Міністерство освіти і науки України
Харківський національний університет радіоелектроніки

Факультет Інформаційно-аналітичних технологій та менеджменту
(повна назва)
Кафедра Інформатики
(повна назва)

КВАЛІФІКАЦІЙНА РОБОТА
Пояснювальна записка

рівень вищої освіти другий (магістерський)

ДОСЛІДЖЕННЯ ВПЛИВУ КОНТЕЙНЕРИЗАЦІЇ НА МАСШТАБУВАННЯ МІКРОСЕРВІСІВ
(тема)

Виконав:
студент 2 курсу, групи ІНФм-23-1
Сердінов Б.А.
(прізвище, ініціали)

Спеціальності 122 Комп'ютерні науки
(код і повна назва спеціальності)

Тип програми освітньо-професійна

Освітня програма Інформатика
(повна назва освітньої програми)

Керівник проф. Машталір В.П.
(посада, прізвище, ініціали)

Допускається до захисту

Зав. кафедри _____ Кобилін О.А.
(підпис) (прізвище, ініціали)

2025 р.

Харківський національний університет радіоелектроніки

Факультет Інформаційно-аналітичних технологій та менеджменту
(повна назва)

Кафедра Інформатики
(повна назва)

Рівень вищої освіти другий (магістерський)

Спеціальність 122 Комп'ютерні науки
(код і повна назва)

Тип програми освітньо-професійна

Освітня програма Інформатика
(повна назва освітньої програми)

ЗАТВЕРДЖУЮ:
Зав. кафедри _____
(підпис)
«____»_____ 2025 р.

ЗАВДАННЯ
НА КВАЛІФІКАЦІЙНУ РОБОТУ

студентові Сердінову Богдану Андрійовичу
(прізвище, ім'я, по батькові)

1. Тема роботи Дослідження впливу контейнеризації на масштабування мікросервісів

затверджена наказом по університету від 25 листопада 2024 року № 1246Ст

2. Термін подання студентом роботи до екзаменаційної комісії 30 грудня 2024 р.

3. Вихідні дані до роботи алгоритми застосування контейнеризації у масштабуванні сучасних програмного забезпечення, перелік використовуваних програмних засобів: Docker, Kubernetes, Prometheus, Golang, теоретичні відомості про методи мастабування мікросервісів: горизонтальне масштабування, вертикальне масштабування.

4. Перелік питань, що потрібно опрацювати в роботі

1. Аналіз впливу контейнеризації на продуктивність і масштабованість програмних застосунків.
2. Особливості використання контейнеризації для забезпечення масштабування мікросервісів.
3. Розробка стратегій ефективного масштабування мікросервісів із використанням технологій контейнеризації.

5. Перелік графічного матеріалу із зазначенням креслеників, схем, плакатів, комп'ютерних ілюстрацій (п.5 включається до завдання за рішенням випускової кафедри) актуальність впливу контейнеризації на ефективне масштабування мікросервісів, постановка задачі, дослідження впливу контейнеризації на масштабування мікросервісів, схема архітектури мікросервісного застосунку у контейнеризованому середовищі та схема контейнеризації мікросервісів, ілюстрації горизонтального та вертикального мастабування за допомогою технології Kubernetes, графіки що демонструють продуктивність системи під час пікового навантаження, таблиці та діаграми, що ілюструють ефективність мастабування мікросервісів у контейнерах

6. Консультанти розділів роботи (п.6 включається до завдання за наявності консультантів згідно з наказом, зазначеним у п.1)

Найменування розділу	Консультант (посада, прізвище, ім'я, по батькові)	Позначка консультанта про виконання розділу	
		підпис	дата

КАЛЕНДАРНИЙ ПЛАН

№ з/п	Назва етапів роботи	Терміни виконання етапів роботи	Примітка
1	Отримання завдання на кваліфікаційну роботу	25.11.2024	
2	Аналіз завдання, підбір літератури	25.11.24-27.11.24	
3	Аналіз літератури з досліджуваної проблеми	27.11.24-28.11.24	
4	Аналіз методів контейнеризації	28.11.24-30.11.24	
5	Розробка методів контейнеризації та оркестрації	01.12.24-05.12.24	
6	Програмна реалізація	05.12.24-10.12.24	
7	Оформлення пояснювальної записки	10.12.24-16.12.24	
8	Перевірка на плагіат	17.12.2024	
9	Рецензування	20.12.2024	
10	Підготовка презентації та доповіді	25.12.2024	
11	Занесення роботи в електронний архів	05.01.2025	
12	Попередній захист кваліфікаційної роботи	09.01.2025	

Дата видачі завдання 25 листопада 2024 р.

Студент _____
(підпис)

Керівник роботи _____ проф. Машталір В.П.
(підпис) (посада, прізвище, ініціали)

РЕФЕРАТ/ABSTRACT

Пояснювальна записка до кваліфікаційної роботи: 77 с., 1 рис., 40 джерел.

МІКРОСЕРВІСИ, МАСШТАБУВАННЯ, КОНТЕЙНЕРИЗАЦІЯ, ОРКЕСТРАЦІЯ, DOCKER, DOCKER-COMPOSE, KUBERNETES, ВІЗУАЛІЗАЦІЯ, ПРОДУКТИВНІСТЬ СИСТЕМИ, ГНУЧКІСТЬ СИСТЕМИ.

Об’єктом дослідження є програмні системи, реалізовані у вигляді мікросервісів з використанням технологій контейнеризації, таких як Docker та Kubernetes.

Метою дослідження є розробка методів, які дозволяють оцінити ефективність масштабування мікросервісів у контейнерному середовищі та оптимізувати управління ресурсами для досягнення стабільної продуктивності.

Досліджено підходи до масштабування мікросервісів у контейнеризованому середовищі за допомогою Docker і Kubernetes, а також методи автоматичного управління контейнерами. Проведено аналіз впливу контейнеризації на продуктивність мікросервісів та розроблено алгоритм оптимізації розподілу ресурсів для стабільної роботи системи.

У результаті роботи було досліджено механізми масштабування мікросервісів за допомогою контейнеризації та розроблено рекомендації для підвищення ефективності управління розподіленими системами.

MICROSERVICES, SCALING, CONTAINERIZATION, ORCHESTRATION, DOCKER, DOCKER-COMPOSE, KUBERNETES, VISUALIZATION, SYSTEM PERFORMANCE, SYSTEM FLEXIBILITY.

The object of the research is software systems implemented as microservices using containerization technologies, such as Docker and Kubernetes.

The aim of the research is to develop methods that enable the assessment of microservices' scalability efficiency in a containerized environment and optimize resource management to achieve stable performance.

Approaches to scaling microservices in a containerized environment using Docker and Kubernetes were investigated, as well as methods for automatic container management. An analysis of the impact of containerization on microservices performance was conducted, and a resource allocation optimization algorithm was developed to ensure stable system operation.

As a result of the work, mechanisms for scaling microservices using containerization were studied, and recommendations for improving the efficiency of managing distributed systems were developed.

ЗМІСТ

ПЕРЕЛІК УМОВНИХ ПОЗНАЧЕНЬ, СИМВОЛІВ, ОДИНИЦЬ, СКОРОЧЕНЬ І ТЕРМІНІВ

K8s – Kubernetes

Back-end – серверна частина проєкту

Load balancer – розподілювач навантаження

Ingress – об’єкт для контролю трафіку

HPA – Horizontal Pod Autoscaler

CLI – Command Line Interface

ВСТУП

У сучасному світі технології розвиваються стрімкими темпами, що створює нові можливості для бізнесу, спрощує багато процесів і забезпечує доступ до інформації та сервісів для величезної кількості людей. Однак, разом із цими перевагами виникає й безліч викликів, зокрема потреба у забезпеченні стабільної роботи додатків при зростанні кількості користувачів. З кожним роком попит на інтернет-сервіси, мобільні додатки та хмарні рішення зростає, що ставить перед розробниками завдання створювати масштабовані системи, які зможуть безперебійно працювати навіть при значних навантаженнях.

Масштабування додатків стає критично важливим фактором для успішного функціонування бізнесу, оскільки від цього залежить здатність систем обробляти великі обсяги даних, виконувати тисячі або навіть мільйони запитів від користувачів одночасно та швидко реагувати на зміни в попиті. Незалежно від сфери застосування — будь то e-commerce, соціальні мережі, онлайн-банкінг або сервіси потокової передачі контенту — користувачі очікують швидкості, надійності та доступності. І якщо система не може забезпечити ці вимоги, це може призвести до втрати клієнтів, зниження довіри до бренду та фінансових втрат.

Отже, перед сучасними розробниками стоїть додаткова задача — не лише створювати функціональні та зручні програми, але й забезпечити їхню масштабованість. Це означає, що архітектура додатка має бути спроектована так, щоб легко адаптуватися до зростання кількості користувачів і запитів, зберігаючи при цьому стабільну продуктивність. У світі, де технології розвиваються так швидко, це завдання стає все більш важливим і складним, вимагаючи від інженерів знань та використання передових технологій, таких як контейнеризація, розподілені обчислення та хмарні сервіси.

Забезпечення масштабованості додатків — це комплексний процес, який включає оптимізацію ресурсів, управління навантаженням, моніторинг та автоматизацію. Успіх у цьому напрямку дозволяє бізнесу швидко реагувати на ринкові зміни, задовольняти потреби користувачів і ефективно використовувати інфраструктуру, забезпечуючи конкурентні переваги на ринку.

З роками еволюції було створено безліч рішень та підходів до архітектури додатків, що дозволяють розробникам ефективніше розгортати й масштабувати програми. Одним із найуспішніших підходів є мікросервісна архітектура, яка дозволяє розбивати складні додатки на менші незалежні компоненти, кожен з яких виконує певну функцію і може працювати окремо від інших. Така модульність дає можливість масштабувати кожен компонент індивідуально, залежно від вимог до навантаження, що значно підвищує гнучкість і стійкість системи.

Однак мікросервісна архітектура, незважаючи на свої численні переваги, також породжує низку проблем. У порівнянні з монолітними додатками, де всі компоненти тісно пов'язані й працюють разом, мікросервіси потребують надійної комунікації між собою через мережу, що ускладнює процеси управління та моніторингу. З'являються виклики з налаштування мережевої взаємодії, балансування навантаження, відмовостійкості та узгодженості даних між різними сервісами. Кожен мікросервіс має бути незалежним не лише функціонально, але й мати свою ізольовану інфраструктуру, що підвищує складність його розгортання та підтримки.

На початку мікросервісної ери використовувалась віртуалізація або інші традиційні методи, які, хоча й дозволяли розділити сервіси на окремі машини чи середовища, мали значні недоліки. Використання віртуальних машин вимагало великих ресурсів, оскільки кожна ВМ мала повноцінну операційну систему, а також великі витрати на обчислювальні ресурси й

пам’ять. Це обмежувало гнучкість та ефективність у керуванні великою кількістю мікросервісів і суттєво ускладнювало їх масштабування.

Згодом на допомогу прийшла технологія контейнеризації, яка стала новим стандартом в індустрії. Контейнери, на відміну від віртуальних машин, дозволяють ізолювати додатки та їх залежності у власних середовищах, але без необхідності запуску повноцінної операційної системи для кожного контейнера. Це значно зменшило витрати на ресурси і зробило процес розгортання мікросервісів швидшим та ефективнішим. Контейнеризація дозволила спростити управління залежностями, масштабування та забезпечити високий рівень узгодженості між середовищами розробки, тестування й продакшн.

Технології, такі як Docker, зробили контейнеризацію доступною для широкого кола розробників, а системи оркестрації, на кшталт Kubernetes, допомогли автоматизувати процеси розгортання і масштабування мікросервісів. Тепер кожен мікросервіс може бути легко упакований у контейнер, швидко розгорнутий у будь-якому середовищі і масштабований відповідно до потреб, що вирішує безліч попередніх проблем і робить контейнеризацію невід'ємною частиною сучасної мікросервісної архітектури.

Актуальність дослідження впливу контейнеризації на масштабування мікросервісів полягає в тому, що зростання попиту на високонавантажені, гнучкі та стійкі до збоїв додатки вимагає нових підходів до їх архітектури та розгортання. Контейнеризація дозволяє спростити процес масштабування мікросервісів, забезпечуючи ізоляцію, швидкість розгортання та ефективне використання ресурсів. Розуміння її впливу допомагає компаніям оптимізувати продуктивність систем та реагувати на вимоги ринку.

1 ОГЛЯД ВПЛИВУ КОНТЕЙНЕРІЗАЦІЇ НА МАСШТАБУВАННЯ МІКРОСЕРВІСІВ

1.1 Огляд основних методів масштабування мікросервісів

Масштабування мікросервісів є критично важливим для забезпечення стабільної роботи системи під час збільшення навантаження та зростання кількості користувачів. Мікросервіси дозволяють кожен окремий компонент системи масштабувати незалежно, що значно підвищує гнучкість і продуктивність. Одним із головних методів є горизонтальне масштабування, яке полягає в додаванні нових інстансів мікросервісу. Це дозволяє розподіляти вхідні запити між кількома інстансами, збільшуючи здатність системи обробляти більше запитів без зниження продуктивності. Горизонтальне масштабування є одним з найефективніших підходів, оскільки дає змогу додавати ресурси в будь-який момент і забезпечує високу відмовостійкість. Вертикальне масштабування передбачає збільшення ресурсів (процесорної потужності, пам'яті) одного інстансу сервісу. Однак цей метод має обмеження за кількістю доступних ресурсів і менш надійний у разі збоїв, оскільки вся система може залежати від одного потужного інстансу.

Автоматичне масштабування є ще одним важливим інструментом, який дозволяє динамічно адаптувати кількість інстансів мікросервісів залежно від навантаження. Хмарні сервіси, такі як AWS чи GCP, пропонують рішення, які автоматично збільшують або зменшують кількість інстансів на основі метрик, таких як використання CPU або кількість запитів. Це оптимізує витрати та забезпечує, що система завжди має достатньо ресурсів для обробки навантаження.

Також важливим є правильне балансування навантаження між інстансами. Використання балансувальника навантаження дозволяє

рівномірно розподіляти запити, запобігаючи перевантаженню окремих компонентів системи. Усе це дозволяє забезпечити масштабованість, високу доступність та надійність системи в умовах зростання навантаження, що є основною метою мікросервісної архітектури.

