МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
ЗАПОРІЗЬКИЙ НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ

МАТЕМАТИЧНИЙ ФАКУЛЬТЕТ
Кафедра комп'ютерних наук

КВАЛІФІКАЦІЙНА РОБОТА МАГІСТРА

на тему: **"МЕТОД ОЦІНКИ ПРОДУКТИВНОСТІ МІКРОСЕРВІСНОЇ АРХІТЕКТУРИ ВЕБ-ДОДАТКІВ У ХМАРНОМУ СЕРЕДОВИЩІ"**

Виконала: студентка 2 курсу, групи 8.1223
спеціальності 122 Комп'ютерні науки
(шифр і назва спеціальності)
освітньої програми Комп'ютерні науки
(назва освітньої програми)
Є. О. Мержвинська
(ініціали та прізвище)
Керівник доцент кафедри комп'ютерних наук, к.т.н. Добровольський Г.А.
(посада, вчене звання, науковий ступінь, прізвище та ініціали)
Рецензент професор кафедри програмної інженерії, доцент, к.ф.-м.н. Кудін О.В.
(посада, вчене звання, науковий ступінь, прізвище та ініціали)

Запоріжжя – 2024

МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ

ЗАПОРІЗЬКИЙ НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ

Факультет математичний

Кафедра комп'ютерних наук

Рівень вищої освіти магістр

Спеціальність 122 Комп'ютерні науки
(шифр і назва)

Освітня програма Комп'ютерні науки

ЗАТВЕРДЖУЮ
Завідувач кафедри комп'ютерних наук,
д.т.н., професор

_____ Шило Г.М.
(підпис)

" 5 " березня 2024 р.

З А В Д А Н Н Я

НА КВАЛІФІКАЦІЙНУ РОБОТУ СТУДЕНТЦІ

Мержвинській Єлизаветі Олександрівні
(прізвище, ім'я та по-батькові)

1. Тема роботи Метод оцінки продуктивності мікросервісної архітектури веб-додатків у хмарному середовищі

керівник роботи Добровольський Генадій Анатолійович, к.т.н.
(прізвище, ім'я та по-батькові, науковий ступінь, вчене звання)

затверджені наказом ЗНУ від " 01 " березня 2024 року № 385-с

2. Строк подання студентом роботи 27.11.2024

3. Вихідні дані до роботи 1. Постановка задачі.
2. Перелік літератури.

4. Зміст розрахунково-пояснювальної записки (перелік питань, які потрібно розробити)

1. Постановка задачі.
2. Основні теоретичні відомості.
3. Реалізація та оцінка симуляційної моделі мікросервісної архітектури веб-додатку.

5. Перелік графічного матеріалу (з точним зазначенням обов'язкових креслень)
презентація

6. Консультанти розділів роботи

Розділ	Прізвище, ініціали та посада консультанта	Підпис, дата	
		завдання видав	завдання прийняв

7. Дата видачі завдання 02.03.2024

К А Л Е Н Д А Р Н И Й П Л А Н

№	Назва етапів кваліфікаційної роботи	Строк виконання етапів роботи	Примітка
1.	Розробка плану роботи.	02.03.2024	
2.	Збір вихідних даних.	02.04.2024	
3.	Обробка методичних та теоретичних	02.06.2024	
	джерел.		
4.	Розробка першого та другого розділу.	02.09.2024	
5.	Розробка третього розділу.	02.10.2024	
6.	Оформлення та нормоконтроль	02.11.2024	
	кваліфікаційної роботи магістра.		
7.	Захист кваліфікаційної роботи.	12.12.2024	

Студент _____ Є.О.Мержвинська
(підпис) (ініціали та прізвище)

Керівник роботи _____ Г.А.Добровольський
(підпис) (ініціали та прізвище)

Нормоконтроль пройдено

Нормоконтролер _____ О.Г.Спиця
(підпис) (ініціали та прізвище)

РЕФЕРАТ

Кваліфікаційна робота магістра “Метод оцінки продуктивності мікросервісної архітектури веб-додатків у хмарному середовищі”: 38 с., 5 рис., 1 табл., 14 джерел.

АВТОМАТИЧНЕ МАСШТАБУВАННЯ, АРХІТЕКТУРНІ ПАТЕРНИ, БАЛАНСУВАЛЬНИК НАВАНТАЖЕННЯ, МІКРОСЕРВІСНА АРХІТЕКТУРА, НАВАНТАЖУВАЛЬНЕ ТЕСТУВАННЯ, ПРОДУКТИВНІСТЬ ВЕБ-ДОДАТКІВ, ХМАРНІ СЕРЕДОВИЩА.

Об’єкт дослідження – архітектура мікросервісних веб-додатків у хмарному середовищі.

Мета роботи: Розробка та впровадження методів оцінки продуктивності мікросервісної архітектури веб-додатків у хмарному середовищі.

Метод дослідження – експериментальний.

У даній роботі розглядаються методи оцінки продуктивності мікросервісної архітектури веб-додатків, розгорнутих у хмарному середовищі AWS. Мікросервісна архітектура стає все більш популярною завдяки своїй гнучкості, масштабованості та можливості незалежного розгортання окремих сервісів. Однак забезпечення високої продуктивності та надійності системи в умовах змінного навантаження є складним завданням, що вимагає застосування ефективних методів оцінки.

Метою роботи є розробка та впровадження комплексних методів оцінки продуктивності мікросервісної архітектури, з особливим фокусом на різні архітектурні підходи, такі як ланцюгові та деревовидні структури. Для досягнення цієї мети були визначені ключові метрики продуктивності, розроблена метод їх вимірювання та аналізу.

SUMMARY

Master's Qualifying Theses; "Method of Estimating the Productivity of Microservice Architecture of Web Applications in the Cloud Environment": 38 pages, 5 figures, 1 table, 14 references.

ARCHITECTURAL PATTERNS, AUTOMATIC SCALING, CLOUD ENVIRONMENTS, LOAD BALANCER, LOAD TESTING, MICROSERVICE ARCHITECTURE, WEB APPLICATION PERFORMANCE.

Object of the study – Microservice architecture of web applications in a cloud environment.

Aim of the study: Development and implementation of methods for evaluating the performance of microservice architecture of web applications in a cloud environment.

Method of research – Experimental.

This study explores methods for evaluating the performance of microservice architecture in web applications deployed in the AWS cloud environment. Microservice architecture is becoming increasingly popular due to its flexibility, scalability, and the ability to independently deploy individual services. However, ensuring high performance and system reliability under variable loads is a challenging task that requires the application of effective evaluation methods.

The aim of this work is to develop and implement comprehensive methods for evaluating the performance of microservice architecture, with a particular focus on different architectural approaches, such as chain and tree structures. To achieve this goal, key performance metrics were identified, and methods for their measurement and analysis were developed.

ЗМІСТ

Завдання на кваліфікаційну роботу……..2
Реферат 4
Summary 5
Вступ..... 8
1 Продуктивність мікросервісної архітектури: принципи, моделі та оцінка ... 9
1.1 Мікросервісна архітектура 9
1.2 Принципи проектування мікросервісів 9
1.3 Переваги та недоліки мікросервісної архітектури 10
1.4 Види взаємодії між мікросервісами 11
1.5 Порівняння мікросервісної та монолітної архітектури..... 12
1.6 Визначення систем масового обслуговування..... 13
1.6.1 Теорія масового обслуговування в контексті монолітної архітектури 14
1.6.2 Теорія масового обслуговування в контексті мікросервісної архітектури 15
2 Реалізація симуляційної моделі мікросервісної архітектури веб-додатку... 17
2.1 Основних компонентів системи 17
2.2 Сценарій використання веб додатку 18
2.3. Використанні технології..... 19
2.3.1 Контейнеризація та оркестрація..... 20
2.3.2 Використання AWS для управління інфраструктурою 21
2.4 Архітектура мікросервісної системи..... 22
2.4.1 Патерн “Ланцюг відповідальностей”..... 22
2.4.2 Патерн “Компонувальник” 24
3 Оцінка продуктивності мікросервісної архітектури веб-додатків..... 26
3.1 Ключові показники для оцінки архітектур..... 26

3.2 Інструмент k6 для навантажувального тестування 27
3.2.1 Моделювання потоку запитів 27
3.3 Альтернативний інструмент µBench 27
3.4 Порівняння ефективності монолітної та мікросервісної архітектури 29
3.5 Аналіз різних підходів до побудови мікросервісних систем 30
Висновки 35
Перелік посилань 37

ВСТУП

У сучасному світі інформаційних технологій мікросервісна архітектура стає все більш популярною парадигмою для розробки веб-додатків. Зростання вимог до веб-додатків щодо швидкості обробки запитів, доступності та масштабованості робить дослідження методів оцінки продуктивності мікросервісної архітектури особливо актуальним. А хмарні середовища надають можливості для розгортання та масштабування мікросервісів, однак ефективне використання цих можливостей потребує ретельної оцінки продуктивності.

