

МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ**Національний університет кораблебудування імені адмірала Макарова**Навчально-науковий інститут комп'ютерних наук та управління проектами
Кафедра інформаційних управляючих систем та технологій

"Допущений до захисту"

Завідувач кафедри ІУСТ



к.т.н, доц. Михелєв І.Л.

" ____ " _____ 2024 р.

КВАЛІФІКАЦІЙНА РОБОТА**на здобуття ступеня вищої освіти "магістр"****на тему:** «Дослідження архітектури та розробка моделей оптимізації
високонавантаженого сервісу»

Виконав: студент групи 6141 м



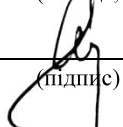
(підпис)

Чопенко М.С.

Керівник роботи:

доцент кафедри ІУСТ, к.т.н.,

(посада, науковий ступінь вчене звання)



(підпис)

Гайда А. Ю.

м. Миколаїв – 2024 р.

МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ

Національний університет кораблебудування імені адмірала Макарова
 Навчально-науковий інститут комп'ютерних наук та управління проектами
 Кафедра інформаційних управляючих систем та технологій
 Спеціальність 122 "Комп'ютерні науки"
 Освітня програма "Комп'ютерні науки"

"ЗАТВЕРДЖУЮ"

Гарант освітньої програми

В. Ажищев

к.т.н, доц. Ажищев В.Ф.

" ____ " ____ 2024 р.

ЗАВДАННЯ НА КВАЛІФІКАЦІЙНУ РОБОТУ на здобуття ступеня вищої освіти "магістр"

Студенту Чопенко Микиті Станіславовичу

(прізвище, ім'я, по батькові)

1. Тема роботи: «Дослідження архітектури та розробка моделей оптимізації високонавантаженого сервісу»

Керівник роботи Гайда А. Ю.

Затверджені наказом ректора № 1055-уч. від "08" жовтня 2024 року.

2. Термін подання роботи: 18 грудня 2024 року

3. Вихідні дані до проекту (роботи) ДСТУ щодо обробки інформації, літературні джерела, технічна документація на існуючі аналоги систем, матеріали наукового стажування.

4. Перелік питань, що належать до розробки (найменування розділів) Зміст; Перелік умовних позначень, символів, скорочень та термінів; Вступ; Дослідження існуючих архітектур високонавантажених систем; Моделі оптимізації високонавантаженого сервісу; Розробка моделей оптимізації високонавантаженого сервісу; Висновки; Список використаних джерел; Додаток (код програми).

5. Перелік презентаційних матеріалів 1. Актуальність та мета роботи; 2. Визначення та сфери використання високонавантажених систем; 3. Основні проблеми високонавантажених систем; 4. Аналіз архітектури високонавантажених систем; 5. Математична модель вибору найкращої архітектури (метод TOPSIS); 6. Експертні оцінки та результати вибору найкращої архітектури; 7. Масштабування сервісів: горизонтальне та вертикальне 8. Критерії та вагові коефіцієнти для вибору масштабування; 9.

Кешування: Redis, Memcached, DragonflyDB; 10. Математична модель вибору технології кешування методом (АНР) аналізу ієрархій; 11. Результати вибору технології кешування методом (АНР); 12. Модель розгортання; 13. Модель оркестрації; 14. Приклади конфігурації оркестратора Kubernetes та результати партиціювання; 15. Висновки.

6. Консультанти розділів роботи

Розділ	Прізвище, ініціали та посада консультанта	Підпис, дата	
		завдання видав	завдання прийняв
1	Гайда А. Ю.	12.10.2024	07.11.2024
2	Гайда А. Ю.	07.11.2024	17.11.2024
3	Гайда А. Ю.	17.11.2024	11.12.2024

7. Дата видачі завдання 10 жовтня 2024 р.

КАЛЕНДАРНИЙ ПЛАН

№ з/п	Назва етапів дипломного проекту (роботи)	Строк виконання етапів роботи	Примітка
1	Отримання завдання на дипломну роботу та побудова плану-графіку виконання робіт.	12.10.2024 – 15.10.2024	
2	Огляд та аналіз архітектур високонавантажених сервісів	16.10.2024 – 20.10.2024	
3	Написання Розділу 1 дипломної роботи.	20.10.2024 – 07.11.2024	
4	Огляд та аналіз моделей оптимізації високонавантажених сервісів Написання Розділу 2 дипломної роботи	07.11.2024 – 15.11.2024	
5	Написання Розділу 3 дипломної роботи. Завершення створення пояснювальної записки дипломної роботи.	15.11.2024 – 01.12.2024	
6	Оформлення пояснювальної записки.	02.12.2024 – 11.12.2024	

Студент

(підпис)

Керівник роботи

(підпис)

Чопенко М.С.

(ПІБ)

Гайда А. Ю.

(ПІБ)

АНОТАЦІЯ

Магістерська робота присвячена дослідженню архітектурних підходів та розробці моделей оптимізації високонавантажених сервісів. Метою роботи є створення ефективних та продуктивних програмних рішень, які забезпечують стабільну роботу сервісів при великій кількості користувачів та запитів у реальному часі.

У роботі представлено результати аналізу ключових архітектурних підходів, досліджено сучасні моделі оптимізації, такі як партиціювання даних, балансування навантаження, масштабованість, а також представлено моделі оптимізації продуктивності сервісів. Практичну апробацію отримала розроблена модель оптимізації SQL-запитів на рівні СКБД, яка довела свою ефективність у продуктовому середовищі.

Результати дослідження можуть бути використані для створення високонавантажених сервісів або вдосконалення існуючих. Запропоновані моделі оптимізації сприяють надійності та масштабованості сервісів.

Магістерська робота викладена на 98 сторінках друкованого тексту, містить 30 рисунків, 16 таблиць, список використаних джерел із 20 найменувань, 1 додаток.

Ключові слова: Високонавантажені системи, архітектура систем, монолітна архітектура, подійно-орієнтована архітектура, мікросервісна архітектура, безсерверна архітектура, сервіс-орієнтована архітектура, багатоваршівна архітектура, моделі оптимізації, оптимізація, масштабування, кешування, партиціювання, сервіси.

ABSTRACT

The master's thesis is devoted to the study of architectural approaches and the development of optimization models for highly loaded services. The aim of the work is to create effective and productive software solutions that ensure stable operation of services with a large number of users and real-time queries.

The work presents the results of the analysis of key architectural approaches, investigates modern optimization models, such as data partitioning, load balancing, scalability, and also presents models for optimizing service performance. The developed model for optimizing SQL queries at the DBMS level has been tested in practice, which has proven its effectiveness in a product environment.

The results of the study can be used to create highly loaded services or improve existing ones. The proposed optimization models contribute to the reliability and scalability of services.

The master's thesis is presented on 98 pages of printed text, contains 30 figures, 16 tables, a list of sources used with 20 names, and 1 appendix.

Keywords: High-load systems, system architecture, monolithic architecture, event-driven architecture, microservices architecture, serverless architecture, service-oriented architecture, multi-tier architecture, optimization models, optimization, scaling, caching, partitioning, services.