1.2 Горизонтальне масштабування

Горизонтальне масштабування — це масштабування вшир та один з ключових методів забезпечення високої доступності та продуктивності мікросервісної архітектури. Принцип роботи цього підходу полягає у збільшенні кількості екземплярів мікросервісів, а не у збільшенні потужності одного екземпляра.

При горизонтальному масштабуванні нові інстанції одного й того ж сервісу запускаються на кількох серверах або контейнерах. Ці інстанції можуть розміщуватися на фізичних або віртуальних машинах, у хмарному середовищі або в оркестрованих середовищах на кшталт Kubernetes.

Для того, щоб усі інстанції працювали ефективно, горизонтальне масштабування потребує використання балансувальника навантаження (load balancer). Балансувальник розподіляє запити між різними інстанціями рівномірно, що дозволяє уникнути перевантаження одного сервера чи контейнера і покращує загальну продуктивність системи.

Горизонтальне масштабування підвищує стійкість системи до збоїв. Якщо одна з інстанцій сервісу виходить з ладу, інші інстанції продовжують працювати, забезпечуючи безперебійність обслуговування. Це значно підвищує рівень доступності системи.

У хмарних середовищах, таких як AWS або Google Cloud, горизонтальне масштабування може бути автоматизоване за допомогою механізмів автоматичного масштабування (autoscaling). Це дозволяє автоматично запускати нові інстанції сервісів у відповідь на збільшення

навантаження та вимикати їх, коли навантаження зменшується, що знижує витрати на інфраструктуру.

Оскільки мікросервіси є незалежними, можна масштабувати лише ті сервіси, які потребують більше ресурсів, не зачіпаючи інші компоненти системи. Це дозволяє гнучко реагувати на різні види навантажень: для одного сервісу можна збільшити кількість інстанцій, тоді як інші можуть працювати з фіксованою кількістю.

Додавання інстанцій дозволяє обробляти більше запитів одночасно, що безпосередньо впливає на продуктивність системи. Це особливо корисно для сервісів, які мають високий рівень навантаження або потребують швидкої обробки великої кількості однотипних запитів (наприклад, API сервіси).

Але, горизонтальне масштабування також має свої труднощі. Наприклад, для сервісів, що зберігають стан (stateful services) потрібно писати логіку для узгодження стану між собою. Для stateless сервісів (stateless) цей процес значно простіший, оскільки екземлпляри не потребують синхронізації між собою. Також, коли кілька екземплярів одного сервісу працюють паралельно, виникають виклики з підтриманням консистентності даних, особливо в розподілених системах. Важливо забезпечити, щоб всі екземпляри працювали з актуальними даними і не виникало проблем з конфліктами записів.

Для розгортання чотирьох мікросервісів у Kubernetes, кожен з яких реалізує певну математичну операцію (додавання, віднімання, множення, ділення), доцільно створити окремі Deployment та Service ресурси для кожного мікросервісу. Це дозволяє Kubernetes ефективно керувати їх масштабуванням, забезпечуючи стабільний доступ до кожної служби у кластері, а також полегшує адміністрування. Такий підхід забезпечує ізоляцію між сервісами, що дозволяє виконувати оновлення чи зміни в одному з них без впливу на інші.
Однією з найважливіших проблем є моніторинг та збір логів, оскільки кожен окремий екземпляр генерує власні метрики та логи. Адже коли паралельно

працюють декілька екземплярів одного сервісу, потрібно використовувати інструменти для централізованого моніторингу та агрегування логів, щоб мати можливість аналізувати стан реплік, наприклад, за допомогою Prometheus або ELK Stack.

1.3 Вертикальне масштабування

Вертикальне масштабування — це масштабування, яке працює методом збільшення продуктивності системи за рахунок додавання більшої кількості ресурсів (процесорів, оперативної пам’яті, дискового простору) до одного сервера або віртуальної машини, на якій працює мікросервіс.

Вертикальне масштабування передбачає збільшення обчислювальних потужностей конкретного серверного середовища. Наприклад, можна додати більше CPU, оперативної пам'яті або швидший SSD-диск на фізичний або віртуальний сервер, що дозволяє обробляти більше запитів і виконувати більш ресурсоємні завдання.

Масштабування вгору є простішим у реалізації порівняно з горизонтальним масштабуванням. Не потрібно турбуватися про розподіл навантаження або синхронізацію між інстанціями сервісів. Достатньо лише додати більше ресурсів до наявного сервера або віртуальної машини.

Перевага вертикального масштабування полягає у тому, що воно не потребує змін у архітектурі програмного забезпечення. Для деяких систем це може бути критичним, оскільки не всі програми легко адаптуються до горизонтального масштабування. Зокрема, старі монолітні програми можуть бути складними для розподілу на окремі мікросервіси, і вертикальне масштабування стає єдиним простим рішенням.

Оскільки вся система залишається на одному сервері, знижується складність моніторингу та управління. Немає потреби використовувати

балансувальники навантаження чи розподілені системи для підтримки консистентності, що спрощує підтримку і експлуатацію системи.

Основні переваги вертикального масштабування полягають у його простоті реалізації та швидкій реакції на зростання навантаження. Воно дозволяє збільшити продуктивність без необхідності змінювати архітектуру системи, що особливо корисно для монолітних додатків або систем, які складно розподілити на мікросервіси. Також вертикальне масштабування зменшує складність управління, оскільки немає потреби у балансувальниках навантаження чи синхронізації між інстанціями сервісів. Це дозволяє швидко і ефективно збільшити ресурси за умов помірного зростання навантаження.

Але, вертикальне масштабування має свої недоліки. Основним недоліком вертикального масштабування є його обмеженість. Є фізичні обмеження на те, скільки ресурсів можна додати до одного сервера. Наприклад, можна додати лише певну кількість процесорів або оперативної пам’яті, після чого зростання продуктивності зупиниться.

Всі ресурси зосереджені в одному сервері, і якщо цей сервер виходить з ладу, вся система стає недоступною. Це знижує загальну стійкість системи до збоїв у порівнянні з горизонтальним масштабуванням, де відмова однієї інстанції не призводить до відмови всього сервісу.

Додавання потужних ресурсів до одного сервера може бути значно дорожчим, ніж розподіл навантаження на кілька менш потужних машин. Крім того, якщо ресурси сервера використовуються неефективно, це призводить до перевитрати бюджетів на інфраструктуру.

Вертикальне масштабування зазвичай стосується лише обчислювальних ресурсів, але не покриває масштабування інших аспектів системи, таких як пропускна здатність мережі або швидкість доступу до баз даних. Таким чином, воно не завжди є повноцінним рішенням для масштабування системи.

1.4 Контейнеризація та її роль у мікросерверній архітектурі

Контейнеризація — це технологія, яка дозволяє ізолювати додатки разом із усіма їхніми залежностями в окремі контейнери, забезпечуючи легку переносимість та узгодженість між різними середовищами. Контейнер можна порівняти з легким віртуальним середовищем, яке містить усе необхідне для виконання конкретного додатку: код, бібліотеки, системні інструменти і налаштування. Найпопулярнішим інструментом для контейнеризації є Docker, який значно спрощує процес створення, управління і розгортання додатків.

Основою контейнеризації є образи — шаблони, які містять усе необхідне для роботи додатка, включаючи операційну систему, середовище виконання, додаткові залежності та сам додаток. Образи використовуються для створення контейнерів і забезпечують стандартизоване середовище для додатка незалежно від того, на якій інфраструктурі він працює. Завдяки шаровій структурі образів, Docker зберігає проміжні шари, що дозволяє оптимізувати процес створення та оновлення контейнерів. Це робить образи ефективними та економними в плані ресурсів, оскільки однакові частини коду і залежностей можуть використовуватися повторно.

Завдяки контейнеризації та образам мікросервіси стають більш незалежними, масштабованими та керованими. Це дозволяє компаніям швидше розробляти нові функції, тестувати їх у реальних умовах і забезпечувати стабільну роботу в умовах зростання навантаження. Образи роблять цей процес ще більш передбачуваним і портативним, оскільки будь-яка версія додатка може бути легко відтворена і запущена на будь-якій системі, що підтримує Docker, забезпечуючи таким чином повну сумісність та стабільність середовища.

Контейнери забезпечують ізоляцію на рівні операційної системи, що дозволяє кожному мікросервісу працювати у власному середовищі, окремо від інших додатків. Це зменшує ризик конфліктів між різними версіями

залежностей і підвищує безпеку, оскільки проблеми, що виникають в одному контейнері, не впливають на роботу інших. Кожен контейнер має обмежений доступ до системних ресурсів, що підвищує рівень захищеності системи в цілому.

Контейнери легко масштабуються, дозволяючи швидко збільшувати або зменшувати кількість екземплярів додатка у відповідь на зміни в навантаженні. Використання інструментів оркестрації, таких як Kubernetes, автоматизує процеси розгортання, балансування навантаження та управління масштабованістю, що дозволяє компаніям оптимізувати використання ресурсів і підвищити стійкість системи до збоїв.

Контейнерні образи можна версіонувати, що дозволяє чітко відслідковувати зміни в коді та залежностях. Це робить процес розробки більш контрольованим і прозорим, сприяючи безперервній інтеграції та безперервному розгортанню. Завдяки образам, розробники можуть легко розгортати різні версії додатків у різних середовищах, що полегшує тестування і забезпечує швидке оновлення додатків без переривання їхньої роботи.

Контейнери є надзвичайно портативними, оскільки вони містять все необхідне для запуску додатка, включаючи операційну систему, середовище виконання та залежності. Це означає, що додатки можуть запускатися однаково на різних платформах — як на локальних машинах, так і в хмарних середовищах. Контейнеризація дозволяє уникати проблем сумісності, оскільки середовище всередині контейнера завжди залишається однаковим, незалежно від хост-системи.

Контейнери споживають менше ресурсів порівняно з традиційними віртуальними машинами, оскільки вони спільно використовують ядро операційної системи хоста. Це дозволяє ефективніше використовувати апаратні ресурси та запускати більше додатків на одному сервері, що знижує витрати на інфраструктуру.

Для створення контейнера для додатка, процес є досить простим завдяки використанню Dockerfile. У Dockerfile можна описати всі кроки, необхідні для створення образу: від установки залежностей до компіляції та запуску додатка.

Лістинг 1.1 Приклад Dockerfile:

```
FROM golang:1.20-alpine
WORKDIR /app
COPY . .
RUN go build -o main .
CMD ["/app/main"]
```

1.5 Оркестрація та її роль у масштабуванні мікросервісної архітерктурі

Оркестрація — це процес автоматизації управління, розгортання та координації контейнеризованих додатків у середовищах мікросервісної архітектури. У сучасному світі, де потреби в швидкості та адаптивності постійно зростають, оркестрація стає критично важливим аспектом для успішного впровадження та функціонування мікросервісів. Цей підхід спрощує роботу з контейнерами, зокрема в умовах динамічних і масштабованих систем, де можуть виникати різноманітні виклики, пов’язані з управлінням великою кількістю мікросервісів.

Одним з найпопулярніших інструментів для оркестрації є Kubernetes, який надає розробникам можливість автоматично розгортати, масштабувати та керувати контейнерами. Kubernetes підтримує автоматичне балансування навантаження, що дозволяє рівномірно розподіляти запити та уникати перевантаження окремих мікросервісів. Це підвищує загальну

продуктивність системи і забезпечує високу доступність додатків, що вкрай важливо в умовах високих навантажень.

Крім того, оркестрація в Kubernetes дозволяє автоматично масштабувати мікросервіси в реальному часі залежно від навантаження. Це означає, що система може регулювати кількість реплік мікросервісів на основі реальних запитів користувачів, що робить систему більш ефективною і чутливою до змін. Наприклад, під час пікових навантажень Kubernetes може автоматично додавати нові інстанси мікросервісів, а коли навантаження знижується, зменшувати їх кількість. Це забезпечує оптимальне використання ресурсів.

Kubernetes є "оркестратором з управлінням станом", що означає, що система завжди намагається підтримувати заданий стан мікросервісів. Якщо контейнер виходить з ладу, Kubernetes автоматично перезапускає його, що підвищує надійність і зменшує час простою. Ця автоматизація дозволяє розробникам зосередитися на створенні нових функцій замість постійного моніторингу стану системи.

Оркестрація також сприяє реалізації стратегій відновлення після збоїв. Kubernetes може бути налаштований для автоматичного перенаправлення трафіку до резервних копій або для масштабування системи в разі високого навантаження, що дозволяє підтримувати стабільну роботу мікросервісів. Це особливо важливо для бізнесів, де доступність сервісу є критичною.

Інтеграція оркестрації з процесами безперервної інтеграції та доставки (CI/CD) також заслуговує на увагу. Kubernetes полегшує автоматизацію розгортання нових версій додатків, зменшуючи ризики, пов’язані з оновленнями та дозволяючи швидше впроваджувати нові функції. Завдяки цьому, команди можуть більш оперативно реагувати на зміни вимог користувачів або ринку, що стає величезною перевагою в конкурентному середовищі.

Моніторинг та логування також мають велике значення в контексті оркестрації. Інструменти, такі як Prometheus і Grafana, можуть бути

інтегровані з Kubernetes для спостереження за станом мікросервісів. Це дозволяє відстежувати продуктивність, виявляти проблеми на ранніх стадіях і приймати рішення на основі даних, що є важливим аспектом управління сучасними додатками.

Ще одним важливим аспектом є управління ресурсами. Kubernetes дозволяє визначати, скільки ресурсів (CPU, пам’яті) потрібно кожному контейнеру, щоб оптимізувати використання інфраструктури. Це дозволяє ефективно управляти ресурсами в умовах зростаючих навантажень і обмежених ресурсів. Завдяки можливостям управління ресурсами, компанії можуть знижувати витрати на інфраструктуру і підвищувати ефективність своїх рішень.