Оцінка ефективності мікросервісів полягає у вимірюванні та аналізі різних параметрів, таких як швидкодія, масштабованість, витрати ресурсів та надійність системи. Основні методи оцінки включають навантажувальне тестування, моніторинг метрик системи, аналіз затримок та використання ресурсів, а також моделювання різних сценаріїв роботи системи, розробка підходів до архітектури таких систем.

Незважаючи на різноманітність патернів, що використовуються в мікросервісній архітектурі, таких як API Gateway, Service Mesh, Event-Driven Architecture, Saga Pattern, Circuit Breaker, CQRS (Command Query Responsibility Segregation), Bulkhead Pattern та Sidecar Pattern, існують численні виклики, які потребують подальших досліджень та вирішення.

Метою роботи є розробка та впровадження методів оцінки продуктивності мікросервісної архітектури веб-додатків у хмарному середовищі. Це включає визначення ключових метрик продуктивності, їх вимірювання та аналізу, а також створення інструментів для автоматизації процесу оцінки. Особливий фокус роботи приділяється різним архітектурним підходам, таким як ланцюгові, деревовидні.

1 ПРОДУКТИВНІСТЬ МІКРОСЕРВІСНОЇ АРХІТЕКТУРИ: ПРИНЦИПИ, МОДЕЛІ ТА ОЦІНКА

1.1 Мікросервісна архітектура

Мікросервісна архітектура — це архітектурний стиль організації програмних систем, що ґрунтується на розподілі функціональних складових системи на слабко пов’язані, але такі, що взаємодіють, компоненти.

У мікросервісній архітектурі сервісом називається незалежний компонент додатку, який виконує чітко визначену функцію. Кілька сервісів взаємодіють у рамках єдиної системи, забезпечуючи реалізацію ключових функцій для досягнення цілей системи.

1.2 Принципи проектування мікросервісів

Мікросервісна архітектура суттєво відрізняється від традиційного монолітного підходу завдяки її особливостям, які забезпечують гнучкість, модульність та незалежність компонентів. Основна концепція полягає у побудові системи на основі незалежних компонентів-сервісів, кожен з яких автономний у розробці, оновленні та розгортанні.

Основні принципи мікросервісної архітектури:

– кожен компонент відповідає за окрему ділянку бізнес-логіки;
– кожен окремий компонент системи є незалежною одиницею;
– слабке зв’язування компонентів системи;
– технологічна незалежність;
– розподілене зберігання даних. [1 с. 2]

1.3 Переваги та недоліки мікросервісної архітектури

Мікросервісна архітектура ґрунтується на низці принципів, кожен із яких формує як переваги, так і недоліки цього підходу. Однією з ключових особливостей є поділ системи на окремі сервіси, кожен з яких реалізує конкретну ділянку функціональної логіки. Такий підхід дозволяє значно спростити підтримку та розвиток системи, адже внесення змін в один компонент не впливає на інші. Однак це водночас вимагає ретельного планування взаємодії між сервісами, щоб уникнути збоїв у роботі системи.

Автономність кожного компонента забезпечує можливість його окремого розгортання, оновлення та виконання. Це підвищує надійність системи, оскільки несправність одного з компонентів не порушує функціонування інших. Водночас така автономність створює додаткові виклики в управлінні залежностями, оскільки необхідно забезпечити узгодженість даних між сервісами.

Слабке зв’язування компонентів є ще одним важливим аспектом, який мінімізує їхню взаємозалежність. Взаємодія між сервісами відбувається через стандартизовані API, що спрощує заміну або оновлення окремих частин системи. Це сприяє її гнучкості, однак вимагає створення додаткової інфраструктури для забезпечення надійного зв’язку між компонентами.

Технологічна незалежність дозволяє використовувати різні платформи, бази даних та інструменти для кожного сервісу, залежно від його потреб. Це сприяє адаптивності системи до змін у вимогах, дозволяючи інтегрувати новітні технології. Проте така різноманітність ускладнює координацію між командами, які працюють над різними компонентами.

Розподілене зберігання даних зменшує навантаження на централізовані бази та підвищує надійність, адже кожен сервіс має власне сховище. Утім, це може створити труднощі в узгодженні даних між сервісами, особливо в умовах високої динаміки змін.

Отже, мікросервісна архітектура забезпечує значну гнучкість, масштабованість і надійність системи, але потребує ретельного управління та контролю.

1.4 Види взаємодії між мікросервісами

У рамках дослідження продуктивності мікросервісної архітектури важливим є аналіз методів взаємодії між компонентами системи, які впливають на узгодженість даних, швидкість обробки запитів та масштабованість.

Серед методів взаємодії між мікросервісами виділяють синхронні та асинхронні способи обміну даними, які визначають принципи організації комунікацій у системі. Синхронна взаємодія передбачає, що один сервіс викликає інший і чекає завершення операції. Цей підхід забезпечує передбачуваність виконання, оскільки сервіс одразу отримує відповідь про успішність чи помилку операції. Така модель є доцільною у випадках, коли критично важливим є забезпечення узгодженості даних між компонентами або своєчасне підтвердження виконання задачі. Синхронні виклики добре підходять для транзакційних операцій чи задач із високими вимогами до контролю процесу.

Асинхронна модель комунікації звільняє викликаючий сервіс від необхідності чекати завершення операції. Сервіс, що ініціює запит, може продовжувати свою роботу, отримуючи результат лише після завершення обробки. Цей підхід є ефективним для довготривалих або ресурсомістких операцій, особливо у середовищах, де зменшення затримок у взаємодії є ключовим фактором. Асинхронна взаємодія особливо актуальна в мікросервісних системах або для задач, де підтримання постійного з’єднання між компонентами є складним або недоцільним. Проте імплементація асинхронної моделі є складнішою через необхідність організації обробки відповідей, управління чергами та забезпечення надійності комунікації.

Класичним методом обміну даними є підхід запит/відповідь, у якому клієнт ініціює запит до сервера та чекає на його обробку. Такий підхід зазвичай є синхронним, однак може бути адаптований до асинхронної взаємодії за допомогою механізмів зворотних викликів. У цьому випадку сервер самостійно повідомляє клієнту про завершення операції. Альтернативою є подійно-орієнтована модель, яка передбачає передачу подій між компонентами без прямого очікування відповіді. В такій архітектурі ініціатор події повідомляє систему про стан чи зміну, а інші сервіси реагують на ці події. Це забезпечує високу масштабованість і дозволяє легко інтегрувати нові сервіси без модифікації існуючих.

Синхронна взаємодія забезпечує простоту контролю і передбачуваність роботи системи, але створює потенційні затримки через необхідність очікування відповіді. Асинхронна модель, хоча й складніша в реалізації, відкриває значні можливості для оптимізації роботи з великими обсягами даних або довготривалими операціями. Вибір підходу залежить від специфіки задачі та вимог до продуктивності системи.

1.5 Порівняння мікросервісної та монолітної архітектури

Монолітна архітектура створює додаток як єдиний логічний блок, де всі компоненти – від клієнтського інтерфейсу до бази даних і серверної логіки інтегровані у спільний процес. Такий підхід забезпечує ефективну комунікацію всередині системи, оскільки немає необхідності у додаткових витратах на зв'язок між компонентами. Це робить монолітні додатки продуктивнішими при обробці невеликої кількості запитів або при роботі з обмеженим числом користувачів. Крім того, монолітна архітектура сприяє простоті управління й розгортання, що робить її доречною для невеликих додатків із стабільною логікою.