ЗМІСТ

ПЕРЕЛІК УМОВНИХ ПОЗНАЧЕНЬ, СИМВОЛІВ, СКОРОЧЕНЬ І ТЕРМІНІВ	8
ВСТУП	10
РОЗДІЛ 1. ДОСЛІДЖЕННЯ ІСНУЮЧИХ АРХІТЕКТУР	
ВИСОКОНАВАНТАЖЕНИХ СИСТЕМ	12
1.1. Поняття високонавантажених систем	12
1.2. Задачі та виклики, які вирішують високонавантажені системи	14
1.3. Аналіз сфер використання високонавантажених систем	17
1.4. Огляд та аналіз існуючих результатів досліджень високонавантажених систем	19
1.5. Основні проблеми високонавантажених систем	21
1.6 Аналіз архітектури високонавантажених систем	22
1.6.1. Визначення архітектури систем	22
1.6.2. Монолітна архітектура	25
1.6.3. Подійно-орієнтована архітектура.....	28
1.6.4. Мікросервісна архітектура.....	32
1.6.5. Безсерверна архітектура.....	35
1.6.6. Сервіс-орієнтована архітектура.....	38
1.6.7. Багатошарова архітектура.....	41
1.7. Порівняння архітектури за ключовими характеристиками	44
1.8. Постановка задачі дослідження.....	50
1.9. Висновки до розділу 1	51
РОЗДІЛ 2. МОДЕЛІ ОПТИМІЗАЦІЇ ВИСОКОНАВАНТАЖЕНОГО СЕРВІСУ	
.....	53

	7
2.1. Оптимізаційна модель	53
2.2. Масштабування сервісів: горизонтальне та вертикальне	54
2.3 Кешування: Redis, Memcached, DragonflyDB	59
2.4. Партиціювання на рівні СКБД PostgreSQL	67
2.5. Висновки до розділу 2.....	75
РОЗДІЛ 3. РОЗРОБКА МОДЕЛЕЙ ОПТИМІЗАЦІЇ	
ВИСОКОНАВАНТАЖЕНОГО СЕРВІСУ	77
3.1 Проектування та реалізація мікросервісної архітектури сервісу	77
3.2 Кешування сервісу з використанням технології Redis	80
3.3. Контейнеризація та оркестрація сервісу з використанням Docker та Kubernetes для масштабування сервісів.....	82
3.4 Висновки до розділу 3	92
ВИСНОВКИ.....	94
СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ	95
ДОДАТОК А КОД ПРОГРАМИ	97

ПЕРЕЛІК УМОВНИХ ПОЗНАЧЕНЬ, СИМВОЛІВ, СКОРОЧЕНЬ І ТЕРМІНІВ

СКБД – Система керування базами даних.

АНР – Аналіз ієрархій рішень.

РСКБД – Реляційна система керування базами даних.

БД – База даних.

IoT – Internet of Things (інтернет речі).

API – Application Programming Interface (інтерфейс програмування застосунків).

SQL – Structured Query Language (мова структурованих запитів).

HTML – HyperText Markup Language (мова розмітки гіпертексту).

SPA – Single Page Application (односторінковий додаток).

EDA – Event-Driven Architecture (подійно-орієнтована архітектура).

SOA – Service-Oriented Architecture (сервісно-орієнтована архітектура).

MVC – Model-View-Controller (шаблон модель-вид-контролер).

CDN – Content Delivery Network (мережа доставки контенту).

ERP – Enterprise Resource Planning (планування ресурсів підприємства).

FinTech – Financial Technology (фінансові технології).

E-commerce – Electronic Commerce (електронна комерція).

HTTP – HyperText Transfer Protocol (протокол передачі гіпертексту).

CSS – Cascading Style Sheets (каскадні таблиці стилів).

CPU – Central Processing Unit (центральний процесор).

RAM – Random Access Memory (оперативна пам'ять).

SpoF – Single Point of Failure (єдина точка відмови).

IT – Information Technology (інформаційні технології).

DNS – Domain Name System (система доменних імен).

CRUD – Create, Read, Update, Delete (функції управління даними «створення, читання, оновлення і видалення»).

ORM – Object-Relational Mapping (об'єктно-реляційне відображення).

JWT – JSON Web Token (стандарт відкритого протоколу для передачі даних).

ВСТУП

З розвитком цифрових технологій виникла потреба у створенні надійних та продуктивних високонавантажених сервісів, які здатні опрацьовувати велику кількість користувачів, обробляти великі обсяги даних та швидко відповідати на запити в реальному часі. Успішне функціонування пошукових сервісів, соціальних мереж, поштових скриньок, фінансових сервісів, електронної комерції вимагає стабільної та масштабованої архітектури, яка повинна давати можливість відповідати на мільйони запитів за невеликий проміжок часу, без збоїв та втрати продуктивності. Проблеми в таких системах можуть призвести до значних фінансових втрат, оскільки навіть короточасні перерви у роботі або зниження продуктивності можуть негативно впливати на доходи компаній.

В основі роботи будь-якої високонавантаженої системи є архітектурні рішення, які забезпечують розподілену обробку запитів, балансування навантаження, шардування даних та оптимізацію ресурсів. Проте вибір і розробка ефективних моделей оптимізації для таких систем є складним завданням, який вимагає дослідження і ретельного аналізу існуючих рішень та підходів. Від правильного вибору архітектури програмного забезпечення, а також вибір сучасних моделей оптимізації, залежить, чи буде система успішно функціонувати та приносити фінансові доходи компаніям.

Актуальність роботи. Дослідження архітектури та розробка моделей оптимізації високонавантаженого сервісу є актуальними через зростаючі вимоги до продуктивності, масштабованості та надійності сучасних систем.

Мета роботи. Підвищення ефективності високонавантажених автоматизованих інформаційних систем за рахунок впровадження сучасних моделей та технологій створення стабільних і продуктивних сервісів.

Завдання дослідження.

1. Визначити ключові архітектурні підходи для високонавантажених сервісів.

2. Дослідити та розробити моделі оптимізації, спрямовані на мінімізацію затримок і підвищення ефективності роботи високонавантажених сервісів.

Об'єкт дослідження. Високонавантажені автоматизовані інформаційні системи, які обробляють дані у реальному часі.

Предмет дослідження: архітектурні шаблони та моделі автоматизованих інформаційних систем.

Апробація результатів дослідження. Одна з розроблених моделей оптимізації, спрямована на підвищення швидкості виконання SQL-запитів шляхом використання партиціювань на рівні СКБД, підтвердила свою практичну ефективність у продуктовому середовищі. Результати цього дослідження були представлені під час наукової конференції:

V Всеукраїнська наукова-практична інтернет-конференція «Інформаційні технології: моделі, алгоритми, системи», м. Миколаїв, 30-31 жовтня 2024 року.

Ця магістерська робота присвячена дослідженню різних архітектурних підходів для створення стабільних і продуктивних високонавантажених систем, а також розробці моделей оптимізації, які дозволять підвищити продуктивність таких систем.