Важливо зазначити, що Kubernetes підтримує багатопровайдерність, що дозволяє розгортати мікросервіси в різних хмарах (громадських, приватних) і на локальних серверах, спрощуючи управління гібридними інфраструктурами. Це робить його потужним інструментом для сучасних компаній, які прагнуть адаптуватися до швидко змінюваного ринку і забезпечити високий рівень сервісу для своїх клієнтів. Оркестрація мікросервісів через Kubernetes не тільки підвищує продуктивність, але й забезпечує їхню гнучкість, безпеку та масштабованість.

1.6 Динамічне масштабування

Динамічне масштабування — це ключовий аспект сучасних систем, що дозволяє автоматично регулювати кількість активних ресурсів, таких як контейнери або сервіси, у відповідь на зміни навантаження. Це стає особливо важливим у контексті мікросервісної архітектури, де адаптація до змін у запитах користувачів може суттєво вплинути на продуктивність та ефективність роботи системи.

Динамічне масштабування забезпечується завдяки інтеграції моніторингових систем, які в реальному часі відстежують ключові метрики, такі як завантаження процесора, використання пам'яті, швидкість відповіді та обсяг трафіку. Коли система виявляє, що навантаження перевищує певні порогові значення, вона автоматично реагує, розгортаючи нові екземпляри контейнерів або зменшуючи їхню кількість у разі зниження попиту.

Динамічне масштабування дозволяє ефективніше використовувати ресурси, оскільки компанії можуть швидко адаптувати кількість активних контейнерів відповідно до реальних потреб. Наприклад, під час пікових навантажень, таких як сезонні розпродажі, система може миттєво додавати нові контейнери, щоб впоратися з підвищеним трафіком, а після закінчення акцій — зменшити їхню кількість, що знижує витрати на інфраструктуру.

Гнучкість, яку забезпечує динамічне масштабування, є критично важливою для компаній, які стикаються з коливаннями в попиті на свої послуги. Системи можуть швидко реагувати на зміни в навантаженні, що сприяє підвищенню доступності та стабільності сервісів. Автоматичне масштабування також дозволяє швидше відновлюватися після збоїв, оскільки нові контейнерні екземпляри можуть бути запущені в разі виходу з ладу попередніх.

Важливу роль у реалізації динамічного масштабування відіграють технології оркестрації, зокрема Kubernetes. Ця платформа надає механізми автоматичного масштабування, такі як Horizontal Pod Autoscaler (HPA), який змінює кількість реплік контейнерів на основі моніторингу метрик, таких як завантаження процесора або пам'яті. Інтеграція з системами моніторингу, такими як Prometheus, дозволяє отримувати детальну аналітику та забезпечує ще більше можливостей для адаптації системи до змінюваних умов.

Динамічне масштабування також може бути реалізовано за допомогою хмарних платформ, таких як AWS, Azure або Google Cloud. Ці сервіси пропонують власні механізми автоматичного масштабування, які

допомагають підприємствам створювати гнучкі архітектури, здатні до адаптації без значних інвестицій у фізичну інфраструктуру.

1.7 Тестування та розгортання

Тестування та розгортання є важливими етапами в життєвому циклі розробки мікросервісів, які забезпечують їх стабільність, продуктивність та відповідність вимогам. У контексті контейнеризації та оркестрації ці процеси отримують нові можливості завдяки автоматизації, що істотно підвищує ефективність роботи команд розробників і операторів.

Тестування мікросервісів — це складний процес, оскільки кожен сервіс може взаємодіяти з іншими, створюючи складну екосистему. Тому важливо забезпечити не лише функціональне тестування кожного сервісу, а й перевірку їхньої інтеграції. Використання контейнерів спрощує цю задачу, адже дозволяє запускати тести в середовищі, яке точно відповідає продуктивному, що зменшує ризик помилок, пов'язаних із конфігураціями.

Серед методів тестування, які активно використовуються в мікросервісній архітектурі, виділяються юніт-тестування, інтеграційне тестування та енд-то-енд тестування. Юніт-тестування фокусується на перевірці окремих компонентів, у той час як інтеграційне тестування перевіряє взаємодію між кількома сервісами. Енд-то-енд тестування забезпечує перевірку функціональності всієї системи в цілому, що є критично важливим для підтвердження правильності інтеграції всіх мікросервісів.

У рамках розгортання мікросервісів, контейнеризація надає можливість автоматизувати процес, зменшуючи кількість ручних дій, які можуть призвести до помилок. За допомогою таких інструментів, як Docker та Docker Compose, команди можуть легко створювати образи для своїх сервісів, що включають всі необхідні залежності, і швидко їх розгортати на різних середовищах.

Процеси розгортання можуть варіюватися від простих сценаріїв, де новий код просто замінює старий, до складних, де використовується стратегія Blue-Green Deployment або Canary Releases. У першому випадку, нова версія мікросервісу розгортається поруч зі старою, а користувачі поступово перемикаються на нову версію. Це дозволяє знизити ризик помилок, адже при виявленні проблем можна швидко повернутися до попередньої версії.

У поєднанні з оркестраторами, такими як Kubernetes, розгортання мікросервісів стає ще простішим і передбачуванішим. Kubernetes надає можливості для автоматичного розгортання, масштабування та управління контейнерами, що дає змогу зосередитися на розробці замість управлінських завдань. Завдяки інтеграції з CI/CD (безперервна інтеграція/безперервне розгортання) процеси тестування та розгортання можуть бути автоматизовані, що дозволяє зменшити час від написання коду до його впровадження в продуктивне середовище.

Автоматизація тестування та розгортання не лише покращує ефективність, але й підвищує якість програмного забезпечення. Завдяки автоматизованим тестам, які запускаються при кожному внесенні змін до коду, команди можуть швидше виявляти та усувати помилки, що значно скорочує час випуску нових функцій.

Крім того, важливими аспектами тестування та розгортання є контроль версій та конфігурація середовищ. Використання систем контролю версій, таких як Git, дозволяє командам відслідковувати зміни, повертатися до попередніх версій коду та управляти конфігураціями мікросервісів у різних середовищах. Таким чином, забезпечується стабільність і передбачуваність у процесі розгортання нових функцій.

Додатково, це спрощує інтеграцію змін, зменшує ризик конфліктів між командами розробників та забезпечує прозорість у процесах розробки та тестування, що є критично важливим для великих команд і складних проєктів.

1.8 Підтримка гібридних та мультихмарних середовищ

Підтримка гібридних та мультихмарних середовищ є невід'ємною складовою сучасної інфраструктури, яка дозволяє організаціям максимально використовувати переваги різних хмарних рішень та локальних ресурсів. У таких умовах бізнеси можуть розгортати свої додатки не тільки на локальних серверах, а й у публічних або приватних хмарах, що забезпечує високий рівень гнучкості та адаптивності. Це, у свою чергу, сприяє швидшій інтеграції нових технологій та інновацій, що є критично важливим у конкурентному середовищі.

Цей підхід дозволяє компаніям оптимізувати витрати, адже вони можуть використовувати ресурси найбільш вигідно, адаптуючи їх відповідно до змінних вимог. Наприклад, у періоди пікового навантаження, коли потреби в обчислювальних потужностях зростають, організації можуть без труднощів розширити свої ресурси, звертаючись до публічних хмар. Навпаки, у моменти, коли навантаження зменшується, можна зменшити використання ресурсів у хмарі, повертаючись до локальної інфраструктури. Це дозволяє уникнути зайвих витрат і забезпечити економічну ефективність.

Крім того, гібридні та мультихмарні середовища забезпечують підвищену надійність і безпеку. Завдяки дистрибуції даних між кількома платформами, компанії можуть уникнути ризиків, пов'язаних з простоєм або втратами даних. Це також дозволяє реалізувати стратегії резервування, коли дані дублюються на кількох серверах або в різних хмарах, забезпечуючи їх доступність навіть у разі збою в одній з систем. Керування даними в таких середовищах також вимагає впровадження інструментів для моніторингу та управління, які забезпечують цілісність та доступність інформації.

Окрім цього, важливим аспектом є інтеграція різноманітних сервісів та додатків, які можуть працювати в різних середовищах. Це вимагає застосування сучасних методів, таких як API-інтеграція та контейнеризація, що дозволяє забезпечити безшовну взаємодію між компонентами. У

результаті, підтримка гібридних та мультихмарних середовищ сприяє створенню більш стійкої та адаптивної архітектури, що відповідає вимогам сучасного бізнесу, дозволяючи організаціям ефективно реагувати на зміни ринку та швидко впроваджувати інновації.

1.9 Вплив контейнеризації на безпеку додатку

Зростання використання мікросервісів зумовлено їхньою здатністю підвищити швидкість розробки, масштабованість та гнучкість. Однак разом з цими перевагами виникають і нові виклики безпеки. В умовах, коли мікросервіси активно взаємодіють між собою, будь-яка вразливість може мати серйозні наслідки для цілісності системи.

Контейнеризація стала одним із ключових факторів, що вплинули на розвиток безпеки у мікросервісних архітектурах. Використання контейнерів дозволяє ізолювати різні компоненти системи, що, в свою чергу, підвищує загальний рівень безпеки. Кожен мікросервіс може бути упакований у свій власний контейнер, що зменшує ризик впливу вразливостей одного сервісу на інші. Це означає, що навіть якщо один мікросервіс зазнає атаки або компрометації, інші сервіси залишаються захищеними.

Контейнеризація забезпечує ізоляцію на рівні операційної системи, що дозволяє запускати кілька контейнерів на одному сервері без ризику перехоплення даних або ресурсів. Це особливо важливо в умовах мікросервісної архітектури, де багато сервісів можуть взаємодіяти один з одним. Ізоляція також полегшує управління правами доступу: адміністратори можуть точно налаштувати, які ресурси та мережеві порти доступні для кожного контейнера, що підвищує контроль над безпекою.

Контейнери дозволяють швидко і легко оновлювати мікросервіси, що є важливим аспектом безпеки. Вразливості часто виявляються в програмному забезпеченні, і регулярне оновлення допомагає зменшити ризик експлуатації

цих вразливостей. Завдяки контейнеризації, організації можуть автоматизувати процес оновлення, що дозволяє швидко застосовувати патчі без значних зусиль або зупинок у роботі системи.

Контейнеризація також сприяє впровадженню безпечних мережевих політик, що обмежують взаємодію між мікросервісами. Наприклад, можна використовувати сервісні сітки (service meshes) для контролю трафіку між контейнерами, що дозволяє реалізувати шифрування трафіку, аутентифікацію та авторизацію. Це підвищує рівень безпеки даних під час їх передачі між мікросервісами, що є важливим аспектом для запобігання перехопленню та зловживанню інформацією.

1.10 Постановка задачі дослідження

Таким чином, контейнеризація є актуальним інструментом для забезпечення масштабованості та ефективності мікросервісної архітектури. Тому ставиться завдання дослідити вплив контейнеризації на масштабування мікросервісів, зокрема, в умовах зростаючого навантаження та необхідності швидкого розгортання.

Об’єктом дослідження є програмні системи, реалізовані у вигляді мікросервісів з використанням технологій контейнеризації, таких як Docker та Kubernetes.

Метою дослідження є розробка методів, які дозволяють оцінити ефективність масштабування мікросервісів у контейнерному середовищі та оптимізувати управління ресурсами для досягнення стабільної продуктивності.

Для досягнення мети необхідно вирішити такі завдання:

– провести аналіз існуючих підходів до контейнеризації та масштабування мікросервісів;

– розробити методологію для оцінки ефективності контейнерного середовища при масштабуванні;

– реалізувати тестову модель, що дозволяє дослідити вплив контейнеризації на продуктивність та ефективність використання ресурсів при масштабуванні мікросервісів;

– провести експериментальне дослідження на основі розробленої моделі для визначення оптимальних параметрів масштабування.

2 ПРАКТИЧНЕ ЗАСТОСУВАННЯ МЕТОДІВ КОНТЕЙНЕРИЗАЦІЇ НА МАСШТАБУВАННЯ МІКРОСЕРВІСІВ

2.1 Вибір архітектури застосунку

Архітектура проєкту є одним із ключових аспектів під час створення проєктів, адже грамотно спроєктована система значно спрощує її підтримку та розширення. Модульна структура та ізольовані компоненти дозволяють легко змінювати, додавати чи видаляти функціональність, не впливаючи на інші частини системи.

Правильно розроблена архітектура дає змогу ефективно масштабувати проєкт, коли це стає необхідним. Вона дозволяє розподілити навантаження, підвищити продуктивність і забезпечити гнучкість системи. Якщо проєкт має продуману архітектуру, підтримка й розвиток проєкту стають значно простішими. Така архітектура гарантує надійність, дає змогу ізолювати проблеми, забезпечує швидке відновлення при збоях і високу доступність системи.

Добре спроєктована архітектура сприяє швидкості розробки, дозволяє ефективно розподіляти завдання між командами, оптимізує процес розробки та забезпечує високу продуктивність. Якісна архітектура також сприяє ефективному використанню та повторному використанню коду, оскільки модульна структура та належний поділ відповідальності дозволяють створювати універсальні компоненти, що можуть використовуватися в різних частинах проєкту.

Існує кілька основних архітектур для розробки застосунків. Монолітна архітектура — це традиційний підхід, де весь застосунок є єдиним компонентом, в якому всі функції та модулі знаходяться в одному великому пакеті. Це простий спосіб розробки, але зі зростанням розміру проєкту така архітектура ускладнює його підтримку та масштабування.

Клієнт-серверна архітектура — застосунок розділяється на клієнтську та серверну частини. Клієнт взаємодіє з користувачем через інтерфейс, а сервер обробляє бізнес-логіку і зберігає дані. Цей підхід дозволяє розподілити навантаження, що полегшує розширення та модифікацію системи.

Розподілена архітектура — компоненти застосунку розташовані на різних фізичних чи логічних вузлах, які працюють разом. Це може включати розподілені сервери, кластери, мережі, мікросервіси, що забезпечує високу доступність, масштабованість та надійність системи.

Сервісно-орієнтована архітектура (SOA) — застосунок поділяється на набір сервісів, які є незалежними компонентами зі своїми інтерфейсами та функціональністю, і взаємодіють один з одним.