Втім, моноліт має й суттєві обмеження. Зміна будь-якої частини додатка вимагає перебудови всього програмного продукту, що значно ускладнює впровадження нових функцій та виправлення помилок у великих системах. Це обмеження стає особливо очевидним у хмарних середовищах, де гнучкість і адаптивність є критично важливими. Масштабування моноліту потребує збільшення ресурсів для всієї системи, що може бути ресурсно затратним.

Мікросервісна архітектура пропонує альтернативу, розбиваючи додаток на набір незалежних сервісів, кожен з яких виконує окремі функції та може масштабуватись окремо. Це дозволяє командам працювати автономно над різними частинами проєкту та використовувати прийнятні інструменти й технології для кожного сервісу. Мікросервісний підхід особливо підходить для великих систем із складною бізнес-логікою, де важлива гнучкість та здатність до швидкого розширення функціоналу. Зокрема, при високих навантаженнях можливість незалежного масштабування дозволяє оптимізувати розподіл ресурсів, концентруючи їх там, де це найнеобхідніше.

Однак, необхідність комунікації між сервісами може створювати затримки й вимагати ретельної координації. Сервіс-виявлення, що використовується для орієнтації в мережі, також впливає на продуктивність.

Таким чином, вибір між монолітною та мікросервісною архітектурою слід робити, зважаючи на особливості проекту: для малих додатків з обмеженою кількістю користувачів моноліт може виявитися продуктивнішим і простішим у підтримці, тоді як для масштабованих і складних систем мікросервіси забезпечать більшу гнучкість, стійкість до збоїв та ефективне управління ресурсами.

1.6 Визначення систем масового обслуговування

Формально під системою масового обслуговування розуміють складну систему, що складається з одного або декількох джерел запитів (заявок, вимог)

на виконання певних дій, декількох приладів обслуговування, які виконують ці дії відповідно до певних правил (дисципліни обслуговування) за запитами, що надійшли в систему [2, с. 54].

Для повного визначення системи масового обслуговування необхідно описати стохастичні процеси, які характеризують потоки запитів, а також структуру та правила обслуговування. Основні елементи для вивчення стохастичних потоків включають:

– процес надходження запитів;
– розподіл часу обслуговування;
– пропускну здатність системи для розміщення клієнтів, що очікують;
– кількість наявних станцій обслуговування;
– правила черги (дисципліну черги);
– наявність або відсутність поведінки клієнтів. [3, с. 3].

Системи масового обслуговування можуть слугувати фундаментом для моделювання процесів у системах з мікросервісною архітектурою, де важливо аналізувати ефективність розподілу запитів, управління чергами та оптимального використання ресурсів.

1.6.1 Теорія масового обслуговування в контексті монолітної архітектури

Монолітна архітектура веб-додатків характеризується інтеграцією всіх компонентів системи в єдиний логічний блок. У цьому контексті теорія масового обслуговування стає важливим інструментом для аналізу продуктивності та ефективності обробки запитів. Основною метою застосування цієї теорії є моделювання поведінки системи під різними навантаженнями та визначення ключових показників, таких як час очікування в черзі (W_q), середня кількість запитів у черзі (L_q) та рівень завантаження системи (ρ).

Монолітна архітектура зазвичай моделюється за допомогою класичних моделей масового обслуговування, таких як *M/M/c* або *M/M/1*, де:

– *M/M/c* означає модель з пуассонівським потоком запитів (перший *M*), експоненційним розподілом часу обслуговування (другий *M*) та *c* паралельними серверами.
– *M/M/1* є спеціальним випадком з одним сервером.

У монолітній системі всі запити надходять до єдиного блоку обробки, що відповідає системі з *c* серверами. Припускаючи, що запити надходять з інтенсивністю λ (запитів на секунду) та мають середній час обслуговування 1/μ (секунд), рівень завантаження системи визначається як:

$$\rho = \frac{\lambda}{c\mu}.$$

Монолітні системи характеризуються централізованим управлінням ресурсами, що призводить до певних особливостей у контексті теорії масового обслуговування.

По перше, всі компоненти системи спільно використовують обчислювальні ресурси, пам’ять та мережеву інфраструктуру. Це призводить до того, що рівень завантаження (ρ) безпосередньо залежить від загальної пропускної здатності системи та інтенсивності запитів.

По друге, у монолітній архітектурі всі запити формують єдину чергу перед блоком обробки. Це означає, що час очікування та кількість запитів у черзі залежать від кількості серверів (*c*) та інтенсивності запитів (λ).

1.6.2 Теорія масового обслуговування в контексті мікросервісної архітектури

Мікросервісна архітектура веб-додатків характеризується розподіленістю компонентів системи, кожен з яких виконує окрему бізнес-функцію та взаємодіє з іншими сервісами через мережеві виклики.

У мікросервісній архітектурі кожен сервіс виконує конкретну функцію та може виступати як джерело або обслуговуючий елемент у системі масового обслуговування. Запити від користувачів або інших сервісів надходять до відповідних мікросервісів, де формуються черги очікування на обробку. Ця розподілена природа взаємодій створює мережу взаємопов'язаних черг, де кожен сервіс може мати власну модель обслуговування.

Мікросервісну систему можна моделювати за допомогою різних моделей масового обслуговування, залежно від специфіки обробки запитів кожного сервісу. Основними моделями, які використовуються для аналізу мікросервісних архітектур, є M/M/c та M/G/c.

Модель M/M/c передбачає пуассонівське розподілення часу між запитами (λ) та експоненційний розподіл часу обслуговування (μ), з *c* паралельними серверами. Ця модель підходить для сервісів з однотипними запитами та стабільним часом обслуговування. Наприклад, сервіс авторизації, який обробляє запити користувачів зі стабільним часом обробки.

З іншого боку, модель M/G/c дозволяє використовувати довільний розподіл часу обслуговування, що є більш гнучким для сервісів зі складною бізнес-логікою або варіативними часами обробки запитів.

2 РЕАЛІЗАЦІЯ СИМУЛЯЦІЙНОЇ МОДЕЛІ МІКРОСЕРВІСНОЇ АРХІТЕКТУРИ ВЕБ-ДОДАТКУ

Для глибшого розуміння впливу різних архітектурних рішень на продуктивність та ефективність обробки запитів, необхідно розробити та проаналізувати симуляційну модель, яка відображає реальні умови роботи розподіленої системи. Цей розділ присвячено опису процесу створення такої моделі.

2.1 Основних компонентів системи

Проект побудовано як мікросервісну систему, основним компонентом якої є сервіс сортування. Цей сервіс реалізує функції сортування масивів числових даних, підтримуючи три алгоритми: швидке сортування, сортування вставками та сортування обміном. Вибір цих алгоритмів обґрунтовано їх різними характеристиками ефективності та складності, що дозволяє моделювати різні сценарії для оцінки продуктивності системи.

Сервіс також має функціонал для симуляції реальних умов роботи. Зокрема, можна задавати частоту виникнення помилок під час виконання запитів, а також середній час затримки перед відповіддю. Це дозволяє моделювати поведінку сервісу в умовах реального навантаження, враховуючи можливі затримки та збої, які типові для розподілених систем.

Взаємодія між сервісами відбувається через HTTP-запити з використанням REST API. Це забезпечує простоту інтеграції та стандартизацію обміну даними. У контексті тестування мікросервісної архітектури сервіс сортування може працювати в різних конфігураціях, виконуючи запити відповідно до обраної моделі архітектури (наприклад, дерево чи ланцюжок

обов’язків). У межах цих моделей сервіс виконує роль як кінцевого виконавця, так і проміжного вузла для перенаправлення запитів іншим мікросервісам.

Всі результати виконання логуються, що дозволяє проводити аналіз роботи сервісу в реальному часі. Для навантажувального тестування використовується інструмент K6, що дозволяє оцінити час виконання сортування, рівень завантаженості процесора та використання пам’яті під час роботи сервісу.

Система розгорнута в контейнерах за допомогою Docker, а для управління контейнерами використовується Kubernetes. Це забезпечує автоматичне масштабування.

Важливою особливістю проекту є тестування різних архітектур мікросервісів, таких як дерево та ланцюжок обов’язків. Також проводиться порівняння мікросервісної моделі з монолітною архітектурою для оцінки ефективності кожного підходу.

Ця архітектура дозволяє комплексно дослідити переваги та недоліки різних підходів до побудови програмних систем.