У роботі буде розглянуто використання моделей оптимізації такі як: масштабованість, оптимізація запитів до сховищ даних та шаблони проектування сервісів. У межах дослідження буде проаналізовано переваги та недоліки існуючих архітектурних шаблонів проектування, а також можливості вдосконалення підходів до оптимізації. Дослідження дасть можливість глибше зрозуміти загальні проблеми та можливості при застосуванні моделей оптимізації для навантаженого сервісу.

Отримані результати будуть цінними для організацій, які планують побудувати високонавантажену систему з великою кількістю користувачів або для тих, які хочуть впровадити моделі оптимізації для існуючої системи для більш надійності та продуктивності.

РОЗДІЛ 1. ДОСЛІДЖЕННЯ ІСНУЮЧИХ АРХІТЕКТУР ВИСОКОНАВАНТАЖЕНИХ СИСТЕМ

1.1. Поняття високонавантажених систем

Зі зростанням популярності цифрових технологій, великі компанії та організації зіткнулися з проблемами доступності їх сервісів для великої кількості користувачів. Це спричинило підвищені вимоги до продуктивності, надійності та масштабованості системи. Раніше при збільшенні одночасних звернень до ресурсу, призводило до частих збоїв, затримок у роботі додатків і нестабільності, що впливало на незадоволеність користувачів і репутацію компаній. Щоб подолати ці виклики організації та компанії вирішили проектувати та впроваджувати архітектурні рішення, звідси і з'явився термін: високе навантаження системи.

Спершу розглянемо одну просту аксіому: високі навантаження – це відносне поняття. Їх не можна виміряти кількістю запитів, які надходять на сервер. Адже не існує певної визначеної величини кількості запитів, бо одна система зможе без проблем обробити тисячі запитів за секунду, так як інша дасть збої при малій кількості звернень.

Високонавантажені системи – це системи, спеціально оптимізовані для обробки великої кількості запитів або транзакцій у режимі реального часу. Вони зазвичай обслуговують значну кількість користувачів або обробляють великий обсяг даних. Вони потребують відповідні архітектурні рішення і технології для забезпечення стабільної та швидкої роботи системи.

Основні риси високонавантажених систем:

- Система з великою аудиторією: високонавантажена система обслуговує величезну кількість користувачів. Для веб-застосунків це тисячі й навіть сотні тисяч одночасних запитів.

– Розподілена система: обробка великих даних вимагає наявності численних серверів. Високонавантажені застосунки, як-от Google або Facebook, працюють на сотнях серверів.

– Система з постійним ростом користувачів: високонавантажені системи часто мають зростаючу аудиторію, оскільки вони надають користувачам цінні послуги.

– Інтерактивна система: користувачі високонавантажених систем очікують негайної відповіді на свої дії, при пошуку інформації або здійсненні покупок.

– Система з використанням великих ресурсів: для забезпечення швидкої відповіді високонавантажені системи, витрачають значні ресурси, такі як: процесорна потужність, оперативна пам'ять і дисковий простір. [1]

Ці характеристики окреслюють основні технічні та функціональні вимоги до високонавантажених систем, визначаючи необхідність особливого підходу до їх розробки, масштабування та підтримки.

Раніше високонавантажені системи зазвичай будувались в якості одного сервісу, тобто функції і компоненти системи були об'єднанні і працювали на одному або декількох потужних серверів. Цей підхід працював, але кількість онлайн користувачів постійно збільшувалось, очевидно також збільшувався обсяг даних та почало виникати обмеження системи, а також постійні затримки при отриманні даних.

Приклад системи, яка вміщає собі візуальну, функціональну частину зображено на рисунку 1.1.

Щоб досягти високої продуктивності та масштабованості такі системи часто використовують розподілені обчислення й серверні кластери, які допомагають розділити навантаження та забезпечують надійність системи. Приклад архітектури сучасної високонавантаженої системи зображений на рисунку 1.2

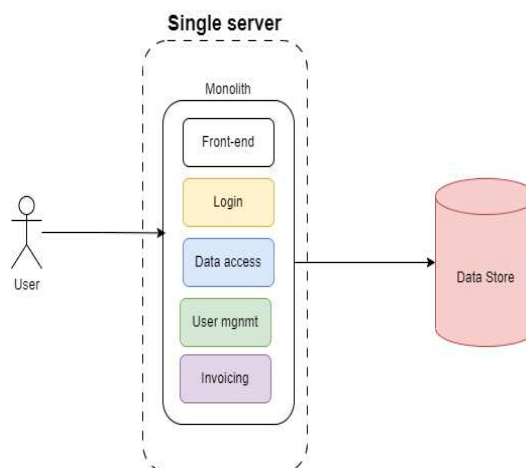


Рис 1.1 Сервіс з декількома складовими системи

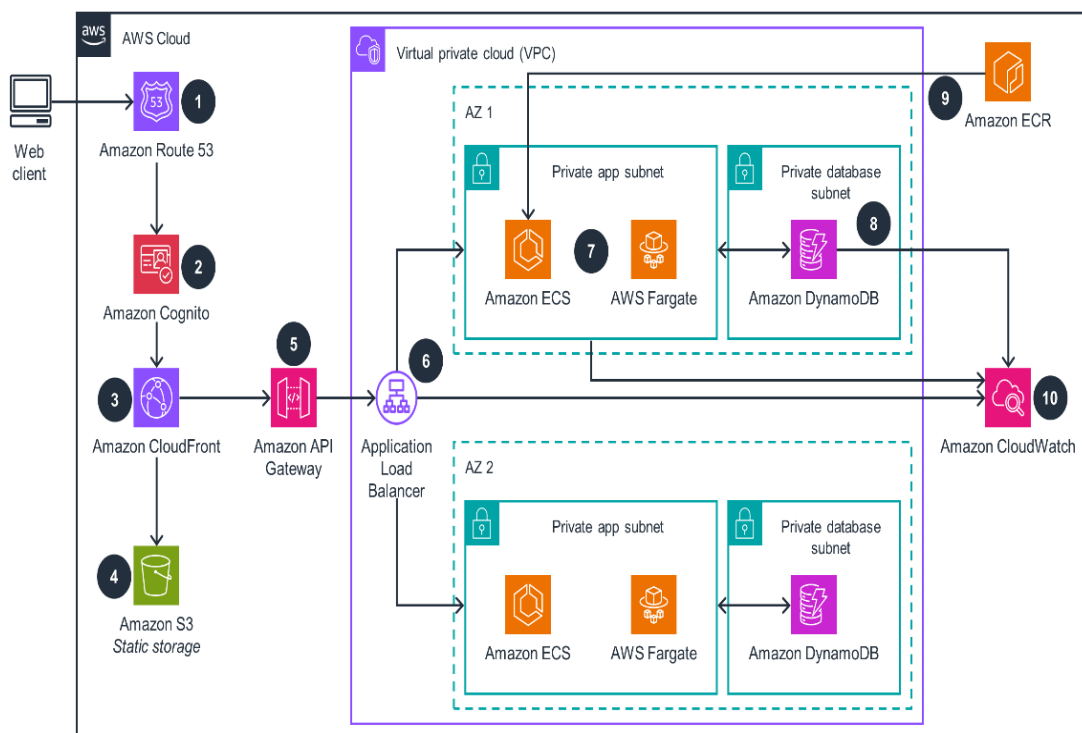


Рис 1.2 Сучасна архітектура високонавантаженого сервісу

1.2. Задачі та виклики, які вирішують високонавантажені системи

Високонавантажені системи можуть вирішувати задачі, які пов'язані з обробкою великих обсягів даних, забезпеченням стабільної роботи в умовах високого навантаження та гарантуванням швидкої відповіді користувачам. Розглянемо основні задачі, які стоять для таких систем.