Однак ці архітектури мають певні недоліки. Монолітна архітектура ускладнює розгортання та оновлення системи. Зміни в одній частині можуть впливати на весь застосунок, що вимагає ретельного тестування перед розгортанням. Різні компоненти часто тісно пов'язані, що ускладнює підтримку та розширення, а також обмежує масштабованість. Мікросервісна архітектура також має переваги, але зростання кількості сервісів підвищує складність управління ними. Потрібно ретельно планувати координацію й моніторинг сервісів. Взаємодія між сервісами через мережу може знизити продуктивність і ускладнює тестування системи, що вимагає багато ресурсів.

Порівнюючи обидва підходи, мікросервісна архітектура здається привабливішою для складних проєктів, що передбачають значне розширення. Незалежна розробка, розгортання та масштабування окремих компонентів, гнучкість і можливість швидких змін, масштабованість та горизонтальний розподіл навантаження є вирішальними факторами для вибору мікросервісного підходу. Крім того, мікросервіси дозволяють командам працювати автономно, що прискорює впровадження нових функцій і зменшує залежність між різними частинами системи. Завдяки цьому підходу

легше інтегрувати нові технології та підтримувати актуальність окремих сервісів у швидкозмінному середовищі.

2.2 Мікросервісна архітектура

Мікросервісна архітектура є підхідом до розробки програмного забезпечення, де застосунок складається з низки незалежних компонентів. Кожен мікросервіс виконує окрему функцію та відповідає за конкретний бізнес-процес або задачу. Ці сервіси взаємодіють один з одним через стандартизовані API (наприклад, HTTP або gRPC) і працюють автономно один від одного.

Кожен мікросервіс є окремим компонентом, який можна розгортати, оновлювати та масштабувати без впливу на інші частини системи. Це дозволяє командам працювати автономно над різними сервісами, що підвищує ефективність розробки.

Кожен мікросервіс відповідає за свою частину бізнес-логіки і має чітко визначену задачу. Наприклад, один сервіс може обробляти платежі, інший — управляти користувачами, а ще один — здійснювати аналітику.

Мікросервіси дозволяють масштабувати тільки ті частини системи, які цього потребують. Наприклад, якщо збільшується навантаження на платіжну систему, можна масштабувати тільки платіжний мікросервіс, не зачіпаючи інші компоненти.

У мікросервісній архітектурі можна використовувати різні технології, мови програмування та бази даних для кожного сервісу. Це дозволяє вибирати оптимальні інструменти для кожного компонента.

Оскільки мікросервіси ізольовані один від одного, збій в одному з них не призведе до відмови всього застосунку. Це підвищує загальну стійкість системи, адже несправності можна ізолювати й локально усунути.

Мікросервіси можна швидко оновлювати та виправляти помилки. Це спрощує реліз нових версій без потреби зупиняти весь застосунок.

Мікросервісна архітектура дозволяє великим командам працювати над окремими частинами системи незалежно одна від одної, що скорочує час розробки.

Мікросервісна архітектура найчастіше використовується в масштабованих системах із розподіленим навантаженням, які потребують частих оновлень, високої гнучкості та незалежного розгортання компонентів.

2.3 Вибір технологій для розробки

Мікросервісна архітектура на основі Golang з інструментами gRPC, gRPC-Gateway, Prometheus і OpenTelemetry Tracing дозволяє створювати високонавантажені, масштабовані та спостережувані додатки, що відповідають принципам 12 Factors App. Така архітектура сприяє модульності та спрощує підтримку великих систем, дозволяючи розробникам швидше впроваджувати зміни і дотримуватися стандартів сучасного DevOps.

Golang є основною мовою для розробки сервісів у додатку завдяки її продуктивності, простоті та природній підтримці конкурентності. У Golang зручно розробляти RESTful API, а також використовувати інші протоколи для комунікації між сервісами. Його стандартна бібліотека включає базові інструменти для роботи з мережею, що особливо зручно для мікросервісної архітектури.

Протокол gRPC забезпечує високопродуктивну комунікацію між мікросервісами за допомогою протоколу HTTP/2 та Protobuf (Protocol Buffers) для серіалізації даних. Це дозволяє сервісам спілкуватися швидко та з мінімальним навантаженням на мережу. Для зовнішніх клієнтів, яким зручніше використовувати REST API, використовується gRPC-Gateway — проксі, що автоматично транслює HTTP-запити у формат gRPC і навпаки,

забезпечуючи зручність взаємодії з фронтендом та іншими сервісами, які очікують REST-інтерфейси.

Prometheus використовується для моніторингу та збору метрик у системі. Він дозволяє збирати дані про продуктивність додатку, як-от час відповіді на запити, кількість активних запитів, стан здоров'я сервісів тощо. Використання prometheus/client_golang бібліотеки в Go дозволяє легко інтегрувати моніторинг, створювати та оновлювати метрики безпосередньо в коді сервісів. Всі метрики експортуються на Prometheus, де можуть бути налаштовані порогові значення для сповіщень за допомогою Alertmanager.

Для трасування запитів у системі використовується OpenTelemetry Tracing, що дозволяє відстежувати повний життєвий цикл кожного запиту у межах мікросервісної системи. Бібліотека OpenTelemetry для Golang дозволяє створювати «спани» (span) для кожної операції, що допомагає відстежувати і аналізувати проходження запиту між сервісами, виявляти затримки та локалізувати проблеми. Дані трасування можна експортувати у спеціалізовані інструменти візуалізації, такі як Jaeger або Zipkin, для подальшого аналізу.

Мікросервісна архітектура відповідає принципам 12 Factors App, які забезпечують масштабованість, надійність та легкість розгортання додатків. Всі конфігураційні параметри зберігаються в середовищі (наприклад, як змінні оточення), що дозволяє легко змінювати налаштування без перекомпіляції коду. Кожен сервіс має свої залежності, що спрощує підтримку та уникнення конфліктів версій бібліотек. Використання Prometheus і OpenTelemetry забезпечує надійне логування та моніторинг, допомагаючи виявляти проблеми в системі. Мікросервіси можуть масштабуватись незалежно один від одного, дозволяючи додаткам обробляти велику кількість запитів.

Загалом, така інфраструктура надає всі необхідні інструменти для створення масштабованого, надійного і продуктивного додатку, який легко розгортається і піддається моніторингу та відлагодженню.

2.4 Розбір архітектури серверної частини

Після визначення набору інструментів для системи розглянемо її архітектурне проектування. Оскільки основна увага в роботі приділяється інфраструктурній частині, серверна архітектура системи має досить спрощену структуру, орієнтовану на розподіл функціональних обов’язків між мікросервісами. Кожен мікросервіс відповідає за виконання певної математичної операції, що дозволяє підвищити гнучкість і масштабованість системи.

Additional Service — мікросервіс, який обробляє запити для виконання операції додавання двох чисел. Це забезпечує модульність і дозволяє ізолювати логіку додавання, що спрощує тестування та оптимізацію операції в окремому сервісі.

Subtraction Service — мікросервіс, відповідальний за операцію віднімання. Подібно до інших математичних сервісів, ізольованість операції дозволяє уникати взаємозалежностей з іншими сервісами та зменшує ризик конфліктів у реалізації бізнес-логіки.

Multiplication Service — мікросервіс, який реалізує операцію множення. Його ізоляція сприяє ефективному управлінню ресурсами для виконання обчислень, зокрема при масштабуванні або оптимізації під високонавантажені запити.

Division Service — мікросервіс для виконання операцій ділення. Використання окремого сервісу забезпечує відокремлення обробки операції ділення, що дозволяє врахувати специфічні вимоги до точності та обробки помилок, таких як ділення на нуль.

Gateway Service — цей мікросервіс виступає в ролі шлюзу, обробляючи вхідні HTTP-запити і виконуючи трансляцію цих запитів у виклики внутрішніх сервісів. Gateway Service реалізує REST API, забезпечуючи зручний доступ для зовнішніх клієнтів до всіх математичних операцій. Gateway Service перетворює HTTP/1-запити у HTTP/2, що покращує

продуктивність і зменшує затримки завдяки мультиплексуванню запитів, ефективнішому використанню з’єднань і зниженню навантаження на сервери. HTTP/2 дозволяє передавати кілька запитів і відповідей по одному з’єднанню, що значно покращує ефективність обробки запитів і знижує затримки в мережах з високим навантаженням.

На рисунку 2.1 показана архітектура серверної частини, яка демонструє організацію та взаємодію основних мікросервісів системи.

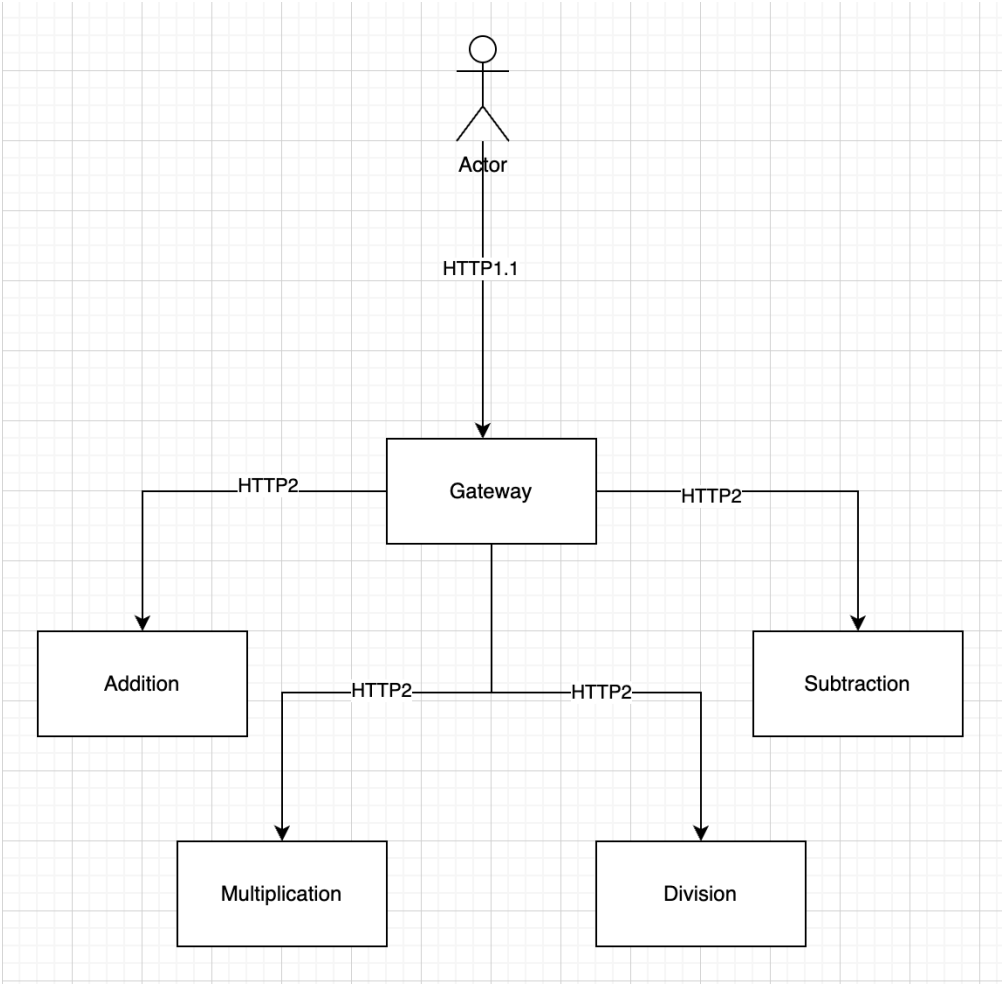

Рис 2.1 — Діаграма архітектури серверної частини

2.5 Контейнерізація додатку

Контейнеризація додатку — це підхід до розгортання програм, який дозволяє упаковувати додаток разом з усіма його залежностями, конфігураціями та бібліотеками в єдину ізольовану одиницю, що називається контейнером. Контейнери працюють незалежно від операційної системи хоста, завдяки чому забезпечують стабільну і передбачувану роботу додатку у будь-якому середовищі, будь-то локальна машина розробника, тестове середовище чи масштабований хмарний кластер.

Контейнеризація значно спрощує процес розгортання та підтримки додатків, дозволяючи одночасно запускати кілька версій одного додатку або навіть різні додатки на одній машині без ризику конфліктів між ними. Це досягається завдяки використанню ізольованих середовищ виконання, де кожен контейнер має свої власні ресурси і не заважає іншим контейнерам на тому ж сервері.

Контейнери легші та швидші за традиційні віртуальні машини, оскільки вони спільно використовують ядро операційної системи хоста, що дозволяє їм запускатися за частки секунди. Це робить контейнеризацію особливо ефективною для мікросервісних архітектур, де кожен компонент системи може бути розгорнутий, оновлений та масштабований окремо, з мінімальним впливом на інші частини системи.

Така гнучкість дозволяє ефективно реагувати на зміни навантаження, легко масштабуючи конкретні сервіси та розподіляючи ресурси відповідно до потреб.

Завдяки контейнеризації додатків можливо досягти кращої продуктивності, простоти підтримки та можливості швидкого оновлення, оскільки зміни в одному контейнері не впливають на інші, і будь-який контейнер може бути швидко замінений на нову версію без переривання роботи системи.

Лістинг 2.1 Реалізація Dockerfile для застосунку:

```
FROM golang:1.21.6 AS builder
RUN git clone https://github.com/magefile/mage && \ cd mage && \ go run bootstrap.go
ADD . /go/src/infra-example
WORKDIR /go/src/infra-example
RUN mage services
RUN mage -compile ./build/mage

FROM alpine:3
RUN apk add --no-cache libc6-compat
WORKDIR /root/infra-example/
COPY --from=builder /go/src/infra-example/build/* ./
COPY --from=builder /go/src/infra-example/docs ./docs/ ENV PATH="/root/infra-example:${PATH}"
```

2.6 Оркестрація

Оркестрація – це процес автоматизації управління, координації та упорядкування різних робочих процесів або служб у комплексній системі, що складається з багатьох компонентів. Задача оркестрації полягає в тому, щоб забезпечити злагоджену взаємодію між цими компонентами, підтримувати їхню ефективність, надійність і масштабованість, мінімізуючи ручне втручання. Оркестрація допомагає системі динамічно адаптуватися до змін у навантаженні, автоматично налаштовувати потрібні сервіси, розгортати нові компоненти та видаляти непотрібні. Це особливо актуально в масштабних додатках, де важливо, щоб усі частини системи працювали безперервно та без конфліктів.

