscalers: A case study in Shandong Province, China, Sustainability, vol. 10, no. 2, p. 3149, 2018.
2. B. Bershad, R. P. Draves, and A. Forin. Using microbenchmarks to evaluate system performance. In [1992] Proceedings Third Workshop on Workstation Operating Systems, pages 148–153. IEEE, 1992
3. J. Dieltjens, E. Heydari Beni, E. Truyen, B. Lagaisse – Reducing cold start during elastic scaling, KU Leuven, Belgium, 2019.
4. Kubernetes Documentation – [Електронний ресурс]. – Режим доступу: https://kubernetes.io/docs/home/

Надійшла/Paper received : 27.09.2022 р. Надрукована/Printed :01.11.2022 р.

ДОДАТОК Б
(Обов’язковий)

ПРЕЗЕНТАЦІЙНІ МАТЕРИАЛИ

Кафедра інженерії програмного забезпечення

Метод підвищення ефективності автоматизації масштабування мікросервісів у відкритій системі автоматичного розгортання і управління контейнеризованими застосунками

Виконав: *студент гр. ІПЗм-21-1* *Маєвський Я.Ю*

Керівник: *канд. пед. наук* *Праворська Н. І.*

Об'єкт, предмет, мета дослідження

Об'єкт дослідження. Відкрита система автоматичного розгортання, масштабування та управління застосунками у контейнерах «Kubernetes»

Предмет дослідження. Методи підвищення ефективності масштабування мікросервісів в «Kubernetes»

Мета роботи. Підвищення ефективності автоматизації масштабування мікросервісів в Kubernates

Задача роботи. Дослідження можливостей для оптимізації автоматичного масштабування, рішення проблеми холодного запуску контейнеризованих застосунків.

Актуальність проблеми

Актуальність теми роботи полягає в необхідності вирішення проблеми холодного старту під час автоматичного масштабування контейнеризованих застосунків а також виявлення факторів, які впливають на час запуску масштабування.

З моменту появи Kubernetes зявилось безліч алгоритмів масштабування, типів контейнерів, які підтримують ПЗ різної складності, написаних на різних мовах програмування різної архітектури, усі ці фактори впливають на швидкість взаємодії користувача з ПЗ.

Завдання дослідження

Основними завданнями роботи виступають:

1. Визначення факторів, які впливають на швидкодію масштабування контейнеризованих застосунків, а також створюють умови для проблеми холодного запуску.
2. Аналіз способів віртуалізації застосунків : Docker, Containerd, CRI-O, Kata Container
3. Дослідження і порівняння алгоритмів автомасштабувальників: HPA, VPA, KEDA, Knative. Представлення CHPA, який вирішує проблему холодного старту.
4. Дослідження способів для зменшення впливу знайдених факторів на автоматичне масштабування контейнерів
5. Створення середовища для тестування і реалізація сформованого рішення
6. Аналіз і оцінка ефективності дослідження

Наукова новизна

- Вирішення проблеми холодного старту контейнерів при автоматичному масштабуванні за рахунок удосконалення горизонтального контейнерного алгоритму масштабування, та усунення факторів середовища запуску контейнерів, що сприяють сповільненню швидкодії запуску ПЗ.

Практичне значення

- Практична цінність отриманих результатів полягає в успішній розробці підходів, методів та алгоритмів помякшення проблеми холодного запуску контейнеризованих застосунків при автоматичному масштабуванні.
- Завдяки результатам дослідження ми можемо оцінити вплив різних факторів на швидкодію автомасштабування, як результат - це дає можливість зменшити час запуску контейнерів, програмного забезпечення і середовища в платформі Kubernetes.

Що таке Kubernetes?