- Масштабування: Масштабування системи в залежності від поточного навантаження. Використання технологій горизонтального масштабування, тобто додавання серверів або вертикальне для збільшення потужності серверів.
- Висока доступність: Забезпечення доступності системи без простоїв, навіть під час оновлень системи.
- Обробка великих обсягів даних: Такі системи повинні підтримувати значні обсяги даних та забезпечувати швидкий доступ до них. На даний час системи використовують масштабовані системи керування базами даних, які підтримують такі технології як партиціювання, реплікація та шардування.
- Безпека та захист даних: Високонавантажені системи дуже часто стають жертвами DDOS-атак, тому захист від таких загроз є важливою задачею. Архітектура системи дозволяє фільтрувати та виявляти аномальний трафік та працювати без збоїв та пікових навантажень.
- Моніторинг та управління системою: Система збирає та використовує метрики доступності системи та помилки.
- Розподіл навантаження: Система повинна розподіляти запити між різними серверами, для того щоб уникнути перенавантаження окремих ресурсів і забезпечити стабільну та надійну роботу системи.
- Підтримка транзакційної обробки: Для фінансових, банківських і комерційних проектів, необхідна чітка послідовність виконання операцій. Це дуже важливо, так як втрата однієї транзакції може слугувати фінансовим втратам.
- Розподілена інфраструктура по всьому світу: Використання різних дата центрів по всьому світу, дозволяє знизити затримки для користувачів та збільшувати швидкість доступу до послуг.
- Мінімізація витрат на обслуговування: Автоматизація і розподіл навантаження, допомагають знизити витрати на обслуговування сервісів, при цьому зберігають якість обслуговування.

– Обробка подій у реальному часі: Високонавантажені системи повинні забезпечувати миттєву відповідь та інформування користувачів про важливі події. Прикладом можуть бути повідомлення у соціальних мережах або фінансові транзакції.

– Реалізація та управління транзакційними чергами: Використання черг дозволяє в системі організувати обробку запитів, зберігаючи стабільність при великій кількості одночасних запитів. Приклад схеми обробки черги повідомлень зображені на рисунку 1.3.

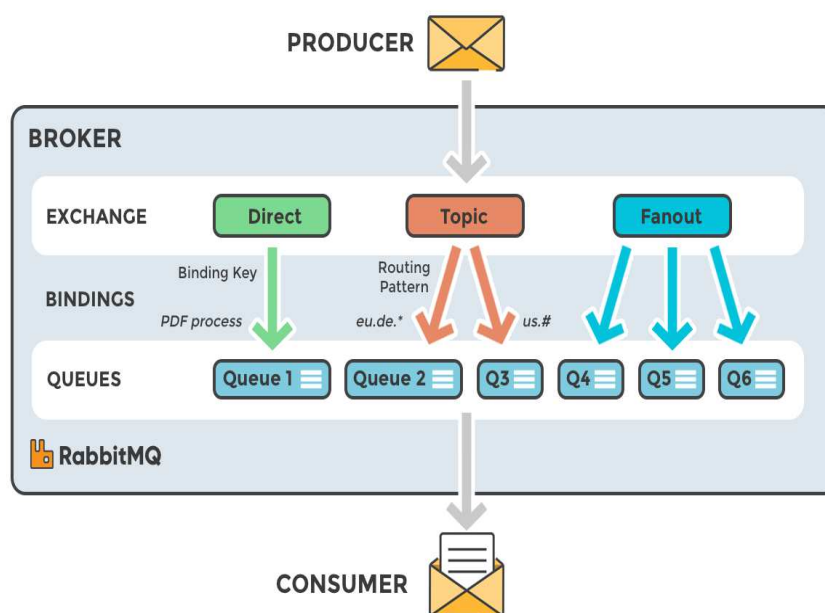


Рис 1.3 Принцип роботи черги повідомлень

Можна зробити висновок, що високонавантажені системи відіграють ключову роль у сучасних великих продуктивних системах. Вони вирішують великий спектр задач – обробка великих обсягів інформації, забезпечення високої доступності надійності та безпеки. Для забезпечення стабільної роботи системи під високим навантаженням, великі організації та компанії будують різні архітектурні підходи. Вибір архітектури системи в майбутньому може впливати на продуктивність та гнучкість системи, а також інтеграції з іншими сервісами у подальшому, адже технології розвиваються з кожним днем та потребують все нових та нових рішень.

1.3. Аналіз сфер використання високонавантажених систем

На сьогодні високонавантажені системи охоплюють різні сфери діяльності й використовуються для обробки великих обсягів даних, а саме тисячі транзакцій і запитів від користувачів. Розглянемо деякі категорії сфер використання сучасних високонавантажених систем.

- Соціальні мережі. Ці платформи обслуговують мільярди користувачів, які щодня генерують величезну кількість контенту – відео, зображення. Система повинна швидко показувати актуальні дані для комфортного використання соціальної мережі. Прикладом таких систем є: Twitter, Facebook, Instagram.

- Торгівля в Інтернеті. Онлайн-магазини обробляють мільйони транзакцій на секунду, особливо під час великих знижок. Вони повинні підтримувати завжди наявність товару, обробляти платіжні транзакції, забезпечувати швидкий пошук товарів на сторінці за допомогою різноманітних фільтрів. Затримки і збої під час пікових навантажень загрожує значним фінансовим втратам. Прикладом є: Rozetka, Comfy, Amazon.

- Сервіси онлайн-трансляцій. Вони забезпечують миттєву трансляцію відео та аудіо для мільйонів користувачів у різних країнах. В якості технологій вони використовують доставку даних (CDN), для того щоб передавати дані з мінімальною затримкою. До таких систем відносяться: Netflix, YouTube, Spotify.

- Платіжні системи та фінансові сервіси. Платіжні системи обробляють величезну кількість транзакцій у режимі реального часу, забезпечуючи обмін коштами по всьому світу. Надійність таких систем є критичною, оскільки навіть короточасні збої зможуть призвести до великих фінансових втрат для мільйонів клієнтів. Прикладом є: Privatbank, Monobank, PayPal, Stripe.

- Сервіси електронної пошти. Ці системи щохвилини обробляють мільярди повідомлень. Їх задача швидко доставити листи користувачу, а також фільтрувати листи від спаму. Прикладом є: Gmail, Outlook, Yahoo.