2.2 Сценарій використання веб додатку

Клієнт (клієнтом виступає браузер або інструмент для навантажувального тестування) надсилає запити до сервісу через HTTP-запити, в яких передаються параметри операції: масив даних для сортування та обраний алгоритм. Вибір алгоритму сортування є важливим елементом функціоналу, оскільки різні алгоритми мають різну ефективність залежно від розміру масиву.

Після виконання операції сервіс повертає клієнту відповідь, яка містить: статус виконання запиту (успішно чи з помилкою) та відсортований масив.

2.3. Використанні технології

Технічна реалізація проєкту базується на використанні сучасних інструментів та технологій. Основним компонентом системи є сервіс сортування, реалізований на платформі Spring Boot 3.1.5.

Для управління схемою бази даних використовується Flyway 10.17.0. Міграції даних реалізовані як окремі SQL-скрипти, що дозволяють автоматично налаштовувати структуру бази під час розгортання системи.

База даних PostgreSQL 16.0. зберігає ключову інформацію про виконання запитів, зокрема:

– ідентифікатор операції;
– обраний алгоритм сортування;
– розмір масиву;
– час виконання операції;
– статус виконання (чи виникли помилки).

Симуляція затримки та помилок є частиною основної логіки сервісу сортування. Перед поверненням відповіді до клієнта додається контрольована затримка, що задається у конфігурації запиту. Частота виникнення помилок також конфігурується і дозволяє моделювати непередбачувані ситуації, характерні для реальних систем.

Записи про виконання запитів логуються за допомогою бібліотеки SLF4J 2.0.9 із використанням елементу LoggerFactory. Це дозволяє фіксувати ключові події, пов’язані з обробкою запитів, і спрощує діагностику під час тестування.

Розгортання та тестування системи проходить в хмарному середовищі AWS. AWS забезпечує інфраструктуру для автоматичного масштабування ресурсів, що дозволяє додатку адаптуватися до змінного навантаження.

Документування API реалізовано за допомогою Swagger, що забезпечує простоту інтеграції та тестування взаємодії з сервісом, хоча цей аспект не є ключовим у межах дослідження.

Така архітектура та технічна реалізація дозволяють не лише ефективно виконувати сортування, але й забезпечують можливість моделювання реальних умов роботи, аналізу продуктивності системи та її компонентів у розподілених середовищах.

2.3.1 Контейнеризація та оркестрація

Контейнеризація та оркестрація є критично важливими при розробці веб-додатків мікросервісної архітектури, оскільки вони забезпечують управління розподіленими системами.

У рамках реалізації проекту використовуються Docker 24.0.5 та Kubernetes 1.28.2, які забезпечують ізоляцію, масштабованість сервісу сортування.

Docker [4] застосовується для контейнеризації сервісу, що дозволяє створити уніфіковане середовище для його виконання. Контейнер включає всі необхідні залежності, конфігурації та додатки, що забезпечує однакову роботу сервісу незалежно від середовища розгортання. Завдяки Docker, розробники можуть легко розгортати сервіс на будь-якій платформі, що підтримує цю технологію, зводячи до мінімуму ризики несумісності між середовищами.

Kubernetes [5] використовується для оркестрації контейнерів. Він дозволяє автоматизувати процеси розгортання, управління та масштабування сервісу. Використання Kubernetes дозволяє автоматично масштабувати систему, додаючи або видаляючи контейнери залежно від поточного навантаження, що забезпечує оптимальне використання ресурсів. Крім того, Kubernetes здійснює балансування навантаження шляхом рівномірного розподілу запитів між контейнерами. Важливою складовою є також постійний моніторинг та управління станом контейнерів, Kubernetes автоматично перезапускає їх у разі збоїв, забезпечуючи безперервну роботу сервісів.

2.3.2 Використання AWS для управління інфраструктурою

Використання Amazon Web Services [6] спрощує управління інфраструктурою завдяки інтегрованим сервісам, які автоматизують та оптимізують різні аспекти розгортання та підтримки додатків. Amazon Elastic Compute Cloud (EC2) [7] надає можливість швидкого розгортання віртуальних серверів з необхідними ресурсами та конфігураціями.

Amazon Elastic Kubernetes Service (EKS) [8] спрощує розгортання та управління Kubernetes-кластером, забезпечуючи автоматизацію оновлень, масштабування та безпеки.

Amazon Elastic Load Balancing (ELB) [9] забезпечує автоматичний розподіл вхідного трафіку між кількома ресурсами, що підвищує відмовостійкість та продуктивність системи. ELB підтримує різні протоколи та типи балансування для конкретних потреб додатку.

Сукупне використання цих сервісів надає можливість створювати масштабовані, стійкі та високодоступні системи з мінімальними зусиллями з боку розробників та адміністраторі, зосередитися на ключових аспектах проекту та прискорює процес розробки та розгортання додатків.

Використано також Horizontal Pod Autoscaler (HPA) [10] для автоматичного масштабування подів у кластері Kubernetes. HPA моніторить ключові метрики, такі як використання процесора або пам'яті, і динамічно регулює кількість реплік подів відповідно до поточного навантаження. Принцип роботи HPA полягає у періодичному зборі метрик з подів та порівнянні їх зі встановленими цільовими значеннями. Якщо фактичні показники навантаження перевищують задані пороги, HPA збільшує кількість подів, щоб розподілити навантаження. У протилежному випадку, коли навантаження знижується, він зменшує кількість реплік для економії ресурсів.

2.4 Архітектура мікросервісної системи

У цьому розділі детально описується архітектура розробленої мікросервісної системи, побудованої з використанням різних патернів проектування для оцінки їх ефективності та продуктивності. Основною метою є дослідження впливу обраних патернів на продуктивність системи масового обслуговування, а також визначення їхніх переваг та обмежень у контексті мікросервісної архітектури.

Особлива увага приділяється застосуванню патернів “Ланцюжок відповідальностей” та “Компонувальник”, які дозволяють організувати взаємодію між сервісами та оптимізувати процес сортування даних. Далі наведено опис кожного з патернів, їх реалізацію у системі.

2.4.1 Патерн “Ланцюг відповідальностей”

Ланцюг відповідальності — це шаблон поведінкового проектування, який дозволяє передавати запити по ланцюжку обробників.[11, с. 251]

Мікросервісна архітектура, побудована за принципом “Ланцюжка відповідальностей”, передбачає організацію взаємодії між компонентами системи у вигляді послідовної передачі запиту через низку сервісів, кожен з яких виконує певну функцію. Коли перший елемент у ланцюгу отримує запит, він вирішує: обробити його самостійно або передати далі наступному кандидату в ланцюзі, який повторює цей процес. Об'єкт, що ініціював запит, не знає наперед, хто саме його обробить. Такий підхід забезпечує чітке розподілення відповідальності між сервісами, що сприяє підвищенню модульності та гнучкості системи. У рамках цієї топології кожен сервіс функціонує автономно, використовуючи стандартизований протокол для передачі запитів. Така організація роботи дозволяє уникати надмірної взаємозалежності між компонентами. Взаємодія в ланцюзі будується за

принципом слабкого зв’язування, що забезпечує можливість додавання або видалення обробників без необхідності внесення змін у всю систему.

Архітектура з ланцюгом обов’язків є ефективним рішенням у випадках, коли обробка запиту потребує послідовного виконання кількох етапів. У контексті проекту ця архітектура дозволяє організувати процес сортування масивів даних із використанням різних алгоритмів. Наприклад, початковий сервіс може визначати тип алгоритму сортування, другий — виконувати безпосередню обробку даних, третій — моделювати затримку або виникнення помилок, а кінцевий сервіс відповідальний за формування та передачу результату клієнту. Для оцінки та тестування був обраний сценарій, коли сервіси послідовно передають користувацький масив, сортуючи його частину.

Використання цієї архітектури забезпечує структурованість процесу обробки даних, дозволяючи тестувати різні конфігурації взаємодії між сервісами. Однак, слід враховувати, що зростання кількості обробників у ланцюгу може призводити до підвищення затримок під час обробки запитів, а також ускладнювати діагностику та управління системою.

Така модель організації мікросервісів створює передумови для глибокого аналізу продуктивності та оптимізації бізнес-процесів у розподілених середовищах, що робить її важливим інструментом для тестування та вдосконалення сучасних систем.