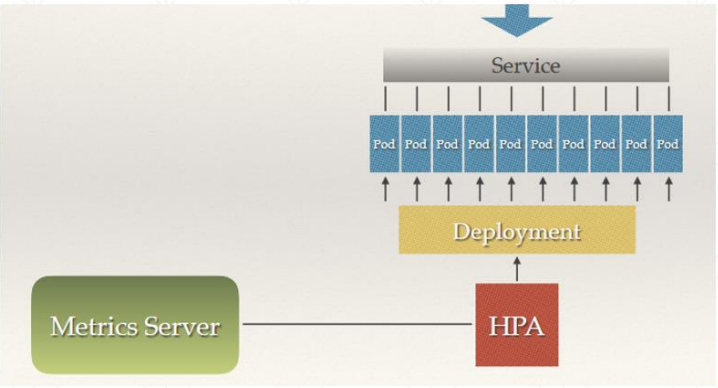
- Kubernetes - це платформа з відкритим вихідним кодом для управління контейнеризованими робочими навантаженнями та супутніми службами.
- Її основні характеристики - кросплатформенність, розширюваність, успішне використання декларативної конфігурації та автоматизації.
  - Виявлення сервісів та балансування навантаження
  - Оркестрація сховища інформації
  - Автоматичне розгортання та відкатування
  - Автоматичне розміщення задач
  - Самозцілення контейнерів
  - Управління секретами та конфігурацією

Що таке Kubernetes?

kubernetes

Автомасштабування

- Автоматичне масштабування - це метод, який використовується в хмарних обчисленнях, при якому кількість обчислювальних ресурсів у межах серверної ферми, яка зазвичай вимірюється за кількістю активних серверів, автоматично масштабується в залежності від навантаження на сервіс.



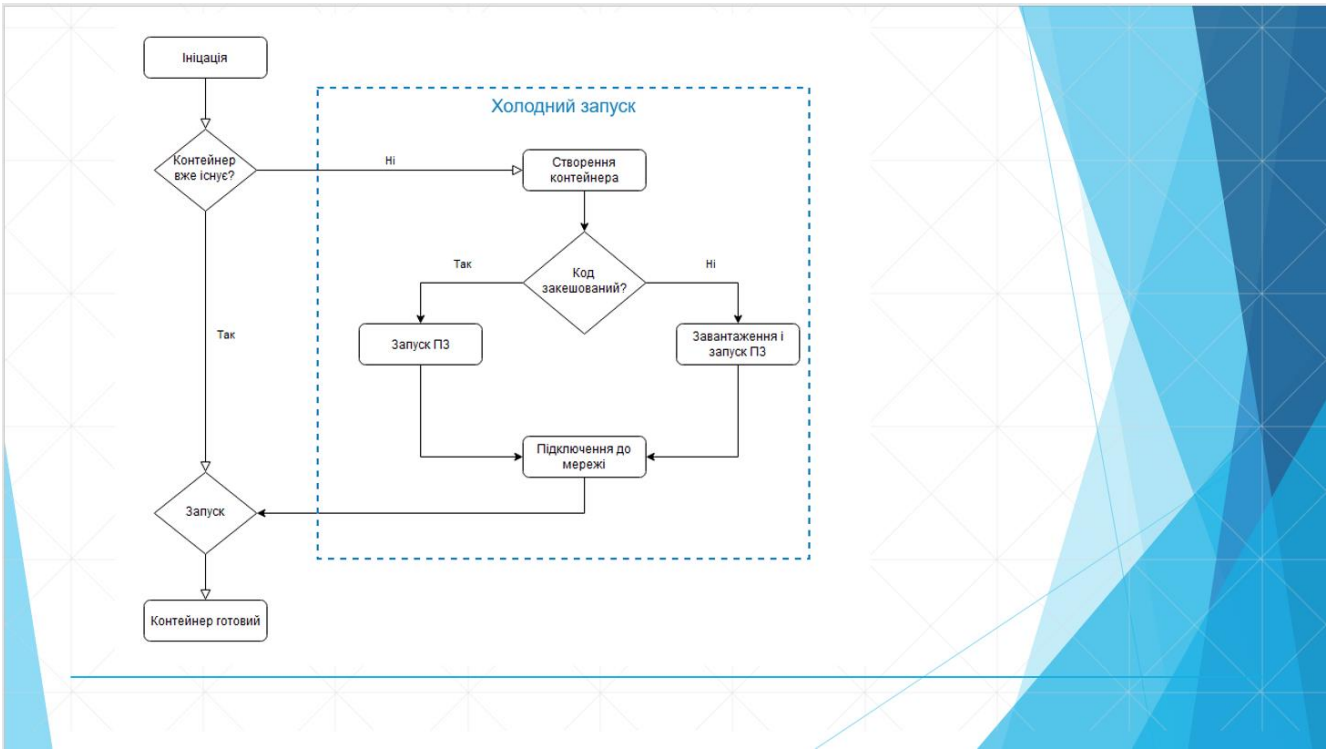
Алгоритми масштабування

- Horizontal Pode Austoscaler
- Vertical Pode Autoscaler
- KEDA - Kubernetes Event-Driven Autoscallor
- KPA – Knative Pode Autoscaler