- Медичні інформаційні системи. Медичні інформаційні системи обробляють великий обсяг даних про пацієнтів. Вони надають медичним працівникам доступ до медичних карт і так далі. До таких систем відносяться: Epic, Cerner.
- Системи бронювання та управління квитками. Системи бронювання дозволяють обслуговувати мільйони користувачів, які одночасно переглядають доступні варіанти квитків на літак або потяг. Прикладом є Booking.com, Airbnb.
- Хмарні платформи. Хмарні сервіси дозволяють організаціям та компаніям використовувати серверні потужності у будь-якій частині світу. Хмарні платформи надають інфраструктуру для високонавантажених додатків. До таких систем можна віднести наступні: Microsoft Azure, Google Cloud Platform, Amazon Web Services, Cloudflare. На рисунку 1.4 зображено карту серверів, які дозволяють фільтрувати високонавантажений трафік сервісів.



Рис 1.4 Глобальна мережа дата центрів Cloudflare

З плином часу, для розвитку високонавантажених систем було створено чимало технологій, які допомагають підтримувати їхню швидку роботу. До них відноситься: балансування навантаження, кешування, шардування, партиціювання, шаблони проектування архітектури.

Для задоволення вимог високонавантажених систем, необхідно проектувати та оптимізувати їх таким чином, щоб забезпечити максимальну

ефективність, масштабованість і доступність. Такі системи є надзвичайно важливими у сучасному світі, дозволяючи великим організаціям та компаніям ефективно обробляти великі обсяги даних, трафіку та транзакцій, задовольняючи усі потреби користувачів у сучасному. На основі описів сфер використання, можна визначити, що високонавантажені системи повинні володіти характеристиками масштабованості та ефективним управлінням ресурсами.

1.4. Огляд та аналіз існуючих результатів досліджень високонавантажених систем

Дослідження в області архітектури систем, підкреслюють важливість проектування архітектури на етапі створення системи, оскільки саме від цього етапу залежить здатність системи справлятися з високими навантаженнями та адаптуватися до нових вимог в майбутньому.

Автор Вудс Е. дослідив еволюцію програмної архітектури, як вона розвивалась та з якими потребами та викликами програмних систем стикалась. Сьогодні архітектурна практика більше інтегрована в основу створення програмних систем, ніж раніше, оскільки від цього залежить майбутнє проекту. Він також передбачає, що наступна фаза еволюції буде ще більш значущою, з акцентом на інтелектуальні системи та розгортання на хмарних платформах. Ці нові виклики вимагатимуть ще одного етапу розвитку архітектурної практики, відкриваючи перспективи для архітекторів, готових до інновацій та адаптації в умовах постійних змін.[2]

У своїй роботі Мурабет Е. та Абтой А. дослідили різні методи валідації архітектури систем та здійснили аналіз їхніх сильних та слабких сторін. Вони наголосили, що перевірка архітектури є критично важливим етапом у процесі розробки програмного забезпечення, оскільки допомагає забезпечити відповідність архітектури високим вимогам системи та її високу якість. Залежно від специфіки проекту, таких як бюджет, строки та складність системи, певні

методи валідації можуть бути більш придатними. Дослідники також зазначають, що на практиці для повної оцінки якості архітектури часто використовуються декілька підходів до валідацій одночасно. В дослідженні представлено огляд трьох основних типів валідації програмної архітектури, що дозволяє визначити оптимальний метод для конкретного випадку та мінімізувати ризики та збитки, які пов'язані з помилками в програмному забезпеченні. [3]

Дослідник Гольдінер Д. проаналізував підходи до створення архітектури програмного забезпечення, призначеного для моделювання системи масового обслуговування. У результаті аргументованого аналізу було прийнято про доцільність такого проектування. Для розв'язання поставленого завдання було створено загальну діаграму алгоритму обробки заявок та проектну модель, яка охоплює 4 рівні (контекст, компоненти, контейнери та код). В цій роботі було вперше запропоновано архітектуру програмних систем для систем масового обслуговування із застосуванням паралельних обчислень. [4]

Автор Сальман та інші підкреслюють вирішальну роль архітектури в подоланні викликів та використанні можливостей цифрової трансформації для організацій. Розуміння ключових викликів та перспектив, описаних у дослідженні, дає можливість архітекторам розробляти ефективні стратегії створення високонавантажених систем. Автори рекомендують зосередитись на дослідженні архітектурних практик, для створення нових підходів розробки у майбутньому. [5]

Автор Склярєнко та інші визначають що масштабованість архітектури є критичним аспектом розробки сучасних застосунків. Застосування мікросервісної архітектури, хмарних рішень, кешування та балансування навантаження дає можливість ефективно справлятися зі зростанням навантаження і забезпечувати високу продуктивність та надійність системи. Кожен з цих підходів, які описані у статті має свої сильні та слабкі сторони та вибір конкретного рішення залежить від особливостей проекту, вимог до продуктивності, бюджетних обмежень та інших чинників. Дослідники зазначають, що важливо ретельно продумувати архітектуру на початкових

етапах розробки, щоб уникнути потенційних труднощів у майбутньому та забезпечити можливість подальшого масштабування у майбутньому. [6]

Таким чином, результати досліджень підкреслюють критичну роль архітектури в забезпеченні надійності та ефективності систем. З плином часу технології розвиваються, архітектори стикаються з новими проблемами, що потребують адаптивних та інноваційних підходів.

1.5. Основні проблеми високонавантажених систем

Системи, які обслуговують велику кількість користувачів або обробляють великі обсяги даних, повинні забезпечувати продуктивність, надійність та безперервну роботу. Це ставить складні вимоги до архітектури таких високонавантажених систем, які повинні не лише підтримувати великі обсяги трафіку, а й забезпечувати швидке відновлення у разі збоїв. Через збої та інші проблеми з продуктивністю призводять до додаткових витрат компаній. Таким чином, за даними статті Gartner, втрати великих онлайн-сервісів у разі простою досягають 300 000\$ за годину. Ключовими джерелами проблем у інфраструктурі високого навантаження є обсяг даних, складність та швидкість змін. [1]

Розглянемо проблеми, які можуть виникнути при не правильному плануванні архітектури при високому навантаженні системи: [7]

- Втрата даних: перевантаження серверів може призводити до втрати даних, що є особливо критичним у системах, які обробляють важливу інформацію.
- Неоптимізовані бази даних: оптимізована база даних, забезпечуючи високий рівень безпеки та продуктивності, тоді як неоптимізована база даних може завдати шкоди додатку в продуктовому середовищі.
- Нерівномірний розподіл навантаження: може спричинити повільний час відгуку через неправильний розподіл навантаження по сервісам.