Ще однією важливою характеристикою є динамічний ланцюг обробників. Під час виконання програми є можливість змінювати ланцюжок обробників, оскільки можна додавати або видаляти обробники без необхідності змінювати основну логіку системи. Це значно спрощує підтримку та розширення функціоналу без порушення існуючих структур.

Принцип єдиної відповідальності також відіграє ключову роль у патерні “Ланцюжок відповідальності”. Кожен обробник у ланцюжку має одну чітку задачу: або обробити запит, або передати його наступному обробнику в ланцюгу.

Запити рухаються по ланцюжку один за одним, що забезпечує послідовний порядок їх обробки. Кожен обробник отримує можливість обробити запит у визначеному порядку, що гарантує та передбачуваність поведінки системи.

Крім того, патерн “Ланцюжок відповідальності” передбачає механізм резервного варіанту обробки запитів. Якщо жоден з обробників не може впоратися з запитом, ланцюжок може містити опцію за замовчуванням для обробки таких запитів. Це забезпечує наявність стандартного способу вирішення запитів, які не підпадають під конкретні обробники, підвищуючи надійність та стійкість системи до несподіваних ситуацій.

Таким чином, патерн “Ланцюжок відповідальності” забезпечує ефективний механізм обробки запитів у мікросервісних системах, дозволяючи досягти високої гнучкості, модульності та стійкості до збоїв.

2.4.2 Патерн “Компонувальник”

Патерн “Компонувальник” є одним із фундаментальних шаблонів проектування, який дозволяє створювати ієрархічні деревоподібні структури об'єктів, де окремі компоненти та їхні композиції обробляються єдиним способом. [3, с.183]

Цей патерн особливо корисний у контексті мікросервісної архітектури, де система складається з численних незалежних сервісів, організованих у складні структури.

У мікросервісній системі патерн “Компонувальник” дозволяє будувати дерево сервісів, де кожен сервіс може виступати як окремим об'єктом, так і контейнером для інших сервісів. Це забезпечує гнучкість у розподілі завдань та дозволяє легко масштабувати систему шляхом додавання нових сервісів на різних рівнях дерева без необхідності змінювати існуючу структуру.

Застосування цього патерну у мікросервісній архітектурі сприяє уніфікації інтерфейсів взаємодії між сервісами, що спрощує процес інтеграції

та управління ними. Крім того, патерн “Компонувальник” підвищує модульність системи, оскільки кожен сервіс відповідає за виконання конкретної функції, при цьому маючи можливість делегувати частини роботи своїм дочірнім сервісам. Це дозволяє ефективно розподіляти навантаження при зростанні обсягу оброблюваних даних.

В системі, де кореневий сервіс розбиває запит на сортування масиву даних та розподіляє його між дочірніми сервісами, патерн “Компонувальник” дозволяє організувати цю взаємодію у вигляді дерева сервісів. Кожен сервіс може або безпосередньо виконувати сортування отриманих частин, або делегувати цю задачу своїм власним дочірнім сервісам, забезпечуючи таким чином рекурсивну обробку даних. Після завершення сортування всі частини повертаються до кореневого сервісу, який об'єднує їх у фінальний відсортований масив та відправляє результат клієнту. Така архітектура сприяє високій гнучкості, легкості масштабування та спрощенню управління складними процесами обробки запитів у розподілених системах.

Опис роботи системи:

- кореневий сервіс отримує запит на сортування масиву даних;
- він розбиває масив на частини відповідно до кількості своїх дочірніх сервісів і відправляє кожну частину відповідному дочірньому сервісу;
- дочірні сервіси приймають отримані частини масиву і:
  - або сортують їх самостійно;
  - або, якщо вони мають власних дочірніх сервісів, поділяють отриману частину ще на менші частини і передають їх своїм дочірнім сервісам;
- цей процес рекурсивно повторюється, поки частини масиву не будуть достатньо малими для ефективного сортування;
- після завершення сортування, відсортовані частини відправляються назад до батьківських сервісів;
- кореневий сервіс збирає всі відсортовані частини, об'єднує їх у фінальний відсортований масив і відправляє результат клієнту.

3 ОЦІНКА ПРОДУКТИВНОСТІ МІКРОСЕРВІСНОЇ АРХІТЕКТУРИ ВЕБ-ДОДАТКІВ

3.1 Ключові показники для оцінки архітектур

Оцінка архітектур мікросервісних систем базується на трьох ключових показниках, які мають критичне значення для їхньої ефективності.[12, с.73]

Масштабованість є основним критерієм, що визначає, наскільки легко система може бути адаптована до зростання обсягу запитів або обробки даних. У цьому контексті важливим є горизонтальне масштабування, яке забезпечує додавання нових вузлів до системи без втрати продуктивності. Архітектури, які підтримують слабке зв’язування між компонентами, мають кращу масштабованість, оскільки окремі сервіси можуть бути розгорнуті незалежно.

Продуктивність визначається швидкістю обробки запитів і часом відгуку системи. На неї впливають такі фактори, як ефективність обміну даними між сервісами, складність обробки запитів та обсяг мережевого трафіку. Архітектури з мінімальними затримками під час передачі даних між компонентами демонструють вищу продуктивність.

Надійність оцінює здатність системи продовжувати функціонувати у разі відмови одного чи кількох її компонентів. Вона забезпечується через ізоляцію сервісів, їхню автономність та використання механізмів автоматичного відновлення. Розподілені архітектури зазвичай мають перевагу в цьому аспекті завдяки локалізації помилок і відмовостійкості.

3.2 Інструмент k6 для навантажувального тестування

k6 — це інструмент із відкритим вихідним кодом і хмарний сервіс, який виконує навантажувальне тестування. Він дозволяє створювати скрипти тестування мовою JavaScript, що спрощує моделювання реалістичних сценаріїв користувацької активності. k6 надає можливість збирати детальні метрики, такі як час відповіді, пропускна здатність та показники відмов. [13]

3.2.1 Моделювання потоку запитів

Для моделювання потоку запитів розроблено скрипт для k6. Задано параметр λ, який визначає середню інтенсивність вхідного потоку запитів та становить 5 запитів в секунду. Параметр *c* відповідає кількості каналів обслуговування та становить 1.

У скрипті для k6 моделюється потік запитів на основі моделі масового обслуговування M/M/c. Параметр λ (lambda) визначає середню інтенсивність вхідного потоку запитів, а параметр c відповідає кількості каналів обслуговування. Розроблена функція exponentialInterval(lambda), яка генерує інтервали між запитами за експоненціальним розподілом.

Скрипт імітує запити з різними комбінаціями розмірів масивів та алгоритмів сортування, що дозволяє оцінити поведінку сервісу під різним навантаженням. Скрипт збирає метрики, такі як час відповіді та коефіцієнт успішних запитів.

3.3 Альтернативний інструмент µBench

µBench [14] — це інструмент для генерації та тестування мікросервісних застосунків у хмарних і кордонних обчислювальних платформах. Він дозволяє створювати застосунки, що складаються з мікросервісів із регульованою

кількістю та моделлю роботи. Клієнти взаємодіють із застосунком через API-шлюз, який спрямовує вхідні HTTP-запити до відповідних внутрішніх сервісів.

Інструмент дозволяє користувачам створювати конфігураційні файли, які описують взаємозв'язки між мікросервісами, шляхи проходження запитів через мережу (послідовно, паралельно або випадково) та ймовірності виклику кожного сервісу. Наприклад, можна задати централізовану топологію типу "зірка", де центральний мікросервіс викликає одночасно кілька інших сервісів. Користувачі можуть контролювати різні аспекти створюваних застосунків, зокрема поведінку сервісів за допомогою набору стрес-функцій (навантаження на CPU, пам'ять, ввід-вивід, мережу), протоколи взаємодії (HTTP або gRPC), а також ресурси та кількість реплік для кожного мікросервісу.