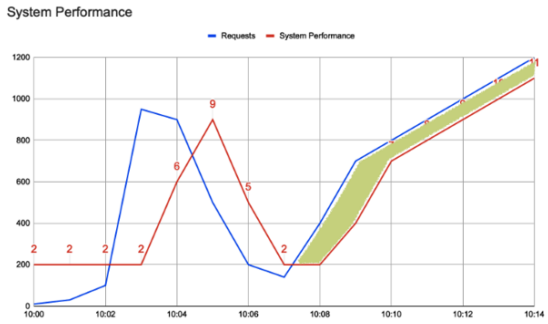
Проблема холодного запуску контейнерів

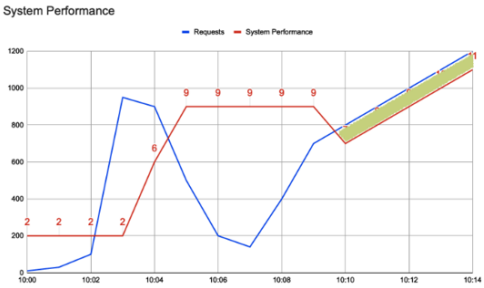
- Холодний запуск у автоматичному масштабуванні контейнеризованих застосунків – це подія, яка описує момент значного збільшення навантаження на сервіс, але в противагу програмному забезпеченні не достатньо ресурсів для опрацювання цих запитів. Тому, платформа розпочинає процес запуску додаткових контейнерів, які будуть опрацьовувати новий трафік.

- На цей процес витрачається багато часу, і якщо інші контейнери не звільнять свої ресурси – користувач буде очікувати час запуска додаткових контейнерів. Як результат – запити користувача довго опрацьовуються, або користувач взагалі не отримує відповіді на свої запити.



Постановка проблеми автомасштабування





- Швидкість масштабування
- Відсутня можливість стабілізації надлишкових ресурсів
- Важлива конфігурація заблокована

Оптимізація проблеми масштабування CHPA

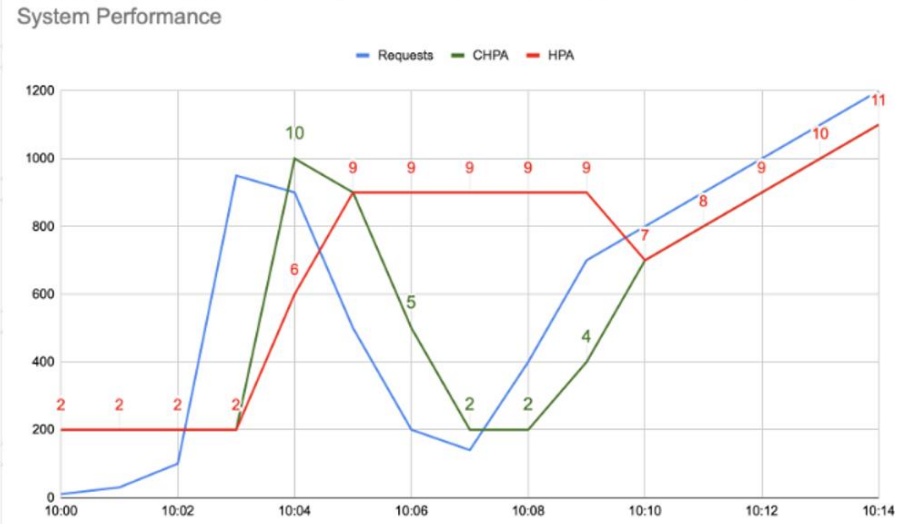
```yaml
apiVersion: chpa.my.domain/v1
kind: CHPA
metadata:
  name: chpa-sample
spec:
  # defines auto-scaling group

scaleTargets:
- scaleTargetRef:
    apiVersion: apps/v1
    kind: Deployment
    name: workload_a
  # metric name and desired value
  metric:
    name: qps_a
    targetValue: 1000
  # weight mapping in the data flow
  weight: 1
- scaleTargetRef:
    apiVersion: apps/v1
    kind: Deployment
    name: workload_b
  metric:
    name: qps_b
    targetValue: 1000
  weight: 1
status:
  lastScaleTime:
  scaleTargets:
  - scaleTargetRef:
      apiVersion: apps/v1
      kind: Deployment
      name: workload_a
      # replica count since last scale
      replicas: 2
  - scaleTargetRef:
      apiVersion: apps/v1
      kind: Deployment
      name: workload_b
      replicas: 2
```