Основна перевага оркестрації – це можливість централізованого контролю за всією системою, що дозволяє ефективно розподіляти ресурси, мінімізуючи простої та забезпечуючи максимальну продуктивність. У контейнерних середовищах, наприклад, таких як Kubernetes, оркестрація дає змогу автоматизувати розгортання, управління та масштабування контейнерів. Це означає, що система може самостійно вирішувати, коли створювати нові екземпляри контейнерів для обробки підвищеного навантаження або видаляти їх, коли вони стають непотрібними. Такий підхід не лише знижує ризики простоїв, а й економить ресурси, адже контейнерні платформи можуть точно регулювати кількість активних інстансів відповідно до навантаження.

Оркестрація також значно спрощує процес оновлень і релізів, що є критичним аспектом у швидкорозвиваючих додатках. Завдяки автоматизованому контролю можна керувати різними версіями компонентів системи, поступово впроваджувати нові оновлення без ризику для стабільності, роблячи «м’яке» розгортання (наприклад, blue-green deployment або canary releases). Це дозволяє протестувати нові версії на окремих користувачах або частині трафіку, забезпечуючи плавний перехід від однієї версії до іншої та мінімізуючи можливі збої.

Важливим аспектом оркестрації є також моніторинг та автоматичне відновлення системи в разі збою. Оркестрація дозволяє швидко реагувати на відмови, відслідковуючи стан кожного компонента і автоматично перезапускаючи його при виявленні проблем. Така функціональність підвищує надійність додатка та допомагає уникати тривалих простоїв, особливо в критичних системах, де висока доступність є ключовою вимогою. Таким чином, оркестрація дозволяє не лише ефективніше використовувати ресурси, а й забезпечити стабільність роботи додатка в умовах постійних змін і зростаючого навантаження.

2.7 Порівняльний аналіз Docker Compose, Docker Swarm та Kubernetes

Docker Compose – це інструмент для спрощення роботи з контейнерами Docker на локальному комп'ютері або сервері. Його основна функція полягає в тому, щоб дозволити запускати кілька контейнерів одночасно як єдиний додаток, описуючи їх конфігурацію у файлі *docker-compose.yml*. У цьому файлі задається, які образи використовувати, які порти потрібно відкрити, змінні середовища, томи для зберігання даних та інші параметри для кожного контейнера. Docker Compose особливо корисний для розробників, коли потрібно швидко підняти цілі стекові середовища для тестування (наприклад, базу даних, бекенд і фронтенд), не налаштовуючи окремо кожен компонент. Однак Docker Compose переважно підходить для локальної розробки та невеликих проєктів, оскільки він не підтримує автоматичне масштабування або розподілені обчислення на рівні кластера.

Docker Swarm – це вбудований оркестраційний інструмент Docker, що дозволяє об’єднувати кілька Docker-хостів у кластер та керувати ними як однією логічною одиницею. У режимі Swarm Docker автоматично розподіляє контейнери по всіх доступних хостах, забезпечуючи балансування навантаження, автоматичне масштабування та відновлення у разі збою. Docker Swarm дозволяє визначати бажану кількість реплік для кожного сервісу, і система сама вирішує, де розміщувати ці репліки для оптимальної роботи. Він має простіший інтерфейс і менше функціональних можливостей, ніж Kubernetes, тому підходить для невеликих або середніх проєктів, яким потрібна базова оркестрація без складних налаштувань. Попри це, Swarm забезпечує високу доступність і дозволяє швидко налаштувати кластер з мінімальними ресурсами.

Kubernetes – це розширена платформа оркестрації контейнерів з відкритим кодом, яка стала стандартом для масштабованих розподілених систем. Kubernetes надає ширші можливості для управління контейнерами

порівняно з Docker Swarm і може використовуватися для розгортання, масштабування та обслуговування великих контейнерних додатків у хмарних або гібридних середовищах. Він підтримує автоматичне масштабування контейнерів, детальну конфігурацію ресурсів, обробку оновлень без простоїв, управління конфігураціями та секретами, а також складні стратегії деплойменту (наприклад, canary та rolling updates). Kubernetes орієнтований на високонавантажені додатки, де важливі надійність, масштабованість та можливість швидкого відновлення. Завдяки своїй архітектурі Kubernetes дозволяє побудувати потужну та гнучку інфраструктуру, в якій компоненти можуть розгортатися і керуватися на різних серверах, автоматично адаптуючись до змін у навантаженні або інфраструктурі.

Kubernetes є відмінним вибором для оркестрації контейнерів, адже наукова робота потребує гнучкості, масштабованості та надійності. На відміну від інших інструментів, Kubernetes автоматизує масштабування і розподіл контейнерів по вузлах кластера, що дозволяє додатку динамічно адаптуватися до змін у навантаженні без втручання з боку розробників. Це забезпечує стабільну роботу в умовах зростаючих вимог, оскільки система автоматично додає або видаляє ресурси залежно від потреби.

Окрім масштабованості, Kubernetes надає високий рівень надійності завдяки вбудованим інструментам для відновлення після збоїв і балансування навантаження. У випадку виходу з ладу одного з компонентів або вузлів, система автоматично перепризначає контейнери на доступні ресурси, гарантуючи безперервну роботу додатку. Крім того, Kubernetes підтримує механізми горизонтального та вертикального автоскейлінгу, що дозволяє динамічно адаптувати ресурси до змін навантаження. Також Kubernetes підтримує гнучкі стратегії деплойменту, що дозволяє оновлювати додаток без простоїв, проводити тестування нових версій або поступово вводити зміни. Ці функції роблять Kubernetes ідеальним рішенням для масштабних і складних систем, що потребують високої доступності та можливості швидкого відновлення.

2.8 Архітектура Kubernetes

Архітектура Kubernetes складається з головного (master) вузла та робочих (worker) вузлів, які разом утворюють кластер. Головний вузол керує всіма аспектами роботи кластеру, забезпечуючи координацію, моніторинг і розподіл завдань. Він містить кілька ключових компонентів, включаючи API Server, який приймає та обробляє запити від користувачів і внутрішніх компонентів; Scheduler, що визначає, на якому вузлі запускати нові контейнери; та Controller Manager, який відповідає за підтримання бажаного стану додатку, наприклад, кількості реплік.

Робочі вузли безпосередньо виконують додатки у вигляді контейнерів, керуючи ними через kubelet – агент, що отримує команди від головного вузла та забезпечує роботу контейнерів відповідно до заданої конфігурації. На кожному робочому вузлі також працює kube-proxy, який контролює мережевий трафік і забезпечує зв’язок як між контейнерами, так і між зовнішнім трафіком та сервісами кластера. Таким чином, робочі вузли виконують функцію обчислювальних ресурсів, на яких розміщуються і масштабуються додатки.

Ще одним важливим аспектом архітектури Kubernetes є система сховища, яка дозволяє керувати збереженням даних для контейнерів. Kubernetes підтримує різні типи сховищ, забезпечуючи стійкість даних навіть при переміщенні контейнерів між вузлами.

Система Persistent Volumes (PV) та Persistent Volume Claims (PVC) дозволяє ефективно розподіляти ресурси сховища між подами та зберігати їх навіть після завершення роботи контейнера. Архітектура Kubernetes оптимізована для масштабованості та високої доступності, що дозволяє розгортати складні додатки з гарантованим балансуванням навантаження і автоматичним відновленням у випадку збоїв.

2.9 Розгортання додатку у Kubernetes

Для розгортання чотирьох мікросервісів у Kubernetes, кожен з яких реалізує певну математичну операцію (додавання, віднімання, множення, ділення), доцільно створити окремі Deployment та Service ресурси для кожного мікросервісу. Це дозволяє Kubernetes ефективно керувати їх масштабуванням, забезпечуючи стабільний доступ до кожної служби у кластері, а також полегшує адміністрування.

Першим етапом є контейнеризація кожного з мікросервісів із застосуванням Docker. Створені контейнери завантажуються в контейнерний реєстр, доступний для Kubernetes (наприклад, Docker Hub або приватний реєстр), що дозволяє кластеру використовувати ці образи під час розгортання.

Далі для кожного мікросервісу необхідно визначити специфікацію Deployment, яка включає опис необхідної кількості реплік, політики оновлення та ресурсних обмежень. Service ресурс забезпечує доступність мікросервісів всередині кластеру або ззовні, використовуючи відповідні типи, такі як ClusterIP, NodePort або LoadBalancer.

Лістинг 2.2 Функція для створення зображення для сервісу:

```
func (Docker) BuildImage() error {
 return sh.Run(
"docker",
"build",
"--build-arg",
"CACHE_DATE=$(date+%Y-%m-%d:%H:%M:%S)",
"-t",
"bogdanserdinov/calculator-example:1.0.0",
".", )
}
```

Лістинг 2.3 Команди для створення та завантаження зображення для сервісу:

```
mage -v docker:buildImage
docker push bogdanserdinov/calculator-example:1.0.0
```

Для кожного мікросервісу доцільно розробити конфігураційні файли типу Deployment та Service. Deployment забезпечує підтримку заданої кількості реплік та автоматичне відновлення у випадку збоїв. Service створює логічну точку доступу до кожного мікросервісу через IP-адреси та порти, забезпечуючи зв’язність всередині кластера.

Аналогічні файли Deployment та Service створюються для інших мікросервісів.

Лістинг 2.4 Deployment файл для сервісу додавання:

```
apiVersion: apps/v1
kind: Deployment
metadata:
   name: addition
   namespace: addition
spec:
   replicas: 1
   selector:
     matchLabels:
     app: addition
   template:
     metadata:
       labels:
         app: addition
     spec:
```

```
      containers:
       - name: addition
       image: bogdanserdinov/calculator-example:1.0.0
       command: [ "/bin/sh", "-c", "addition" ]
       ports:
        - containerPort: 9090
       envFrom:
        - configMapRef:
          name: addition-config
       livenessProbe:
        httpGet:
        path: /heathz
        port: 8081
        initialDelaySeconds: 15
        periodSeconds: 10
```

Лістинг 2.5 Service файл для сервісу додавання:

```
apiVersion: v1
kind: Service
metadata:
   name: addition
   namespace: addition
spec:
   ports:
     - name: http
      protocol: TCP
     port: 9090
     targetPort: 9090
   selector:
```

app: addition

Для розгортання чотирьох мікросервісів у Kubernetes, кожен з яких реалізує певну математичну операцію (додавання, віднімання, множення, ділення), доцільно створити окремі Deployment та Service ресурси для кожного мікросервісу. Це дозволяє Kubernetes ефективно керувати їх масштабуванням, забезпечуючи стабільний доступ до кожної служби у кластері, а також полегшує адміністрування. Такий підхід гарантує ефективне управління ресурсами, високу доступність та легкість у підтримці мікросервісної архітектури.

Лістинг 2.6 Команди для застосування змін у кластері:

kubectl apply -f addition-deployment.yaml
kubectl apply -f addition-service.yaml
kubectl apply -f subtraction-deployment.yaml
kubectl apply -f subtraction-service.yaml
kubectl apply -f multiplication-deployment.yaml
kubectl apply -f multiplication-service.yaml
kubectl apply -f division-deployment.yaml
kubectl apply -f division-service.yaml

Після застосування цих команд буде створений Kubernetes кластер на робочому вузлі, готовий до обслуговування мікросервісів. На цьому етапі важливо перевірити статус ресурсів, щоб переконатися, що всі сервіси та поди працюють коректно.

Такий підхід гарантує ефективне управління ресурсами, високу доступність та легкість у підтримці мікросервісної архітектури.

3 ДОСЛІДЖЕННЯ ВПЛИВУ МЕТОДІВ КОНТЕЙНЕРІЗАЦІЇ НА МАСШТАБУВАННЯ МІКРОСЕРВІСІВ

3.1 Розгортання кластеру у хмарному середовищі

Розгортання додатків у хмарному середовищі є критично важливим для забезпечення гнучкості та масштабованості інфраструктури. Хмарні платформи, такі як Amazon Web Services (AWS), Google Cloud Platform (GCP) та Microsoft Azure, надають необхідні інструменти для ефективного розміщення додатків і сервісів. Одним із основних аспектів хмарного розгортання є налаштування Virtual Private Cloud (VPC), що дозволяє створювати ізольовані мережі для безпечної комунікації між різними компонентами додатка. Це важливо для забезпечення належного рівня безпеки та керованості ресурсами, оскільки дозволяє чітко визначити, які сервіси можуть взаємодіяти один з одним.

Лістинг 3.1 Конфігурація у AWS CloudFormation для налаштування VPC:

```
Resources:
  MyVPC:
    Type: AWS::EC2::VPC
    Properties:
      CidrBlock: 10.0.0.0/16
      EnableDnsSupport: true
      EnableDnsHostnames: true
      Tags:
        - Key: Name
          Value: MyVPC
```

```
MySubnet:
  Type: AWS::EC2::Subnet
  Properties:
    VpcId: !Ref MyVPC
    CidrBlock: 10.0.1.0/24
    AvailabilityZone: us-east-1a
    Tags:
      - Key: Name
        Value: MySubnet
```

Код, представлений у лістингу 3.1 створює VPC з певним CIDR-блоком та додає підмережу в одну з доступних зон AWS. За допомогою таких налаштувань можна ізолювати інфраструктуру і визначити правила доступу для різних сервісів у межах цієї мережі.

Наступним важливим етапом є налаштування правил маршрутизації та брандмауерів. Для цього у VPC можна налаштувати Security Groups та Network Access Control Lists (NACLs), щоб обмежити доступ до ресурсів лише для авторизованих користувачів і сервісів.

Лістинг 3.2 Налаштування Security Group та Network Access Control Lists:

```
MySecurityGroup:
  Type: AWS::EC2::SecurityGroup
  Properties:
    GroupDescription: Allow HTTP and SSH access
    SecurityGroupIngress:
      - CidrIp: 0.0.0.0/0
        IpProtocol: tcp
        FromPort: '22'
        ToPort: '22'
```

```
- CidrIp: 0.0.0.0/0
  IpProtocol: tcp
  FromPort: '80'
  ToPort: '80'
```

Фрагмент, представлений у лістингу 3.2, дозволяє доступ до інстансів через порти SSH (22) і HTTP (80) для будь-якої IP-адреси. Такі налаштування забезпечують доступ до ресурсів тільки через визначені порти, підвищуючи безпеку.