Процес генерації застосунку у μBench складається з трьох основних етапів. Перший етап, Service Mesh Generator, відповідає за створення топології мережі мікросервісів на основі заданих параметрів або випадкових алгоритмів, таких як модель Барбаші-Альберта. Другий етап, Work Model Generator, визначає поведінку кожного мікросервісу, включаючи використовувані ресурси та взаємодію з іншими сервісами, з можливістю генерації поведінки як випадковим чином, так і на основі заданих шаблонів. Третій етап, Kubernetes Deployer, автоматизує розгортання згенерованих застосунків у Kubernetes-кластері, створюючи необхідні ресурси, такі як Deployment, Service та ConfigMap для кожного мікросервісу.

Після розгортання застосунок μBench функціонує таким чином: вхідні запити надходять від клієнтів через API-шлюз, який маршрутизує їх до відповідних мікросервісів.

Для моніторингу продуктивності застосунку μBench інтегрується з системами Prometheus, Grafana, Istio, Kiali та Jaeger, що дозволяє детально аналізувати метрики продуктивності та трасування взаємодій між сервісами. Аналіз продуктивності в μBench здійснюється двома основними підходами: випадкові навантаження (stochastic-driven benchmarks) та навантаження на основі трасувальних даних (trace-driven benchmarks). Перший підхід дозволяє

моделювати випадкові взаємодії між мікросервісами, направляючи запити випадковим чином, що імітує реалістичні сценарії використання. Другий підхід використовує реальні дані про послідовність викликів між мікросервісами.

Таким чином, µBench надає потужний інструментарій для створення та тестування мікросервісних архітектур, забезпечуючи гнучкість налаштувань та детальний моніторинг, що робить його цінним інструментом для дослідників, розробників хмарних платформ.

3.4 Порівняння ефективності монолітної та мікросервісної архітектури

На основі графіків можна зробити висновок про продуктивність монолітної архітектури відносно деревовидної та архітектури "ланцюг".

Архітектура "ланцюг" показує (див. рис. Рисунок 3.*1* – Графік порівняння продуктивності алгоритмів (Монолітна архітектура/Архітектура "ланцюг")) найкращу продуктивність для більшості алгоритмів і розмірів масивів, особливо при малих та середніх розмірах даних.

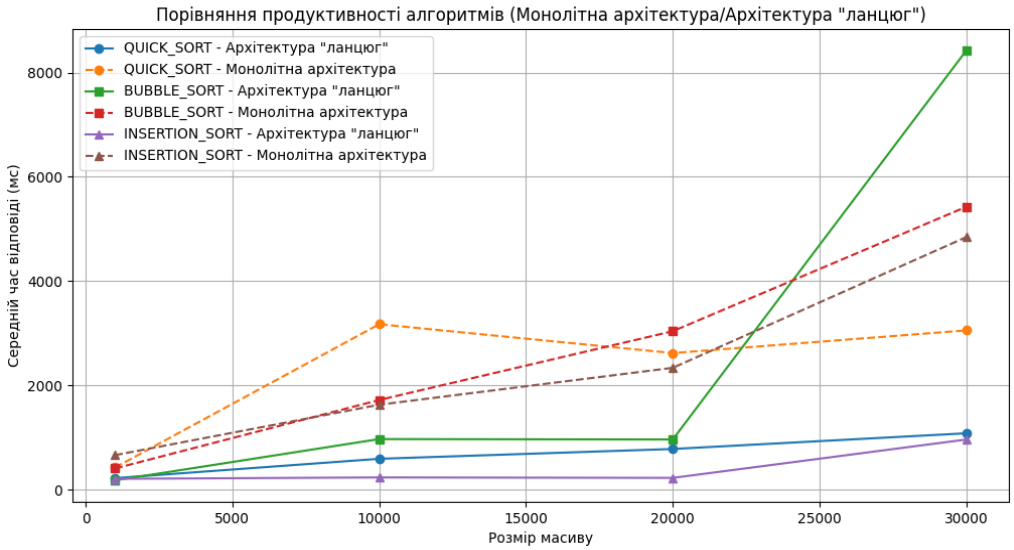


Рисунок 3.1 – Графік порівняння продуктивності алгоритмів (Монолітна архітектура/Архітектура "ланцюг")

Продуктивність монолітної архітектури знаходиться між деревовидною та архітектурою "ланцюг". Вона перевершує деревовидну архітектуру при менших розмірах масиву та поступається їй при більших. (див. рис.Рисунок 3.*2*)

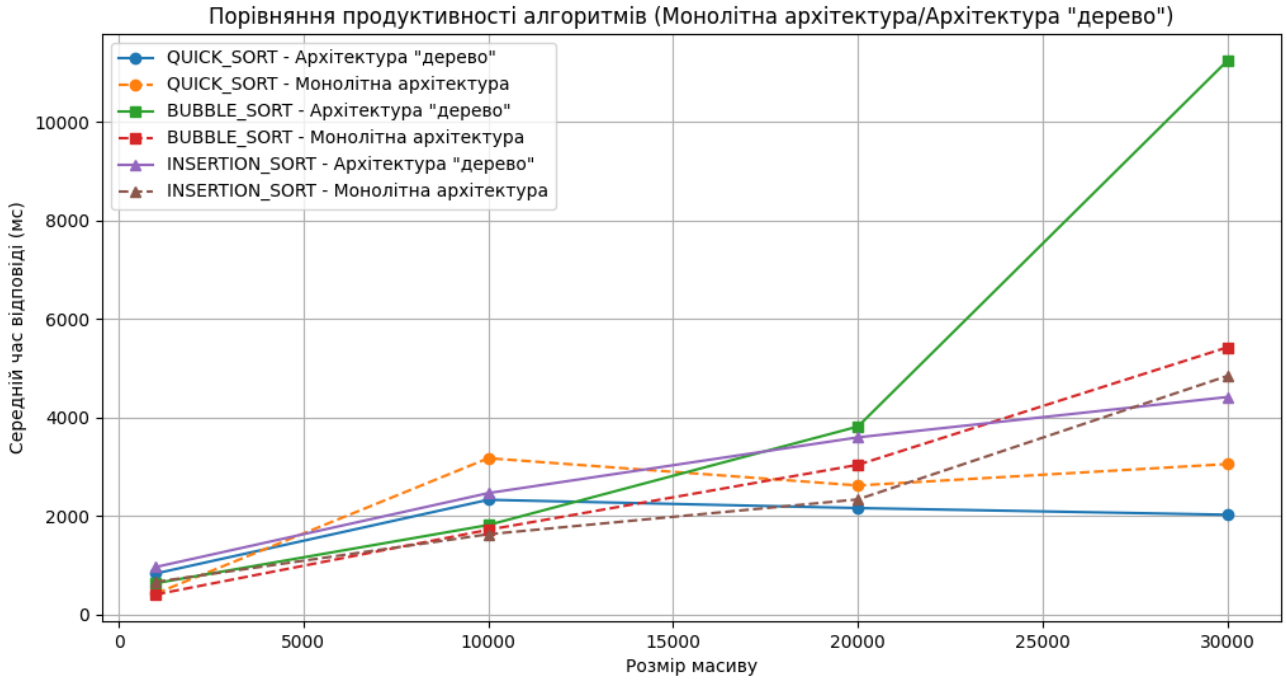


Рисунок 3.2 – Графік порівняння продуктивності алгоритмів (Монолітна архітектура/Архітектура "дерево")

Архітектура "ланцюг" показує найкращу продуктивність для більшості алгоритмів і розмірів масивів, особливо при малих та середніх розмірах даних.

Деревовидна архітектура має найгіршу продуктивність для великих масивів, особливо з алгоритмом BUBBLE_SORT.

3.5 Аналіз різних підходів до побудови мікросервісних систем

Проведений аналіз охоплював дослідження продуктивності трьох алгоритмів сортування—QUICK_SORT, BUBBLE_SORT та INSERTION_SORT—у двох різних архітектурних рішеннях: деревоподібній структурі та архітектурі на основі партерну "Ланцюг відповідальностей". Метою було визначити переваги та недоліки кожної архітектури при обробці

масивів різного розміру, а також встановити найбільш доцільні сфери застосування кожної моделі.

З графіку (див. рис.Рисунок 3.3) видно, що архітектура "дерево" показала стабільну продуктивність на більшості з масивів.