$$ReplicaProposal_i = ReadyPodCount_i * \frac{TotalMetricValue_i}{TargetMetricValue_i}$$

$$NewReplica_j = \max_{i=1,\ldots,n} ReplicaProposal_i * \frac{Weight_j}{Weight_i}$$

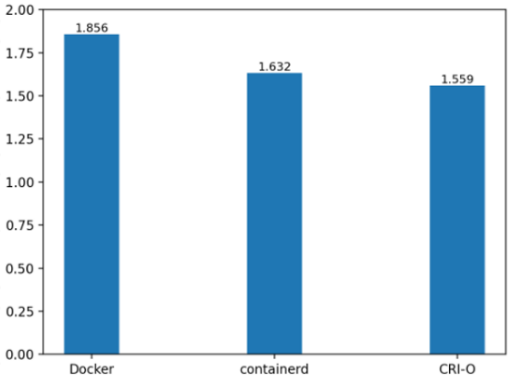
- Kubernetes
- Docker (container runtime)
- Cilium CNI
- EFK stack
- Prometheus

CHPA



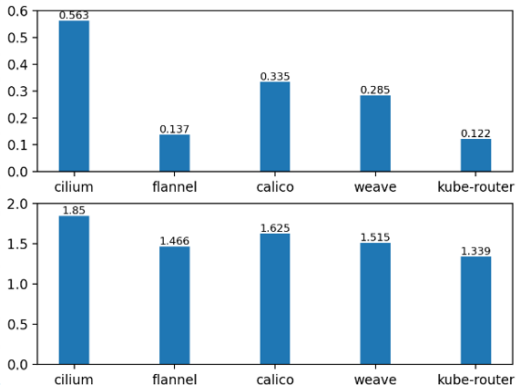
Контейнери запуску ПЗ

- Ми порівнюємо три доступні середовища виконання контейнерів: container (v1.4.4), (v1.20.3). Вони відрізняються реалізацією Kubernetes Container Runtime Interface (CRI).
- Docker (v19.03.15),
- CRI-O
- Containerd
- Podman



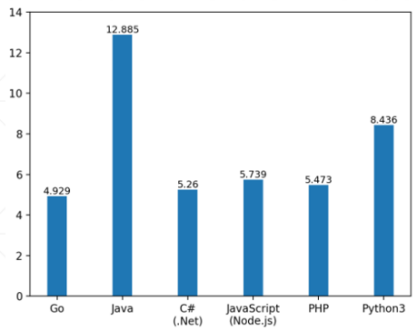
Налаштування мережевого інтерфейсу CNI

- CNI плагіни відповідають за підключення мережі до контейнера під час його ініціалізації. Такі плагіни відрізняються за собою механізмами реалізації, протоколами і мережевою моделлю. Порівняємо 5 поширених рішень, а саме : Cilium, Flannel, Calico, Weave Net, Kube-router.

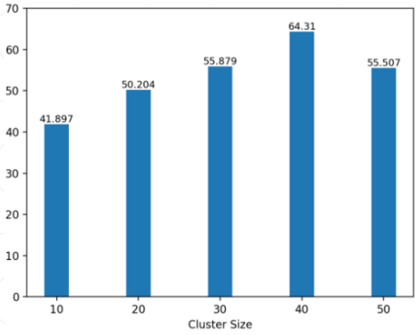


Фактори, які впливають на час запуску

- Мова Реалізації ПЗ



- Розмір кластера контейнера



Наукові публікації

1. Маєвський Я. І., Праворська Н.І. ПІДВИЩЕНЯ ЕФЕКТИВНОСТІ АВТОМАТИЗАЦІЇ МАСШТАБУВАННЯ МІКРОСЕРВІСІВ У СИСТЕМІ КЕРУВАННЯ КОНТЕЙНЕРИЗОВАНИМИ ЗАСТОСУНКАМИ KUBERNETES - Вісник ХНУ, серія Технічні науки, N 4 , 2022 (подано до друку)

Висновки та рекомендації

В результаті виконання дипломної роботи було проведено аналіз предметної області, а також проаналізовано фактори, які впливають на швидкодію масштабування контейнерів.