Після налаштування мережевої інфраструктури можна приступити до розгортання сервісів, таких як Kubernetes для оркестрації контейнерів. В цьому випадку налаштування Kubernetes в хмарному середовищі зазвичай включає створення кластера Kubernetes, який працюватиме в межах VPC, що забезпечить належну ізоляцію та масштабованість додатків. Використання таких інструментів, як Helm, дозволяє автоматизувати процес розгортання додатків, надаючи простоту в управлінні складними сервісами.

Розгортання в хмарі дозволяє значно підвищити гнучкість інфраструктури, оскільки ви маєте можливість швидко масштабувати ресурси в залежності від потреб додатка. Наприклад, використовуючи автоматичне масштабування Kubernetes (HPA), можна адаптувати кількість реплік контейнерів до поточного навантаження, що забезпечує оптимальне використання ресурсів.

Нарешті, хмарне середовище надає величезні можливості для моніторингу та управління інфраструктурою. З інструментами на, наприклад, AWS CloudWatch, можна створювати метрики, алерти і візуалізації для відстеження стану системи в реальному часі. Це дозволяє швидко реагувати на збої та аномалії в роботі додатків.

Таким чином, розгортання в хмарному середовищі забезпечує не лише масштабованість і безпеку, а й дозволяє ефективно керувати ресурсами, знижуючи витрати та забезпечуючи високу доступність додатків. Це також

спрощує автоматизацію процесів, що прискорює оновлення та підвищує надійність системи.

3.2 Аналіз типів навантаження для тестування системи

Під час тестування продуктивності та стійкості системи важливо ретельно визначити типи навантаження, які будуть використовуватись для перевірки її поведінки в різних умовах. Такий підхід дозволяє не тільки оцінити, як система реагує на різні рівні запитів, але й зрозуміти, як вона поводиться під тиском та як її компоненти працюють разом під час різноманітних сценаріїв. Оскільки сучасні системи повинні забезпечувати стабільну роботу при різних навантаженнях, тестування за допомогою кількох типів навантаження є критично важливим етапом у процесі їх розробки та експлуатації.

Постійне навантаження є основним методом для тестування стабільності системи на довготривалих проміжках часу. При такому тестуванні система зазнає певного постійного потоку запитів, що дозволяє перевірити її здатність підтримувати стабільну роботу протягом тривалого періоду. Це тип навантаження найбільш характерний для сценаріїв, де система повинна витримувати стабільний потік користувацьких запитів, як це буває, наприклад, при постійній активності користувачів або надання різних послуг через інтерфейси API. Такий тест допомагає виявити потенційні проблеми, які можуть виникнути при тривалому навантаженні, наприклад, вичерпання ресурсів, зниження продуктивності або накопичення помилок, які не були б виявлені при менш інтенсивному тестуванні.

Стресове навантаження — це метод тестування, при якому навантаження поступово збільшується до тих пір, поки система не досягне своїх меж можливостей. Метою цього тесту є виявлення точок відмови системи, коли її компоненти не можуть впоратися з високими запитами і

починають працювати неналежним чином або взагалі не відповідають. Такий тип тестування дозволяє зрозуміти, на яких етапах система починає збільшувати латентність, втрачає частину запитів або навіть виходить із ладу. Стресові тести є особливо корисними для виявлення проблем із масштабованістю, помилками в розподілі ресурсів або виявленням критичних уразливих точок, які можуть не бути очевидними під час звичайної експлуатації, але можуть стати причиною серйозних збоїв у випадку пікового навантаження.

Тест пікового навантаження має на меті перевірити, як система реагує на короткочасне різке збільшення навантаження. Наприклад, це може бути різке зростання числа запитів під час проведення рекламних акцій, розпродажів, запуску нових функцій або інших подій, що викликають високий інтерес користувачів. Важливо, щоб система була готова до подібних пікових навантажень і могла без втрат продуктивності обробити велику кількість запитів за короткий період. Тест пікового навантаження зазвичай триватиме лише кілька хвилин або годин, але при цьому виявляються критичні уразливі місця, наприклад, у системах балансування навантаження або обробці запитів на серверному рівні, що можуть привести до затримок, відмов або навіть повних збоїв.

Ці типи навантаження допомагають не тільки оцінити здатність системи до роботи під різними рівнями навантаження, а й дозволяють розробникам і інженерам точно визначити слабкі місця, де можуть виникнути проблеми, і підготувати систему до ситуацій, з якими вона може стикатися в реальних умовах. Тестування під різними типами навантаження дозволяє зменшити ризик збоїв у реальному середовищі, покращити користувацький досвід та зменшити ймовірність фінансових втрат, пов'язаних із несправностями або втратами даних.

Для більш детальної оцінки системи важливо поєднувати різні типи навантаження в процесі тестування. Це дозволяє створити повну картину її продуктивності та стійкості до різних сценаріїв, що значно підвищує

надійність системи і робить її готовою до обробки запитів у будь-яких умовах. В результаті, таке тестування стає важливим етапом у процесі оптимізації програмного забезпечення та інфраструктури для масштабування і забезпечення високої доступності та продуктивності.

3.3 Тестування застосунку методом постійного навантаження

Для нашої системи був обраний метод постійного навантаження, оскільки він найкраще відповідає вимогам щодо перевірки стабільності та надійності сервісів протягом тривалого часу при реальних умовах експлуатації. Однією з головних задач нашої інфраструктури є забезпечення безперервної роботи та стабільності, навіть при високому рівні запитів від користувачів або інших систем. Тестування за допомогою постійного навантаження дозволяє моделювати умови реальної експлуатації, коли система повинна обробляти запити без значних коливань у часі або ресурсах. Крім того, використання Kubernetes дозволяє забезпечити автоматичне масштабування, балансування навантаження та швидке відновлення після збоїв, що є критичним для безперебійної роботи системи.

Оскільки наша система розгорнута в Kubernetes, вона має здатність автоматично адаптуватися до змін у навантаженні завдяки вбудованим механізмам масштабування подів. Kubernetes забезпечує високу доступність за рахунок використання реплік і стратегій розподілу навантаження між різними вузлами кластера, що дає змогу ефективно обробляти великі обсяги запитів навіть під час пікових навантажень. Це дозволяє перевірити не лише окремі компоненти системи, а й їх взаємодію під час тривалої роботи в умовах реального навантаження.

Завдяки Kubernetes, ми можемо автоматично збільшувати кількість реплік наших сервісів в разі високого навантаження, що дозволяє підтримувати стабільність навіть при зростанні кількості запитів. Кластери

Kubernetes також дозволяють забезпечити контроль за використанням ресурсів, що є важливим аспектом для запобігання перевантаження окремих компонентів. Тестування з постійним навантаженням в таких умовах дозволяє нам не лише перевірити здатність системи до стабільної роботи, а й оцінити ефективність стратегій масштабування та балансування навантаження.

Також, за допомогою Kubernetes, ми можемо реалізувати механізми автоматичного відновлення після збоїв, що є важливою складовою тестування постійним навантаженням. У разі виявлення несправностей на одному з компонентів системи, Kubernetes автоматично перенаправить трафік на здорові репліки, що дозволяє мінімізувати вплив помилок на загальну доступність сервісів. Це дає змогу не лише виявити слабкі місця в системі, але й перевірити, як вона реагує на збої, відновлення та автоматичне масштабування.

Таким чином, тестування постійним навантаженням в умовах Kubernetes допомагає забезпечити стабільність та надійність нашої інфраструктури, дозволяючи ефективно виявляти потенційні проблеми до того, як вони вплинуть на кінцевих користувачів. Завдяки можливостям автоматичного масштабування та балансування навантаження в Kubernetes, ми можемо гарантовано підтримувати високу доступність і стабільність системи, що є критичним для будь-якого виробничого середовища.

Лістинг 3.3 Функція симуляції навантаження для перевірки масштабованості та відмовостійкості:

```
// Usage mage LoadTest http://localhost:8080 1 100
func LoadTest(gatewayURL string, interval, duration int) error {
    client:= http.Client{
      Timeout: 10 * time.Second,
    }
    done := make(chan struct{})
```

```
    ticker := time.NewTicker(time.Duration(interval) * time.Millisecond)
    defer ticker.Stop()

    go func() {
      time.Sleep(time.Duration(duration) * time.Second)
      done <- struct{}{}
    }()

    for {
      select {
      case <-ticker.C:
        doRequest(client, gatewayURL)
      case <-done:
        return nil
      }
    }
}

const routePattern = "%s/api/%s/v1/load"

func doRequest(client http.Client, baseURL string) {
    for _, service := range services {
      route := fmt.Sprintf(routePattern, baseURL, service.name)

      resp, err := client.Get(route)
      if err != nil {
        log.Printf("Error to call route %v, err = %v", route, err)
        continue
      }
```

```
        // 404 ignored because some services are isolated for http gateway,
        // and used only for dwn
        if resp.StatusCode == http.StatusOK || resp.StatusCode ==
    http.StatusNotFound {
            log.Println("success", route, service.name, resp.StatusCode)
        } else {
            log.Println("error", route, service.name, resp.StatusCode)
        }
    }
}
```

Функція, описана у лістингу 3.3 реалізує механізм навантажувального тестування для визначеного URL-шлюзу (gateway), виконуючи періодичні HTTP запити до певних сервісів протягом визначеного часу. Вона ініціалізує HTTP клієнт з таймаутом у 10 секунд для кожного запиту, а також використовує time.Ticker для періодичного надсилання запитів до зазначеного адреси через задані інтервали.

Функція приймає три параметри: gatewayURL — базовий URL для шлюзу, interval — інтервал між запитами в мілісекундах, та duration — тривалість тесту в секундах. Тести виконуються до завершення заданої тривалості, після чого процес тестування завершується.

Основний механізм функціонування полягає в тому, що за кожен інтервал часу відправляється запит до кожного сервісу, що вказаний у масиві services. Для кожного сервісу складається URL-шлях, і виконується GET запит до цього шляху. Якщо сервер відповідає кодом 200 (OK) або 404 (Not Found), виводиться відповідне повідомлення, а в разі інших статусів повідомляється про помилку. 404 статуси ігноруються, оскільки деякі сервіси можуть бути доступні лише для внутрішніх запитів через інші механізми.

Цей підхід є корисним для тестування відмовостійкості та масштабованості системи, оскільки дозволяє перевірити, як система реагує

на навантаження у вигляді повторюваних запитів, зокрема оцінити час відгуку та стабільність сервісів під час тривалого навантаження. Виконання таких тестів дає змогу виявляти вузькі місця в системі, а також визначати необхідні налаштування для забезпечення високої доступності та стабільності сервісів.

3.4 Використання системи розподілу навантаження для забезпечення стійкості системи

Використання load balancer є критично важливим аспектом для забезпечення стійкості, масштабованості та високої доступності розподілених систем. Для досягнення цих цілей в нашій інфраструктурі було обрано Traefik як інструмент для балансування навантаження між сервісами в середовищі Kubernetes. Traefik було обрано через його здатність до інтеграції з оркестраторами контейнерів, зокрема Kubernetes, що дозволяє автоматизувати налаштування маршрутизації та забезпечує гнучкість при розподілі запитів між різними компонентами системи.

Однією з основних характеристик Traefik є підтримка автоматичного виявлення сервісів і здатність до динамічного конфігурування маршрутизації запитів без необхідності в ручному втручанні. Така функціональність особливо важлива в умовах швидко змінюваного середовища, де кількість контейнерів і сервісів може варіюватися залежно від поточних потреб. Автоматичне налаштування маршрутизації забезпечує безперебійне функціонування системи навіть при значних змінах у її архітектурі.

Traefik підтримує різноманітні стратегії балансування навантаження, такі як round robin, least connections, а також інші методи, що дозволяють оптимізувати розподіл запитів залежно від конкретних характеристик навантаження на сервери. Наприклад, стратегія round robin дозволяє рівномірно розподіляти запити між усіма доступними екземплярами сервісів,

а стратегія least connections надає пріоритет сервісам з найменшою кількістю активних з'єднань, що може бути корисно для запобігання перевантаженню окремих компонентів.

Одним із ключових аспектів використання Traefik є його інтеграція з Kubernetes, що дозволяє зручно автоматизувати конфігурацію маршрутизації через ресурси Ingress. Наприклад, за допомогою Ingress ми можемо налаштувати маршрутизацію запитів від клієнтів до конкретних сервісів за допомогою простого YAML-файлу, що описує правила маршрутизації. Це дозволяє ефективно управляти доступом до сервісів і зменшити складність налаштування мережевої інфраструктури.

Інтеграція Traefik з Kubernetes також дозволяє легко масштабувати сервіси в залежності від навантаження. При необхідності ми можемо збільшити кількість реплік сервісів, і Traefik автоматично буде перенаправляти запити між новими екземплярами, забезпечуючи рівномірний розподіл навантаження. Це дозволяє підтримувати стабільну роботу системи навіть при збільшенні кількості користувачів або при різкому зростанні навантаження.

Для забезпечення надійності і стійкості системи, Traefik також підтримує механізми моніторингу та здоров'я сервісів. Це дозволяє системі автоматично виявляти неполадки в окремих компонентах і оперативно відключати несправні екземпляри від маршрутизації, перенаправляючи трафік на здорові сервіси. Така функціональність допомагає значно знизити ризики відмов системи та підвищує її загальну стійкість до можливих збоїв.