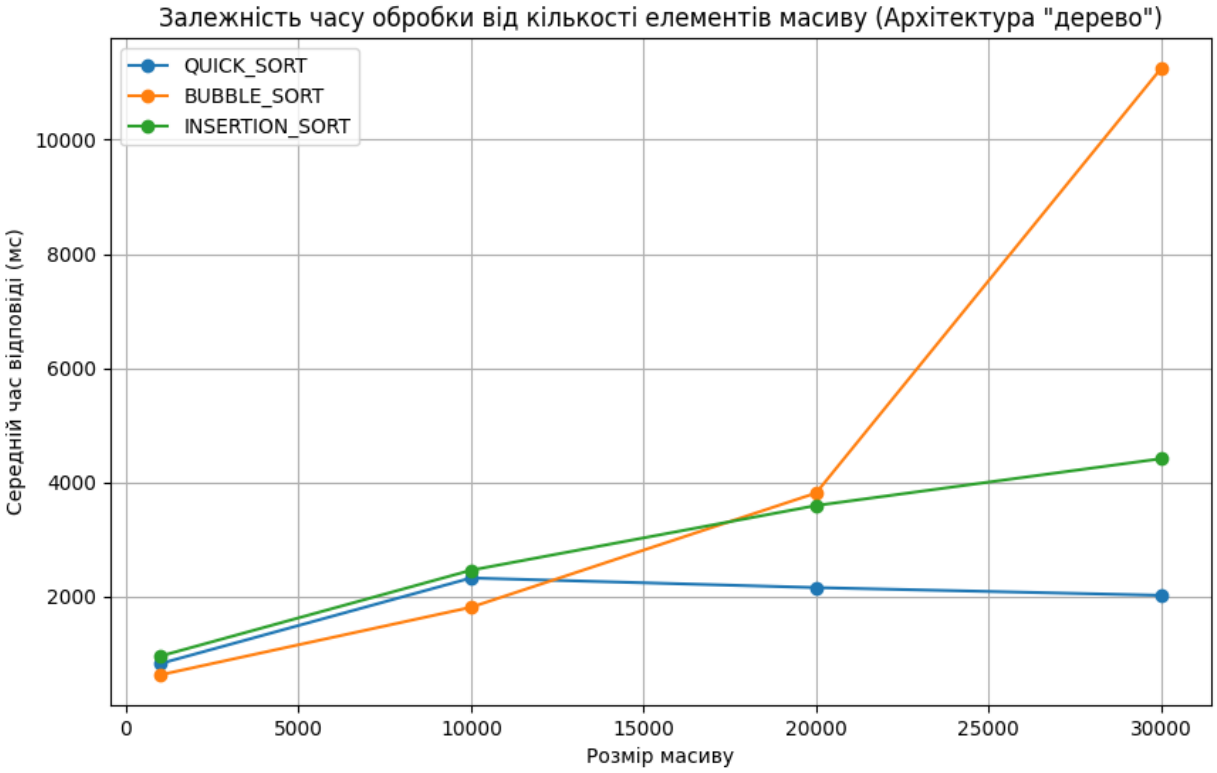


Рисунок 3.3 – Графік залежності часу обробки від кількості елементів масиву(Архітектура "дерево")

Архітектура "дерево" має переваги завдяки можливості паралелізації завдань, що може підвищити продуктивність при обробці великих обсягів даних. Масштабованість цієї архітектури дозволяє ефективно збільшувати обчислювальну потужність системи шляхом додавання нових вузлів. Крім того, структурованість даних у деревоподібній формі може бути корисною при обробці ієрархічних або рекурсивних структур.

Проте, в проведених тестах архітектура "дерево" показала нестабільну роботу, особливо для великих масивів. При роботі з ними запити в 19% випадків закінчувались помилкою. Це пов’язано зі складною структурою системи сервісів та можливістю додавання нових вузлів під час виконання запиту. Також можуть вимагатися додаткові ресурси для управління

паралельними процесами та синхронізацією між вузлами. В ситуації з дуже обмеженими ресурсами така архітектура є не ефективною.

Незважаючи на отримані результати про помилки при запитах, архітектура "дерево" має свої переваги та сфери застосування. При достатніх ресурсах вона підходить для обробки великих та складних даних, що вимагають роботи з ієрархічними структурами або виконання рекурсивних алгоритмів. В умовах ефективного управління паралелізмом "дерево" може забезпечити значне прискорення за рахунок одночасної роботи багатьох вузлів.

Архітектура "ланцюг" продемонструвала (див. рис.Рисунок 3.4**Помилка! Джерело посилання не знайдено.**) стабільне збільшення часу виконання зі зростанням розміру масиву, що відповідає теоретичній складності алгоритмів. Простота реалізації цієї архітектури спрощує управління потоком даних та синхронізацію між вузлами. Ефективне використання ресурсів досягається завдяки меншій складності комунікацій між вузлами.

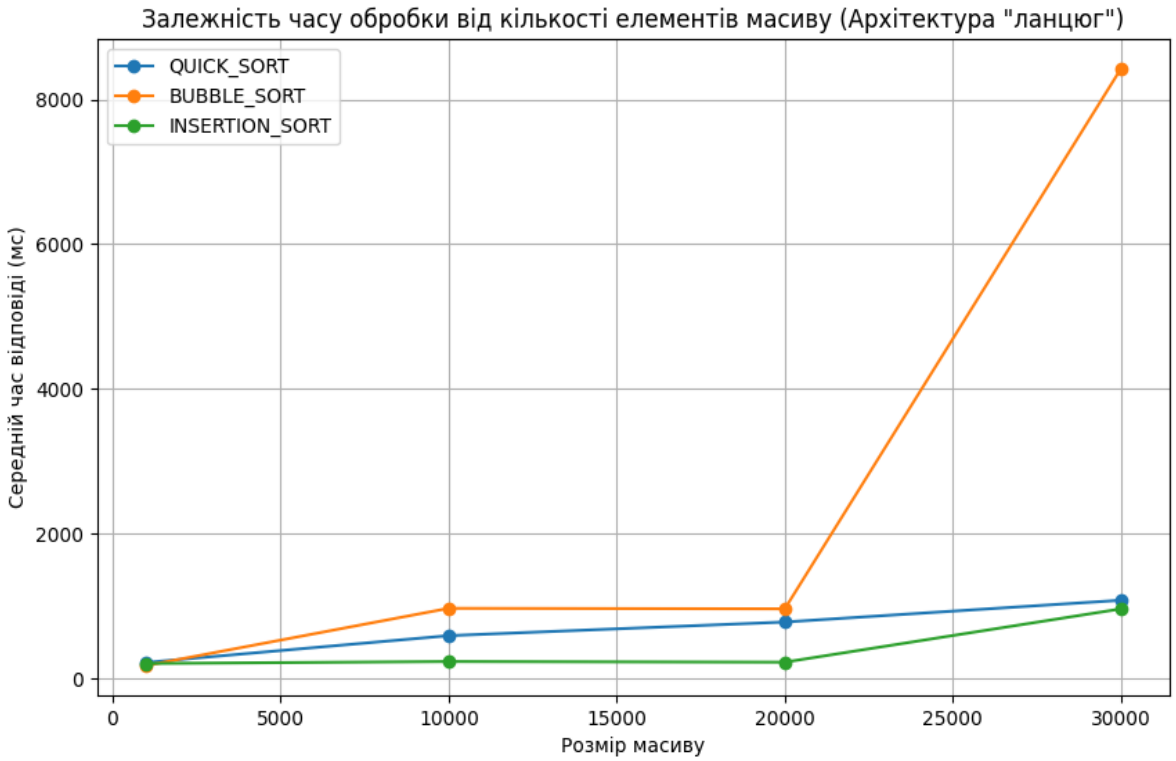


Рисунок 3.4 – Графік залежності часу обробки від кількості елементів масиву(Архітектура "ланцюг")

Однак, архітектура "ланцюг" має обмежені можливості для паралельного виконання завдань, що може стати вузьким місцем при обробці дуже великих обсягів даних. Потенційні затримки через послідовну обробку можуть впливати на загальну продуктивність системи, якщо один із вузлів працює повільно.

Така архітектура може вимагати збільшене споживання пам'яті через передачу повного масиву кожному сервісу, а не його частин. Вона ефективна при обмежених ресурсах або коли потрібно мінімізувати складність реалізації.

На графіку Рисунок 3.5) продемонстровані всі результати експеременту.