- Різкі стрибки трафіку: стрибки трафіку трапляються особливо під час акцій і система може виявитись неготовою до додаткового навантаження. Це може призвести до уповільнення серверів, погіршення продуктивності і негативно вплинути на імідж.
- Неконтрольований розподіл даних: системи зберігання даних схильні до погіршення продуктивності з часом, розробка плану для управління та моніторингу даних у процесі їхнього зростання є невід’ємною частиною успішної роботи ресурсу.
- Ефект доміно: збій в одному місці може вплинути на інші частини непередбачуваним чином. Можливо необхідно розглянути стратегію, яка вводить аномальні помилки в мережу, щоб перевірити межі стійкості та можливості системи.
- Репутаційні: нестабільна робота системи, може негативно позначитись на репутації компанії та призвести до втрати довіри клієнтів.
- Фінансові витрати: перевантаження серверів у високонавантажених системах призводить до різноманітних додаткових витрат, які можуть серйозно вплинути на фінансовий стан компанії.

1.6 Аналіз архітектури високонавантажених систем

1.6.1. Визначення архітектури систем

Високопродуктивні програмні системи є критично важливими компонентами сучасних технологій, оскільки вони дозволяють бізнесу працювати ефективно, забезпечуючи досвід для користувачів та надають конкурентні переваги на ринку. Високопродуктивні програмні системи – це застосунки або системи, що демонструють виняткову продуктивність, масштабованість, надійність та безпеку. В сучасному ринку повільні програмні системи можуть призвести до втрати доходів, зниження реєстрації користувачів та завдання шкоди репутації компанії.

Архітектура – це набір рекомендацій або умовностей, які визначають як структурувати та організовувати програмну систему. Вона описує компоненти системи, їхні обов’язки, взаємодії та залежності. Архітектура не є жорстким стилем для створення програмної системи, а скоріше загальним підходом, яку можна адаптувати та налаштувати відповідно до обмежень продукту. Немає оптимального стилю для всіх випадків, кожен стиль має свої плюси та мінуси, які треба враховувати при виборі. Чітко визначена архітектура допомагає розробникам краще зрозуміти систему та свою роль у її побудові. Вона допомагає сприяти ефективній комунікації, співпраці та узгодженню цілей бізнесу між різними командами. Архітектура повинна відповідати своїм функціональним вимогам (тобто, які функції повинна виконувати) та нефункціональним вимогам, такі як продуктивність, масштабованість, стабільність, модульність, повторне використання, безпека та можливість змін, а також знижує ризики та витрати. [8]

При виборі певної архітектури необхідно дотримуватись певних концепцій, розглянемо деякі з них.

- Модульність: розбиття складної системи на менші та простіші компоненти, які можуть розроблятися.
- Повторне використання коду: уникнення дублювання бізнес логіки шляхом створення багаторазових компонентів.
- Розширюваність: можливість системи розвиватись та адаптуватись до нових вимог.
- Підтримка: спрощення розуміння програмного коду, модифікації та рефакторинг системи.
- Тестування: спрощення тестування бізнес-логіки завдяки розділенню компонентів та їхніх залежностей.

Приклад сучасної архітектури, на прикладі Netflix зображено на рисунку 1.5

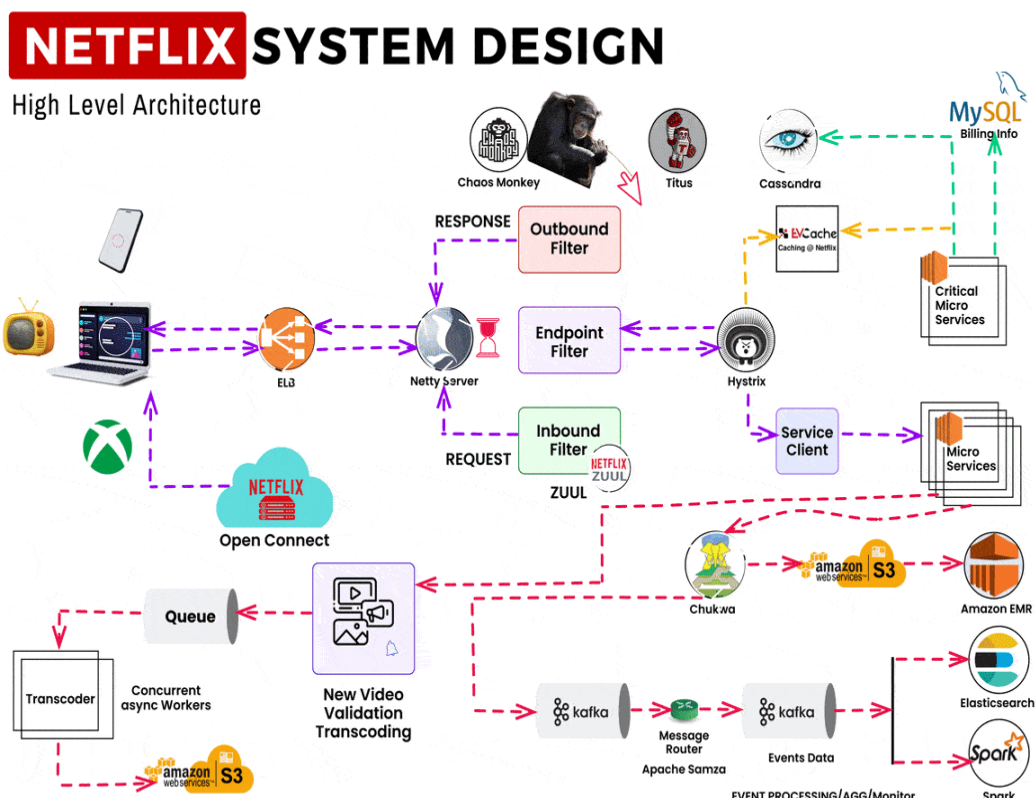


Рисунок 1.5 Сучасна архітектура Netflix

У сучасному світі потокових сервісів Netflix є лідером за кількістю користувачів та обсягом контенту, який він надає у реальному часі. Для того щоб забезпечити стабільність та масштабованість та високу якість обслуговування для мільйонів глядачів по всьому світі, вона використовує складну архітектуру, яка включає сучасні технології. Вона побудована за принципами мікросервісної архітектури, яка дозволяє кожному компоненту функціонувати незалежно, підвищуючи гнучкість системи.

Наразі існує декілька основних популярних архітектурних шаблонів, які обирають в залежності від задач, які необхідно вирішити, до них відносяться: монолітна, мікросервісна, сервісно-орієнтована, подійно-орієнтована, безсерверна та багатошарова. Далі розглянемо основні типи архітектури високонавантажених систем, та як саме вони допомагають забезпечувати надійність та підтримку роботи сервісів.

1.6.2. Монолітна архітектура

Монолітна архітектура – традиційний підхід до розробки програмного забезпечення, при якому програмне забезпечення розробляється як єдиний, неподільний блок, який складається з декількох пов'язаних між собою компонентів або модулів. Ця архітектура містить в собі усі функціональні частини, такі як інтерфейс користувача, бізнес-логіка та доступ до даних. Раніше, коли створювались перші додатки, монолітний підхід був єдиним методом розробки великих систем до появи більш модульних шаблонів, які спростили розробку програмного забезпечення. Ця архітектурна модель охоплює три основні компоненти. Перший компонент – інтерфейс користувача на стороні клієнта, який відповідає за взаємодію користувача із застосунком і відображення даних. Другий компонент – серверна частина, що керує бізнес-логікою додатку та обробляє запити користувачів, виконуючи основні операції. Третій компонент – інтерфейс даних, який забезпечує з'єднання та доступ до бази даних для зберігання та отримання інформації.[9]

Приклад монолітної архітектури зображено на рисунку 1.6.