Запропоновано метод та рішення, які дозволять покращити роботу горизонтального автоматичного масштабувальника а також пом'якшити холодний старт контейнерів в Kubernetes.

Завідувачу кафедри інженерії програмного
забезпечення проф. Бедратюку Л. П.
здобувача вищої освіти
Масвського Я.Ю.
факультет ІТ, 2 курс, група ІПЗм-21-1

ЗАЯВА

З правилами чинного Положення «Про дотримання академічної доброчесності в Хмельницькому національному університеті» від 26.09.2020 (зі змінами від 26.11.2020), згідно з яким виявлення плагіату є підставою для відмови в допуску кваліфікаційної роботи до захисту та застосування заходів дисциплінарної та академічної відповідальності, ознайомлений(а). Про використання програмно-технічних засобів для перевірки кваліфікаційних робіт здобувачів вищої освіти на плагіат оповіщений(а) та надаю свою згоду на обробку та збереження університетом моєї роботи в інституційному репозитарії університету.

Також надаю університету право на передачу моєї роботи для обробки та збереження в базах даних програмно-технічних засобів (Unicheck та Anti-Plagiarism) та використання роботи для виявлення плагіату в інших роботах, які перевіряються програмно-технічними засобами та користувачами, що мають доступ до цих програмно-технічних засобів, виключно в обмежених цілях для виявлення плагіату в текстах робіт.

Робота для перевірки університетом надається в друкованому та електронному варіанті. Електронна версія моєї роботи збігається (ідентична) з друкованою.

23.11.2022 р.
дата


підпис

UNICHECK

Ім'я користувача:
Кафедра ІПЗ

ID перевірки:
1013143894

Дата перевірки:
02.12.2022 08:57:46 EET

Тип перевірки:
Doc vs Internet + Library

Дата звіту:
02.12.2022 09:27:43 EET

ID користувача:
100005589

Назва документа: **Маєвський_Ярослав_ІПЗ_21_м_без дод**

Кількість сторінок: **100** Кількість слів: **17883** Кількість символів: **137827** Розмір файлу: **1.98 MB** ID файлу: **1012911671**

7.73%
Схожість

Найбільша схожість: **4.6%** з джерелом з Бібліотеки (ID файлу: **1009344152**)

3.01% Джерела з Інтернету	214	Сторінка 102
7.13% Джерела з Бібліотеки	135	Сторінка 104

0% Цитат

Вилучення цитат вимкнене

Вилучення списку бібліографічних посилань вимкнене

0%
Вилучень

Немає вилучених джерел

Mon Dec 05 11:09:33 EET 2022, Хіврич Володимир Русланович, Хмельницький національний університет, ХНУ

Anti-Plagiarism v-15.257

Максимальне співпадіння з одним документом 4.0%

Словники перевірки: en_US, ru_RU, ua_UA. **Помилок в документах: 14%**

ID: 108879 Назва: КРМ на тему: Метод підвищення ефективності автоматизації масштабування мікросервісів у відкритій системі автоматичного розгортання і управління контейнеризованими застосунками Додано в БД: 2022-12-02 Автора: Маєвський Я.Ю. Керівники: Праворська Н.І. Консультанти: Опоненти:	Документ		Сумарний збіг по Базі Даних	
	Символи	Лексеми	Символи	Лексеми
	125241	1066	7263 (6%)	97 (9%)

Джерело плагіату

ID	Опис	Наявність плагіату в документі	
		Символи	Лексеми

РІШЕННЯ ЕКСПЕРНОЇ КОМІСІЇ
КАФЕДРИ ІНЖЕНЕРІЇ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ
ПРО ДОПУСК КВАЛІФІКАЦІЙНОЇ РОБОТИ ДО ЗАХИСТУ

Підтверджуємо ознайомлення з результатом звіту подібності щодо роботи, генерованого системою виявлення текстових збігів/ідентичності/схожості:

Назва: «Метод підвищення ефективності автоматизації масштабування мікросервісів у відкритій системі автоматичного розгортання і управління контейнеризованими застосунками»
Автор: Маєвський Ярослав Юрійович
Спеціальність: 121 – Інженерія програмного забезпечення
Освітня програма: Освітньо-професійна програма «Інженерія програмного забезпечення»
Науковий керівник: Праворська Наталія Іванівна, кандидат педагогічних наук, доцент
Після аналізу звіту подібності зроблено такий висновок:

№	Висновок	Позначка про відповідність
1	Запозичення, виявлені в роботі, є законними і не є плагіатом. Робота приймається до захисту.	**відповідає**
2	Виявлені запозичення не є плагіатом, розміщені в розділах, які не описують безпосередньо авторське дослідження, але кількість цитат перевищує обсяг, виправданий поставленою метою роботи. Робота приймається до захисту, але має бути відкоригована. Відкоригований варіант має бути поданий на кафедру за 2 дні до захисту, разом із заявою щодо самостійності виконання письмової роботи та ідентичності друкованої та електронної версії роботи	
3	Виявлені запозичення не є плагіатом, але частково розміщені в розділах, які описують безпосередньо авторське дослідження, а кількість цитат перевищує обсяг, виправданий поставленою метою роботи. В зв'язку з цим мета роботи та поставлені завдання не були досягнені. Робота може бути допущена до захисту (наступного року) після того як буде відкоригована та допрацьована і успішно пройде повторну перевірку на академічний плагіат.	
4	Робота містить навмисні текстові спотворення, передбачувані спроби укриття запозичень або інші прояви академічного плагіату. Робота містить фабрикацію або фальсифікацію даних. Робота не допускається до захисту.	
5	Інше:	

Підтвердження:

Запозичення, виявлені в роботі, є законними і не є плагіатом, оскільки:

1) у тексті дипломної роботи системами перевірки на плагіат виявлено схожість з деякими документами в частині загальновживаних обов'язкових словосполучень у стандартних бланках (титулка, бланк завдання), у структурі змісту, назвах розділів/підрозділів тощо та в назвах публікацій у переліку джерел посилання;

2) в якості запозичень системою було зафіксовано деякі послідовності вихідного коду і посилання на бібліотеки, які є стандартними мовними конструкціями програмування та не можуть розглядатися як об'єкт авторських прав і, відповідно, їх порушення;

3) усі запозичення є фрагментарними або мають належним чином оформленні посилання;

4) виявлені модифікації тексту не впливають на відсоток схожості.

Сумарний обсяг всіх запозичень, визначений системою виявлення збігів ідентичності/ схожості, складає 7,7% і адресується до 213 джерел, що, з урахуванням наведених обґрунтувань, відповідає характеру теми і свідчить на користь дипломної роботи.

Керівник _____ Н. І. Праворська

Гарант ОП _____ О. М. Яшина

Завідувач кафедри _____ Л. П. Бедратюк

також надає рекомендації для покращення швидкодії і зменшення затримки автоматичного масштабувальника в системі Kubernetes

5. Негативні сторони роботи Реалізований метод вирішує описану проблему холодного холодного старту контейнерів, але має недоліки в складності конфігурації і підбіру коректних параметрів запуску CHPA. Якщо недостатньо добре проаналізувати програмне забезпечення, відповідно якого треба зібрати метрики - реалізоване рішення може погіршити автоматичношо масштабувальника.

6. Оцінка графічного оформлення та пояснювальної записки роботи Графічне оформлення виконане відповідно до теми дипломної роботи з дотриманням вимог стандартів. Пояснювальна записка відповідає вимогам стандартів до її оформлення.

7. Відгук про роботу в цілому В цілому дипломна робота заслуговує позитивної оцінки. Весь матеріал дипломної роботи структурований, чіткий та послідовний. Усі розділи роботи є послідовними та логічними, що дозволяє чітко розуміти викладений матеріал у рамках тематики дипломної роботи. Графічний матеріал дозволяє наочно побачити доцільність та ефективність рішень, які були прийняті за основу для вирішення поставленої задачі.

8.Інші зауваження

9. Оцінка дипломної роботи Розглянувши позитивні та негативні сторони представленої дипломної роботи, можна зробити висновок, що вона заслуговує оцінки «відмінно».

РЕЦЕНЗЕНТ (прізвище, ім'я, по батькові, посада, місце роботи)
Мисенко Сергій Миколайович, д.т.н. професор, професор кафедри комп'ютерної інженерії та інформаційних систем

«2» грудня 2022 р. _____
(підпис)