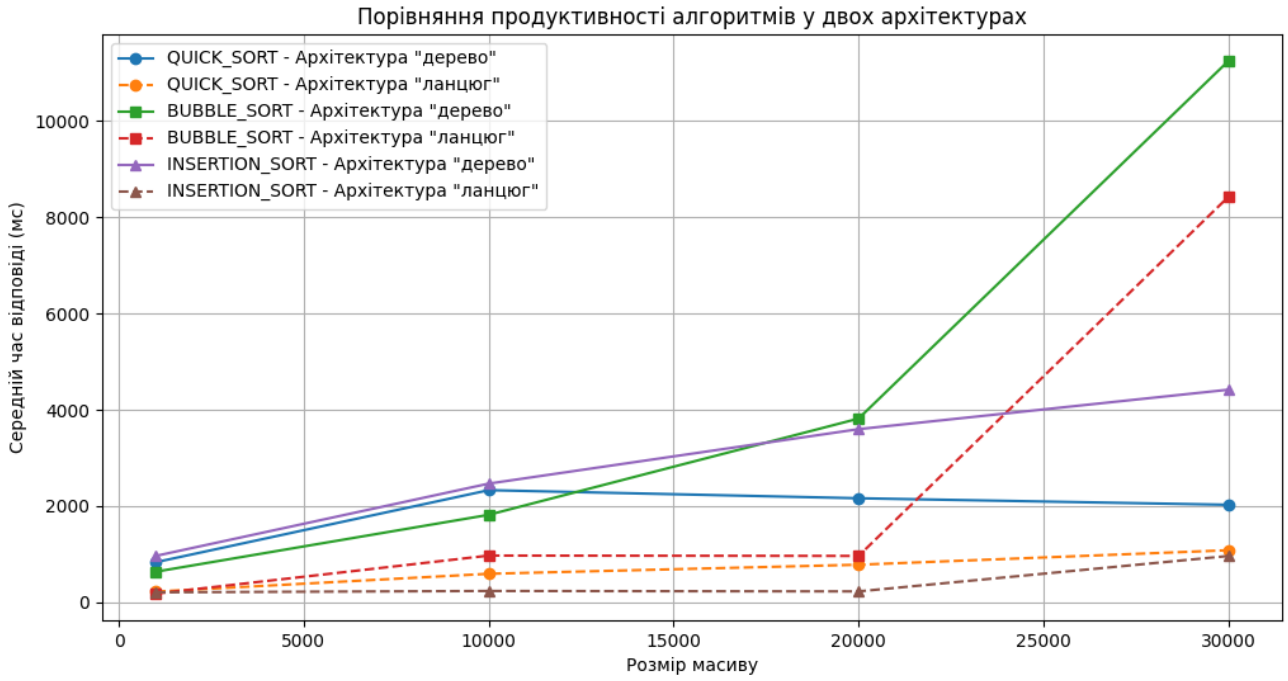


Рисунок 3.5 – Графік порівняння продуктивності архітектури "ланцюг" та "дерево"

Середній час виконання запитів для деревоподібної архітектури та архітектури на основі патерну “Ланцюг відповідальностей” (див. табл. Таблиця 3.1Таблиця 3.1 – Порівняльна таблиця

).

Таблиця 3.1 – Порівняльна таблиця

Розмір масиву	Quick Sort ("дерево"), час виконання, мс	Quick Sort ("ланцюг"), час виконання, мс	Bubble Sort ("дерево"), час виконання, мс	Bubble Sort ("ланцюг"), час виконання, мс	Insertion Sort ("дерево"), час виконання, мс)	Insertion Sort ("ланцюг"), час виконання, мс
1000	832.34	221.26	635.27	179.03	963.60	207.32
10000	2329.32	591.95	1817.07	969.82	2465.23	234.43
20000	2161.20	780.12	3816.53	962.50	3597.03	225.58
30000	2025.65	1081.17	11247.15	8424.19	4416.71	961.67

ВИСНОВКИ

У результаті виконання роботи було розроблено веб-додатки на основі різних архітектурних підходів та впроваджено методи оцінки продуктивності мікросервісної архітектури веб-додатків у хмарному середовищі. Особливу увагу приділено дослідженню різних архітектурних підходів, таких як ланцюгові та деревовидні структури, для оцінки їхньої ефективності та визначення сфер застосування.

Для моделювання потоку запитів використано скрипт у середовищі k6, який дозволяє імітувати вхідний потік запитів за моделлю масового обслуговування M/M/c. Це дозволило збирати ключові метрики, такі як час відповіді, коефіцієнт успішних запитів та кількість помилок, для подальшого аналізу продуктивності системи.

Дослідження архітектурних рішень виявило переваги та обмеження кожного підходу. Деревовидна архітектура показала високу ефективність при роботі з великими масивами даних завдяки можливості паралельного виконання завдань. Проте, складність синхронізації та підвищені витрати на управління паралельними процесами можуть призводити до нестабільності при обробці складних запитів. В умовах обмежених ресурсів ефективність цієї архітектури знижується, що виявлено під час тестування великих масивів, де в 19% випадків виникали помилки.

Архітектура "ланцюг" продемонструвала стабільність роботи, що підтверджується рівномірним збільшенням часу виконання зі зростанням розміру масиву. Її основна перевага полягає в простоті реалізації, яка зменшує ймовірність помилок та полегшує управління. Однак її обмежена здатність до паралельної обробки може впливати на продуктивність при обробці великих обсягів даних.

Результати роботи підтвердили, що мікросервісна архітектура є потужним інструментом для розробки веб-додатків, однак її ефективність значною мірою залежить від правильного вибору архітектурного підходу. Використання хмарних середовищ, таких як AWS, та інструментів автоматичного масштабування, таких як LoadBalancer і HPA, дозволяє покращити продуктивність системи та адаптувати її до змінних умов навантаження.

ПЕРЕЛІК ПОСИЛАНЬ

1. Newman S. Building Microservices. – Sebastopol, CA : O'Reilly Media, 2015. 280 с.

2. Литвинов А. Л. Теорія систем масового обслуговування : навч. посібник, Харків. нац. ун-т міськ. госп-ва ім. О. М. Бекетова. Харків : ХНУМГ ім. О. М. Бекетова. 2018. 141 с.

3. Kleinrock L. Queueing systems. Volume 1: Theory, New York : Wiley, 1975. 417 p.

4. Docker. Огляд Docker. URL: https://docs.docker.com/get-started/docker-overview/ (дата звернення: 02.09.2024).

5. Kubernetes. Документація. URL : https://kubernetes.io/docs/home/ (дата звернення: 02.09.2024).

6. Amazon Web Services. Що таке AWS. URL : https://aws.amazon.com/what-is-aws/ (дата звернення: 02.09.2024).

7. Amazon Web Services. Amazon Elastic Compute Cloud (EC2) URL : https://aws.amazon.com/ec2/ (дата звернення: 02.09.2024).

8. Amazon Web Services. Amazon Elastic Kubernetes Service (EKS) URL : https://aws.amazon.com/eks/ (дата звернення: 02.09.2024).

9. Amazon Web Services. Amazon Elastic Load Balancing. URL : https://aws.amazon.com/elasticloadbalancing/ (дата звернення: 02.09.2024).

10. Kubernetes Documentation. Horizontal Pod Autoscaler. URL : https://kubernetes.io/docs/tasks/run-application/horizontal-pod-autoscale/ (дата звернення: 02.09.2024).

11. Gamma E., Helm R., Johnson R., Vlissides J. Design Patterns: Elements of Reusable Object-Oriented Software, 1-е вид. Boston : Addison-Wesley Professional, 1995. 396 с.

12. Engel T., Langermeier M., Bauer B., Hofmann A. Evaluation of Microservice Architectures: A Metric and Tool-Based Approach. В кн.: Mendling J., Mouratidis H. (ред.) Information Systems in the Big Data Era. CAiSE 2018. Lecture Notes in Business Information Processing. Springer, Cham, 2018. С. 73–88.

13. Grafana Labs. k6 Documentation URL : https://grafana.com/docs/k6 /latest/ (дата звернення: 02.09.2024)

14. Detti A., Funari L., Petrucci L. μBench: An Open-Source Factory of Benchmark Microservice Applications, IEEE Transactions on Parallel and Distributed Systems. 2023. С. 968–980.