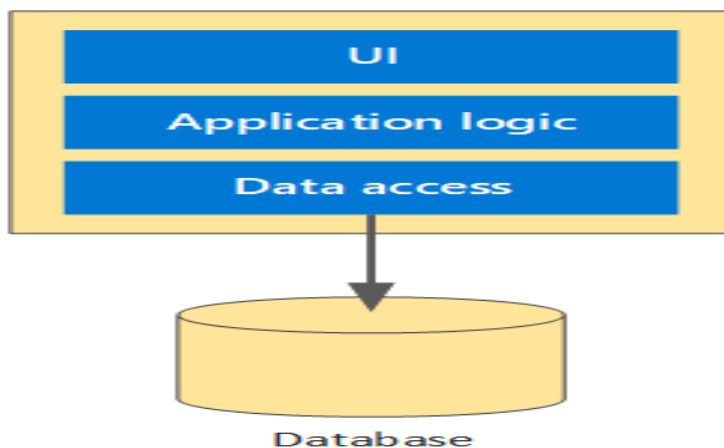


Рис 1.6 Загальна схема монолітної архітектури

Монолітна архітектура є популярним вибором для певних типів проектів, де важлива простота, зручність підтримки та єдина кодова база. Розглянемо основні типи, де ця архітектура може бути корисною.

- Малі та середні проекти: архітектура ідеально підходить для невеликих застосунків, які не потребують складного масштабування системи.
- Внутрішні проекти: багато внутрішніх інструментів для організацій використовують монолітну архітектуру для забезпечення швидкого доступу до даних та централізованого управління.
- Автоматизовані системи управління контентом (CMS): багато систем конструювання контенту, прикладом якого є: WordPress, Joomla, Drupal використовують монолітну архітектуру. Для адміністраторів вона спрощує обслуговування веб-сайту, а також надає зручність у використанні єдиного інтерфейсу управління контенту.
- Корпоративні інформаційні системи: ERP-системи дуже часто будуються на основі монолітної архітектури. Це дозволяє уникати складності між взаємодією інтерфейсу та внутрішньої бізнес логіки.

Існує кілька основних популярних фреймворків, які побудовані на основі монолітної архітектури. Вони дозволяють забезпечити всіма необхідними інструментами для побудови монолітної архітектури додатків.

Django – один із найпопулярніших фреймворків для розробки на Python. Він надає всі необхідні компоненти для створення інтерактивних веб-сайтів, який надає власну систему шаблонів, яка дозволяє створювати HTML-сторінки з динамічним контентом. Така система спрощує розробку інтерфейсу користувача і інтеграції з серверною частиною.

Laravel – це популярний open-source фреймворк для мови програмування PHP, який відомий своєю надійною та простотою у використанні. Laravel базується на шаблоні проектування модель-вид-контролер, який забезпечує структуру коду та полегшує процес розробки веб-додатків. Laravel надає можливість створювати монолітні додатки, сторінки та бізнес-логіку.

Основні плюси монолітної архітектури:

- Простота розробка і підтримки: Монолітна архітектура містить єдиний код, який використовуються і для побудови візуальної частини, і для

бізнес-логіки. Всі модулі та компоненти об'єднані, що спрощує налагодження підтримку та внесення змін.

- Швидке розгортання архітектури: Моноліт – це єдиний блок, процес розгортання на сервері відбувається швидко. Адміністраторам достатньо лише розгорнути один додаток, що знижує ймовірність помилок і спрощує управління версіями.

- Ефективне тестування: Монолітний застосунок можна тестувати як єдине ціле системи. Це значно полегшує процес інтеграційного тестування і знижує кількість помилок.

- Висока продуктивність: Завдяки тому, що усі компоненти системи тісно пов'язані між собою, можна досягти високої продуктивності.

Хоча монолітна архітектура має багато переваг, які дають простоту, особливо для невеликих проєктів, вона й має певні обмеження, які можуть стати критичними при високих навантаженнях системи.

- Проблеми з масштабуванням: масштабування монолітного застосунку означає, що необхідно масштабувати усю програму, яка може займати значну кількість ресурсів, а також деяка логіка може це не підтримувати взагалі.

- Низька гнучкість архітектури: коли додається нова бізнес-логіка до існуючого програмного забезпечення додаток стає менш ефективним, оскільки це може вимагати значних змін у всій системі, що уповільнює розвиток і ускладнює адаптацію до нових вимог.

- Вибір технологій: додатки зазвичай будуються на певному наборі технологій, які підтримують монолітну архітектуру, але з розвитком технологій створюються нові стеки технологій, які значно покращують продуктивність та надійність системи, і тоді міграція існуючої кодової бази стає важкою.

- Складність розгортання: розгортання такого типу додатку може бути складним під час оновлень системи, оскільки навіть найменші зміни потребують повного перезавантаження системи.

– Складна підтримка розробки додатку: в монолітній архітектурі багато розробників одночасно працюють над одним кодом та це дуже часто призводить до конфліктів при внесенні змін, ускладнює взаємодію між розробниками та приводить до непередбачуваних помилок.

Можна зробити висновок, що монолітна архітектура підходить для невеликих та не складних проектів, та тих які не потребують підтримки високого навантаження на систему та гнучкості при розробці. Багато компаній в минулому починали з використанням монолітної архітектури, але згодом перейшли до більш гнучких та надійних шаблонів проектування систем, такі як: мікросервісна або модульна архітектура для забезпечення кращої гнучкості, продуктивності та надійності.

1.6.3. Подійно-орієнтована архітектура

Подійно-орієнтована архітектура (EDA) – це архітектура, в якій ключові події в системі стають тригерами для виконання певних дій або процесів. Такий підхід часто використовуються для проектування високонавантажених систем, які потребують швидкого реагування на зміни, масштабованості та незалежності компонентів. [10]

Архітектура подій включає кілька ключових елементів. Подія (Event) є основною одиницею інформації, яка відображає зміну стану сутності або дію в системі, наприклад, реєстрацію користувача, оформлення замовлення чи поповнення балансу. Джерело подій (Producer) – це компонент, що генерує події. Наприклад, при поповненні балансу цей компонент створює подію, яка передає відповідну інформацію для подальшої обробки. Оброблювач подій (Event Consumer) отримує події, на які він підписаний, і виконує завдання на основі отриманої інформації, наприклад, створення транзакції при поповненні балансу. Шина подій (Event Bus) є основним механізмом архітектури, який забезпечує пересилання подій між різними компонентами системи.