Лістинг 3.4 Конфігурація Traefik:

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: traefik
  labels:
```

```
    app: traefik
spec:
  replicas: 1
  selector:
    matchLabels:
      app: traefik
  template:
    metadata:
      labels:
        app: traefik
    spec:
      containers:
        - name: traefik
          image: traefik:v2.10
          args:
            - "--api.insecure=true
            - "--providers.kubernetescrd=true
            - "--entrypoints.web.address=:80
            - "--entrypoints.websecure.address=:443
          ports:
            - name: web
              containerPort: 80
            - name: websecure
              containerPort: 443
          volumeMounts:
            - name: traefik-tls
              mountPath: /certs
              readOnly: true
      volumes:
        - name: traefik-tls
```

```
    secret:
      secretName: traefik-cert-secret
```

YAML файл, конфігурації визначає розгортання для Traefik у Kubernetes, який забезпечує розподіл навантаження та керування трафіком між сервісами. Він створює одну репліку контейнера Traefik за допомогою образу traefik:v2.10. Конфігурація включає налаштування для HTTP та HTTPS вхідних точок, а також підключення до Kubernetes CRD для інтеграції з кластером. Включено API для моніторингу та використання секретів для TLS сертифікатів, що дозволяє забезпечити безпечне з'єднання через HTTPS.

Лістинг 3.5 Конфігурація ingress для маршрутизації трафіку:

```
apiVersion: networking.k8s.io/v1
kind: Ingress
metadata:
  name: example-ingress
  annotations:
    traefik.ingress.kubernetes.io/router.entrypoints: "web,websecure"
spec:
  rules:
    - host: my-app.example.com
      http:
        paths:
          - path: /
            pathType: Prefix
            backend:
              addition:
                name: additional
                port:
                  number: 80
```

```
tls:
 - hosts:
    - my-app.example.com
   secretName: additional -tls-secret
```

3.5 Моніторинг сервісів у Kubernetes

Моніторинг інфраструктури є критично важливим аспектом для забезпечення стабільної та безперебійної роботи будь-якої розподіленої системи, особливо в середовищах, що використовують контейнеризацію та оркестрацію через Kubernetes. Одним із основних завдань моніторингу є надання повної картини про стан компонентів системи в реальному часі, що дозволяє швидко виявляти та усувати потенційні проблеми. У контексті Kubernetes для реалізації моніторингу широко використовуються інструменти, які забезпечують збір та візуалізацію метрик, а також централізоване логування.

Одним із найпоширеніших інструментів моніторингу є Prometheus, який спеціалізується на зборі метрик з різних джерел, зокрема з контейнерів, сервісів, та кластеру Kubernetes. Prometheus здатний ефективно обробляти велику кількість даних і підтримує механізм збору метрик за допомогою pull-запитів, що дозволяє отримувати інформацію про стан системи з усіх контейнерів, що працюють у кластері. Зібрані метрики можуть бути використані для побудови графіків і створення алертів, що допомагає відстежувати основні показники роботи системи, такі як використання ресурсів (CPU, пам'ять, диски), час відгуку сервісів, доступність компонентів системи, а також рівень помилок.

Лістинг 3.6 Конфігурація Prometheus:

```
apiVersion: v1
```

```
kind: ConfigMap
metadata:
  name: prometheus-config
  labels:
    app: prometheus
data:
  prometheus.yml: |
    global:
      scrape_interval: 15s
    scrape_configs:
      - job_name: 'kubernetes-apiservers'
        kubernetes_sd_configs:
          - role: endpoint
        relabel_configs:
          - source_labels: [__meta_kubernetes_addition]
            target_label: job
```

Для візуалізації отриманих даних зазвичай використовують Grafana, яка інтегрується з Prometheus і дозволяє створювати інтерактивні дашборди. Це дозволяє командам DevOps, а також операційним командам візуалізувати важливі метрики в реальному часі, а також аналізувати історичні дані для визначення трендів та виявлення можливих аномалій. Наприклад, можна побудувати графіки, що показують використання пам'яті, мережевий трафік або статус підключених сервісів.

Інтеграція інструментів моніторингу з Kubernetes дозволяє не лише збирати дані про стан кластеру, але й спрощує процес автоматичного виявлення проблем з контейнерами, підами, деплойментами та іншими компонентами. Наприклад, інструменти на кшталт kube-state-metrics надають детальні метрики про статус ресурсів Kubernetes, що дає можливість відстежувати не тільки фізичні ресурси, але й більш високий рівень

абстракції, наприклад, стан деплойментів та реплік. Ці метрики можуть бути використані для створення тривожних повідомлень, що зможуть сповістити про будь-які неполадки або відхилення від заданих норм. Використання таких інструментів, як Prometheus і Grafana, дозволяє ефективно візуалізувати ці метрики та налаштовувати дашборди для відстеження ключових показників у реальному часі. Це допомагає оперативно реагувати на потенційні проблеми та забезпечувати стабільну роботу системи.

Лістинг 3.7 Налаштування Grafana для роботи з Prometheus:

```
apiVersion: v1
kind: ConfigMap
metadata:
  name: grafana-config
  labels:
    app: grafana
data:
  grafana.ini: |
    [auth]
    disable_login_form = false
    [server]
    root_url = %(protocol)s://%(domain)s/
    [users]
    default_theme = dark
  provisioning/dashboards/default.json: |
    {
      "apiVersion": 1,
      "providers": [
        {
          "name": "default",
          "orgId": 1,
```

```
        "folder": "",
        "type": "file",
        "disableDeletion": false,
        "updateIntervalSeconds": 5,
        "options": {
          "path": "/etc/grafana/provisioning/dashboards"
        }
      }
    ]
  }
```

Лістинг 3.8 Конфігурація Grafana Dashboard:

```
apiVersion: v1
kind: ConfigMap
metadata:
  name: grafana-config
  labels:
    app: grafana
data:
  grafana.ini: |
    [auth]
    disable_login_form = false
    [server]
    root_url = %(protocol)s://%(domain)s/
    [users]
    default_theme = dark
  provisioning/dashboards/default.json: |
    {
      "apiVersion": 1,
```

```
"providers": [
 {
  "name": "default",
  "orgId": 1,
  "folder": "",
  "type": "file",
  "disableDeletion": false,
  "updateIntervalSeconds": 5,
  "options": {
   "path": "/etc/grafana/provisioning/dashboards"
  }
 }
]
}
```

Іншою важливою складовою моніторингу є логування, яке є незамінним для виявлення проблем на рівні додатків. Централізовані системи логування, такі як ELK Stack (Elasticsearch, Logstash, Kibana) або EFK Stack (Elasticsearch, Fluentd, Kibana), дозволяють збирати, обробляти та аналізувати логи з усіх контейнерів і сервісів у кластері Kubernetes. Логування надає додаткову інформацію про те, як працюють окремі компоненти, і може бути корисним для відстеження помилок, а також для глибшого аналізу причин несправностей, таких як збої в обробці запитів або проблеми з підключенням до баз даних.

Моніторинг і логування є основними інструментами, які допомагають не тільки зберігати здоров'я інфраструктури, а й покращувати продуктивність системи, оптимізувати використання ресурсів і своєчасно реагувати на непередбачені ситуації. Вони допомагають забезпечити прозорість роботи системи, надаючи детальну інформацію про її стан та дозволяючи оперативно реагувати на зміни в її поведінці. Це дозволяє не тільки

підвищити надійність і ефективність роботи інфраструктури, але й створює умови для постійного вдосконалення процесів та досягнення більш високих показників надійності і продуктивності.

Лістинг 3.9 Налаштування Logstash:

```
apiVersion: v1
kind: ConfigMap
metadata:
  name: logstash-config
data:
  logstash.conf: |
    input {
      beats {
        port => 5044
      }
    }
    filter {
      grok {
        match => { "message" => "%{COMBINEDAPACHELOG}" }
      }
    }
    output {
      elasticsearch {
        hosts => ["http://elasticsearch:9200"]
        index => "logstash-%{+YYYY.MM.dd}"
      }
    }
```

Ця конфігурація, що включає в себе Elasticsearch, Logstash і Kibana (стек ELK), дозволяє ефективно моніторити стан сервісів та інфраструктури,

надаючи потужні інструменти для збору, обробки та візуалізації логів. Elasticsearch виступає як центральне сховище для всіх логів, забезпечуючи швидкий доступ до даних та можливість їх індексації, що дозволяє ефективно шукати та аналізувати великі обсяги інформації. Logstash здійснює прийом і обробку логів, виконуючи необхідні фільтрації та трансформації перед відправкою в Elasticsearch. Це дозволяє створювати детальну картину стану системи та налаштовувати процес збору даних відповідно до специфіки інфраструктури. Kibana забезпечує зручний інтерфейс для візуалізації і аналізу даних, даючи змогу створювати кастомізовані дашборди для моніторингу важливих метрик і подій. Така інтеграція дозволяє здійснювати проактивний моніторинг, виявляти потенційні проблеми до того, як вони стануть критичними, а також оптимізувати процеси на основі отриманої аналітики. Використання цього підходу в інфраструктурі Kubernetes гарантує гнучкість і масштабованість, що є важливими факторами для підтримки стабільної роботи та швидкого реагування на зміни в системі.

3.6 Додавання метрик у сервіси

Для реалізації моніторингу розроблено функцію на мові Go, яка використовує бібліотеку prometheus/client_golang. Створюються дві основні метрики: http_requests_total, яка рахує загальну кількість запитів із зазначенням статусів відповідей, та http_request_duration_seconds, що відображає тривалість обробки запитів у вигляді гістограми. Відлік часу здійснюється за допомогою таймера, що дозволяє обчислити точну тривалість обробки кожного запиту, забезпечуючи високу точність отриманих даних. Кожен запит на сервер завершується реєстрацією відповідного статусу (200 OK або 500 Internal Server Error), що дозволяє

отримати розподіл відповідей за статусами, забезпечуючи моніторинг як успішних, так і помилкових запитів.

Лістинг 3.10 Створення метрик на бекенді:

```
package main

import (
"log"
"net/http"
"time"

"github.com/prometheus/client_golang/prometheus"
"github.com/prometheus/client_golang/prometheus/promhttp"
)

var httpRequestsTotal = prometheus.NewCounterVec(
prometheus.CounterOpts{
        Name: "http_requests_total",
        Help: "Загальна кількість HTTP-запитів, розподілених за статусами.",
},
[]string{"status"},
)

var httpRequestDuration = prometheus.NewHistogramVec(
prometheus.HistogramOpts{
        Name:    "http_request_duration_seconds",
        Help:    "Гістограма тривалості обробки HTTP-запитів.",
        Buckets: prometheus.DefBuckets,
},
```

```
[]string{},
)

func init() {
prometheus.MustRegister(httpRequestsTotal)
prometheus.MustRegister(httpRequestDuration)
}
```

Код забезпечує обробку двох маршрутів: один повертає успішний статус 200 OK, а інший імітує внутрішню помилку сервера зі статусом 500 Internal Server Error. Такий підхід дозволяє моделювати реальні сценарії роботи сервісу, де частина запитів може завершуватись із помилками.

Дані метрики експортуються на маршрут /metrics, де їх може зчитувати Prometheus для подальшого аналізу.

Лістинг 3.11 Конфігурація для збору метрик у Prometheus:

```
scrape_configs:
  - job_name: "addition"
    static_configs:
      - targets: ["localhost:8080"]
```

Завдяки цьому підходу забезпечується доступ до ключових метрик, таких як кількість запитів і їх розподіл за статусами, а також тривалість обробки запитів.

Це дозволяє оцінити загальну продуктивність сервісу, виявити потенційні проблеми у його роботі та своєчасно їх усунути. Окрім цього, детальне логування та моніторинг дозволяють аналізувати поведінку користувачів і ефективно планувати оптимізацію системи.

3.7 Горизонтальне та вертикальне масштабування

Горизонтальне та вертикальне масштабування є ключовими аспектами забезпечення високої доступності та продуктивності системи, що працює в Kubernetes із використанням Traefik як балансувальника навантаження. Ці два підходи дозволяють динамічно адаптувати ресурси відповідно до навантаження, забезпечуючи безперебійну роботу навіть за умов пікового трафіку або різких змін інтенсивності запитів.

Вертикальне масштабування передбачає збільшення або зменшення ресурсів (пам’яті, процесорних ядер), доступних для кожного окремого контейнера або поду. У випадку з Kubernetes це може бути реалізовано через оновлення ресурсних запитів і лімітів у конфігурації подів. Наприклад, якщо певний сервіс починає перевантажуватися через складну обробку запитів або високий обсяг даних, можна збільшити його ресурси за допомогою налаштувань resources.requests та resources.limits у файлі YAML. Це дозволить сервісу отримати більше процесорного часу або пам’яті для обробки запитів, що зменшить затримки та запобіжить падінню продуктивності. Однак вертикальне масштабування має обмеження, оскільки ресурси одного вузла не безмежні.

Горизонтальне масштабування є більш гнучким і ефективним підходом, оскільки воно передбачає додавання нових реплік подів для розподілу навантаження.

У Kubernetes це реалізується через механізм Horizontal Pod Autoscaler (HPA), який автоматично збільшує або зменшує кількість реплік залежно від метрик, таких як завантаження CPU або кількість активних запитів. Наприклад, якщо сервіс, що обробляє запити на /api/load, стикається зі значним зростанням кількості запитів, HPA може збільшити кількість реплік з трьох до десяти, щоб кожен под отримав менше навантаження і міг швидше обробляти запити.

Лістинг 3.12 Налаштування Horizontal Pod Autoscaler:

```
apiVersion: autoscaling/v2
kind: HorizontalPodAutoscaler
metadata:
  name: load-test-hpa
spec:
  scaleTargetRef:
    apiVersion: apps/v1
    kind: Deployment
    name: load-test-app  # Назва Deployment, який будемо масштабувати
  minReplicas: 3  # Мінімальна кількість реплік
  maxReplicas: 10  # Максимальна кількість реплік
  metrics:
    - type: Resource
      resource:
        name: cpu
        target:
          type: Utilization
          averageUtilization: 70  # Цільове завантаження CPU у відсотках
```

Traefik як балансувальник навантаження відіграє ключову роль у горизонтальному масштабуванні, забезпечуючи рівномірний розподіл трафіку між усіма репліками. При додаванні нових реплік він автоматично оновлює маршрутизацію, щоб гарантувати, що всі нові поди беруть участь у прийманні запитів. Наприклад, якщо у піковий момент навантаження кількість подів збільшується до десяти, Traefik розподіляє трафік рівномірно між ними, що запобігає перевантаженню окремих екземплярів і забезпечує стабільну роботу всієї системи.

Практичним прикладом горизонтального масштабування може бути інтернет-магазин, який у звичайні дні має три репліки сервісу для обробки запитів, але під час розпродажів або рекламних кампаній збільшує кількість реплік до десяти або більше. Водночас вертикальне масштабування можна застосувати до сервісу обробки великих обсягів даних, коли потрібно тимчасово збільшити обсяг оперативної пам’яті для аналітичних запитів.

Таким чином, обидва підходи масштабування в Kubernetes із Traefik забезпечують стійкість до навантаження, гнучкість у використанні ресурсів і надійність у будь-яких умовах. Це дозволяє системі залишатися продуктивною та стабільною навіть під час значних коливань навантаження, забезпечуючи якісний користувацький досвід і високу доступність сервісів.

3.8 Огляд роботи алгоритму горизонтального масштабування

Горизонтальне масштабування в Kubernetes є одним із ключових механізмів, який дозволяє системі динамічно адаптуватися до змін навантаження, забезпечуючи стійкість і високу продуктивність. У системах, де використовується балансувальник навантаження Traefik, цей процес забезпечує плавне збільшення або зменшення кількості подів залежно від поточного стану навантаження на додаток.

Процес горизонтального масштабування починається зі збору та моніторингу метрик, таких як використання процесора або пам’яті кожним подом. Kubernetes використовує спеціальні служби, наприклад, Metrics Server або Prometheus, які постійно відстежують стан ресурсів і передають ці дані до механізму Horizontal Pod Autoscaler (HPA). HPA аналізує ці метрики та порівнює їх із заданими цільовими значеннями, щоб оцінити, чи відповідає поточна кількість подів необхідному рівню навантаження.

Якщо середнє значення використання ресурсів перевищує встановлений поріг, наприклад, 70% завантаження CPU, HPA приймає

рішення про збільшення кількості реплік. Kubernetes автоматично створює додаткові поди та додає їх до кластеру, дозволяючи рівномірно розподілити навантаження між усіма активними екземплярами. Навпаки, коли навантаження знижується, HPA може зменшити кількість реплік до мінімально необхідної для підтримання базової доступності додатка.

Traefik як балансувальник відіграє важливу роль у цьому процесі, оскільки він автоматично розподіляє вхідні запити між усіма доступними подами. Це забезпечує рівномірний розподіл навантаження та запобігає перевантаженню окремих реплік. Наприклад, якщо система починає отримувати збільшену кількість запитів, Traefik направляє їх до нових подів, які були створені HPA, що дозволяє підтримувати стабільну швидкість обробки запитів і запобігати збоям.

Практичний приклад такого масштабування можна уявити в сценарії, коли веб-додаток обробляє тисячі запитів на секунду. При раптовому збільшенні навантаження, коли кожен под починає споживати більше ресурсів, Kubernetes автоматично створює нові поди, щоб уникнути зниження продуктивності. Після стабілізації трафіку, коли навантаження зменшується, надлишкові поди поступово видаляються, що дозволяє ефективно використовувати обчислювальні ресурси.

Такий підхід до масштабування гарантує, що система залишається гнучкою, здатною до швидкої адаптації та економічного використання ресурсів, забезпечуючи безперебійну роботу навіть у випадках пікових навантажень або несподіваного зростання запитів. Завдяки автоматизації процесу створення та видалення подів система може реагувати на зміни навантаження в реальному часі, мінімізуючи ризик збоїв і забезпечуючи стабільність. Це особливо важливо для критичних сервісів, де кожен запит має бути оброблений максимально швидко та якісно. Крім того, оптимізація використання ресурсів дозволяє значно зменшити експлуатаційні витрати, оскільки додаткові обчислювальні потужності залучаються лише тоді, коли це дійсно необхідно. У результаті така архітектура підтримує високу

доступність і продуктивність додатка, навіть коли навантаження коливається, створюючи баланс між ефективністю та надійністю.

3.9 Тестування адаптивності системи

Для проведення комплексного аналізу стійкості та продуктивності системи було організовано масштабне навантажувальне тестування. Його метою було виявити межі продуктивності системи, оцінити здатність витримувати тривалий потік запитів та дослідити реакцію на пікові навантаження. Тестування проводилося протягом 3600 секунд (1 година), імітуючи надсилання 30 запитів на секунду. Загальна кількість запитів досягла 108 000. Такий підхід дозволив оцінити довготривалу продуктивність і стабільність.

Для забезпечення безперервного надсилання запитів використовувався високопродуктивна CLI утиліта яка надсилала запити з 10 секундним інтервалом. Всі запити надсилалися на кінцеву точку /api/load із періодичністю кожні 33 мс. Для обмеження тривалості тесту використовувався механізм каналу done, який дозволяв автоматично завершити тест через заданий проміжок часу. Код тестування реєстрував кожну відповідь від сервера, включаючи успішні відповіді та помилки.

За час тесту було здійснено 108 000 запитів, з яких 97 200 (90%) завершилися успішно зі статусом 200 OK. Водночас 10 800 запитів (10%) завершилися помилками, що вказує на обмеження в обробці пікового навантаження. Середня затримка обробки запитів становила 180 мс, а максимальна досягала 500 мс.

Лістинг 3.13 Метрики, зібрані Prometheus:

```
http_requests_total{status="200"} 97764
http_requests_total{status="500"} 4323
```

http_requests_total{status="503"} 3521

http_requests_total{status="502"} 2392

http_request_duration_seconds_bucket{le="0.1"} 76220

http_request_duration_seconds_bucket{le="0.3"} 111735

http_request_duration_seconds_bucket{le="0.5"} 108000

Аналіз зібраних даних демонструє, що система має високий рівень стійкості при постійному навантаженні, оскільки 90% запитів були успішно оброблені. Помилки, що виникали, були переважно пов'язані зі статус-кодами 500 (внутрішня помилка сервера) та 503 (сервіс недоступний), що свідчить про недостатню потужність окремих компонентів або недосконале балансування навантаження.

Затримка в обробці запитів переважно знаходилася в межах 180 мс, що є прийнятним показником для більшості сучасних веб-додатків. Однак 5% запитів мали затримку до 300 мс, а для 1% запитів цей показник перевищував 450 мс, що вказує на наявність критичних моментів, коли продуктивність системи знижується.

ВИСНОВОК

У рамках кваліфікаційної роботи був розроблений і реалізований метод покращення зображень за допомогою.

Контейнеризація є однією з ключових технологій сучасної розробки програмного забезпечення, яка істотно впливає на масштабування мікросервісних архітектур. Дослідження продемонструвало, що застосування контейнерів значно підвищує ефективність використання обчислювальних ресурсів, забезпечуючи гнучкість у розподілі навантаження та стабільність роботи мікросервісів у динамічних середовищах. Завдяки контейнеризації з’являється можливість швидко розгортати нові інстанції сервісів, автоматизувати процеси масштабування та знижувати час простою.

Основною перевагою контейнеризації є ізоляція середовища виконання, що дозволяє уникнути конфліктів між залежностями окремих сервісів. Це забезпечує можливість паралельного розвитку та масштабування компонентів системи без порушення загальної стабільності.

Дослідження показало, що горизонтальне масштабування контейнеризованих мікросервісів є найбільш ефективним підходом у контексті обробки пікових навантажень. За рахунок додавання нових реплік сервісів можна швидко адаптувати систему до зростаючого числа запитів. Водночас вертикальне масштабування, яке передбачає збільшення ресурсів для окремих контейнерів, виявилося менш гнучким, проте може бути корисним для специфічних задач, що вимагають високої обчислювальної потужності.

Результати дослідження апробовано у вигляді 1 тези доповіді під час XII Міжнародної науково-практичну конференції «Prospective directions of modern science and education in the world».

ПЕРЕЛІК ДЖЕРЕЛ ПОСИЛАННЯ

1. Kubernetes, Inc. Kubernetes Documentation URL: https://kubernetes.io/docs/ (дата звернення 15.11.2024).

2. Docker, Inc. Docker Documentation. URL: https://docs.docker.com/ (дата звернення 15.11.2024).

3. Pahl, C., & Jamshidi, P. (2016). Microservices: A Systematic Mapping Study. *CLOSER (1)*, 137-146.

4. Saboor, A., Hassan, M. F., Akbar, R., Shah, S. N. M., Hassan, F., Magsi, S. A., & Siddiqui, M. A. (2022). Containerized microservices orchestration and provisioning in cloud computing: A conceptual framework and future perspectives. Applied Sciences, 12(12), 5793.

5. Acharya, J. N., & Suthar, A. C. (2021, October). Docker container orchestration management: A review. In International Conference on Intelligent Vision and Computing (pp. 140-153). Cham: Springer International Publishing.

6. Oyeniran, C. O., Adewusi, A. O., Adeleke, A. G., Akwawa, L. A., & Azubuko, C. F. (2024). Microservices architecture in cloud-native applications: Design patterns and scalability. Computer Science & IT Research Journal, 5(9), 2107-2124.

7. Coulson, N. C., Sotiriadis, S., & Bessis, N. (2020). Adaptive microservice scaling for elastic applications. IEEE Internet of Things Journal, 7(5), 4195-4202.

8. Srirama, S. N., Adhikari, M., & Paul, S. (2020). Application deployment using containers with auto-scaling for microservices in cloud environment. Journal of Network and Computer Applications, 160, 102629.

9. Keller, L., & Haldar, S. (2020): The role of containerization in microservices architecture. Journal of Software Engineering and Applications.

10. Gouin, V., & Sadiq, S. (2022): Performance evaluation of microservices architecture with containerization.

11. Burns, B., Grant, B., Oppenheimer, D., Brewer, E., & Wilkes, J. (2016). Borg, Omega, and Kubernetes: Lessons learned from three container-management systems over a decade. Queue, 14(1), 70-93.

12. Beda, J., Hightower, K., & Burns, B. (2017). Kubernetes: up and running. O'Reilly Media, Incorporated.

13. Newman, S. (2019). Monolith to microservices: evolutionary patterns to transform your monolith. O'Reilly Media.

14. Richards, M., & Ford, N. (2020). Fundamentals of software architecture: an engineering approach. O'Reilly Media.

15. Richardson, C., & Smith, F. (2016). Microservices: from design to deployment. Nginx Inc, 1, 24-31.

16. Brazil, B. (2018). Prometheus: Up & Running: Infrastructure and Application Performance Monitoring. " O'Reilly Media, Inc.".

17. Leppänen, T. (2021). Data visualization and monitoring with Grafana and Prometheus.

18. Shah, J., & Dubaria, D. (2019, January). Building modern clouds: using docker, kubernetes & Google cloud platform. In 2019 IEEE 9th Annual Computing and Communication Workshop and Conference (CCWC) (pp. 0184-0189). IEEE.

19. Burns, B., Grant, B., Oppenheimer, D., Brewer, E., & Wilkes, J. (2016). Borg, Omega, and Kubernetes: Lessons learned from three container-management systems over a decade. Queue, 14(1), 70-93.

20. Merkel, D. (2014). Docker: lightweight linux containers for consistent development and deployment. Linux j, 239(2), 2.

21. Sneps-Sneppe, M. (2014). On micro-services architecture. International Journal of Open Information Technologies, 2(9).

22. Lewis, J., & Fowler, M. (2014). Microservices: a definition of this new architectural term. MartinFowler. com, 25(14-26), 12.

23. Kaiser, S., Haq, M. S., Tosun, A. Ş., & Korkmaz, T. (2022). Container technologies for arm architecture: A comprehensive survey of the state-of-the-art. IEEE Access, 10, 84853-84881.

24. Turnbull, J. (2014). The Docker Book: Containerization is the new virtualization. James Turnbull.

25. Pahl, C., Brogi, A., Soldani, J., & Jamshidi, P. (2017). Cloud container technologies: a state-of-the-art review. IEEE Transactions on Cloud Computing, 7(3), 677-692.

26. Richardson, C. (2018). Microservices patterns: with examples in Java. Simon and Schuster.

27. Burns, B., Beda, J., Hightower, K., & Evenson, L. (2022). Kubernetes: up and running. " O'Reilly Media, Inc.".

28. Zhou, N., Zhou, H., & Hoppe, D. (2022). Containerization for high performance computing systems: Survey and prospects. IEEE Transactions on Software Engineering, 49(4), 2722-2740.

29. Morabito, R., Kjällman, J., & Komu, M. (2015, March). Hypervisors vs. lightweight virtualization: a performance comparison. In 2015 IEEE International Conference on cloud engineering (pp. 386-393). IEEE.

30. Villamizar, M., Garcés, O., Castro, H., Verano, M., Salamanca, L., Casallas, R., & Gil, S. (2015, September). Evaluating the monolithic and the microservice architecture pattern to deploy web applications in the cloud. In 2015 10th computing colombian conference (10ccc) (pp. 583-590). IEEE.

31. Azure Kubernetes Service (AKS). (2021). Kubernetes best practices for scalability. URL: https://learn.microsoft.com/en-us/azure/aks/best-practices.

32. Mathur, M. (2020). Leveraging Distributed Tracing and Container Cloning for Replay Debugging of Microservices. University of California, Los Angeles.

33. Morabito, R. (2015, December). Power consumption of virtualization technologies: an empirical investigation. In 2015 IEEE/ACM 8th International Conference on Utility and Cloud Computing (UCC) (pp. 522-527). IEEE.

34. Kumar, S., Saxena, R., & Kumar, N. (2020). Comparative analysis of Docker, LXC, and KVM. International Journal of Computer Applications, 176(32), 20-25.

35. Путятін, Є. П., Гороховатський, В. О., & Матат, О. О. (2006). Методи та алгоритми комп'ютерного зору: навч. посібник.

36. Творошенко, І. С. (2021). Технології прийняття рішень в інформаційних системах: навч. посібник. *Харків: ХНУРЕ*.

37. Гороховатський, В. О., & Творошенко, І. С. (2021). Методи інтелектуального аналізу та оброблення даних: навч. посібник.

38. Кобилін, О. А., & Творошенко, І. С. (2021). Методи цифрової обробки зображень: навч. посібник. *Харків: ХНУРЕ*.

39. Гороховатський В.О., Творошенко І.С. (2022) Аналіз багатовимірних даних за описом у формі множини компонент: монографія. Харків: ХНУРЕ, 124 с.

40. Malik, L., & Sandhya, A. (2021). Computing Technologies and Applications. United States: CRC Press.