Схема подійно-орієнтованої архітектури зображено на рисунку 1.7

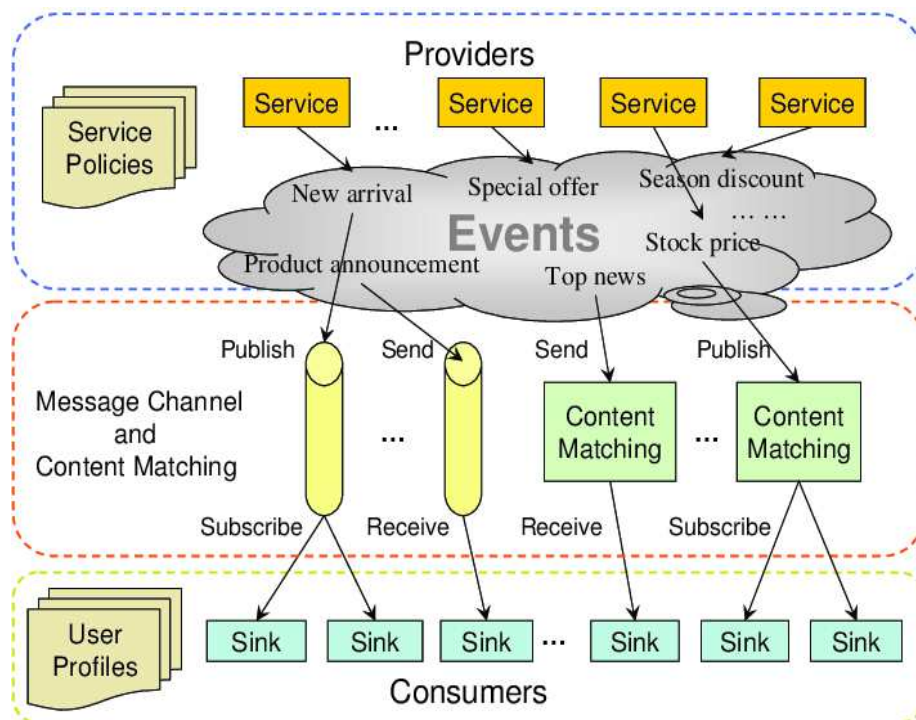


Рис 1.7 Схема подійно-орієнтованої архітектури

Подійно-орієнтована архітектура має кілька значних переваг, які роблять її надзвичайно корисною для розробки високонавантажених систем.

- **Масштабованість:** архітектура дозволяє легко масштабувати окремі компоненти системи без необхідності зміни усієї інфраструктури. Тому що взаємодія компонентів системи відбувається через події. Кожен компонент системи може бути окремо розміщений та мати безліч кількості реплік.
- **Швидкість обробки:** компоненти не чекають відповідь від одного, а дозволяють системі швидко реагувати на події, які виникають в реальному часі та оброблювати події без затримок.
- **Інтеграція з іншими системами:** архітектура дозволяє з легкістю інтегруватися з іншими через загальну шину подій. Цей механізм дозволяє працювати з системами, які мають різну архітектуру та зовнішніми системами. Загальна шина подій дозволяє обмінюватись інформацією один з одним.
- **Гнучкість:** архітектура дозволяє легко змінювати або додавати окремі модулі без ризику порушення роботи інших компонентів, оскільки достатньо додати нову подію в систему та підписатись на неї.

– Відмовостійкість: системи, які побудовані за допомогою цієї архітектури мають високу стійкість до збоїв, події можна повторно запускати або перезапущати в разі виникнення помилок, що дозволяє зберігати події до тих пір поки вони не будуть оброблені системою.

Подійно-орієнтована система, як і інші архітектури, мають свої недоліки, попри значні переваги. Розглянемо деякі з них. Складність управління та відстеження: коли у системі стає все більше подій та компонентів, управління системою стає складнішим. У великих системах це може впливати на стабільність і передбачуваність поведінки. Забезпечення послідовності обробки подій: в таких системах важко забезпечити послідовність обробки даних, оскільки події виконуються паралельно, що може призвести до ситуацій, коли події будуть змінювати одні і ті ж самі дані.

Застосування подійно-орієнтованої архітектури:

– Фінансові системи (FinTech): платежі та транзакції, досягаються через подійно-орієнтовану архітектуру, адже важливо щоб кожна транзакція виконалась, а в разі технічного збою спробувала виконатись повторно.

– Онлайн магазини: аналітика замовлень, обробка замовлень активно використовують архітектуру для швидкого реагування, а також розподілення на окремі події.

– Онлайн-ігри: у високонавантажених онлайн іграх, система подій використовується для синхронізації дій гравального середовища.

– Соціальні мережі: соціальні мережі активно використовують архітектуру на основі подій, бо такі системи мають велику кількість запитів до системи. Прикладом є коли користувач коментує повідомлення, додаток генерує подію, а потім це відображається на сторінках підписників.

– Логістика: багато логістичних додатків, використовують цю архітектуру для відстеження переміщення вантажів по всьому світу. Прикладом є прибуття товару на певну точку, система генерує подію яка дозволить відстежувати процес доставки в реальному часі.

– Інтернет речі (IoT): в таких речах подійно-орієнтована архітектура є однією з ключових, оскільки вона дозволяє обробляти дані від датчиків у реальному часі. Прикладом є відправка подій про стан конкретного датчика для подальшого відображення у додатку.

Багато компаній впровадили EDA для покращення своїх операцій та обслуговування.

Netflix – майстерно використовує архітектуру для координації своєї великої розподіленої системи мікросервісів. Ця архітектура забезпечує обробку даних в реальному часі, що є критично важливим для персоналізації користувацького досвіду на мільйонах пристроїв по всьому світу. Завдяки EDA, Netflix може налаштовувати рекомендації контенту, управляти якістю потокового відео в реальному часі. Такий підхід підвищує задоволеність користувачів, завдяки персоналізованому перегляду та оптимізує використання ресурсів у глобальній інфраструктурі.

Amazon – використовує подійно-орієнтовану архітектуру у своїй платформі електронної комерції, для забезпечення високоякісної обробки замовлень в реальному часі. Завдяки EDA, Amazon може автоматично оновлювати замовлення в реальному часі, а саме оновлювати дані про залишки на складі, інформувати клієнтів про статус їх замовлення. Впровадження EDA дозволяє ефективно масштабувати операції, управляти мільйонами товарів та забезпечувати стабільність і динамічність онлайн-шопінгу, що сприяє задоволенню клієнтів.

Twitter – застосовує подійно-орієнтовану модель для ефективного управління величезними потоками твітів і соціальних взаємодій, які генерують користувачі щосекунди. Ця модель дозволяє Twitter забезпечувати своєчасну доставку оновлень, повідомлень і твітів, незважаючи на великий обсяг даних. Використання EDA, дає можливість фільтрувати, сортувати та розподіляти цю інформацію ефективно, забезпечуючи миттєве надходження відповідного контенту та повідомлень користувачам.

EDA має велике практичне застосування в багатьох високонавантажених системах, завдяки своїй здатності забезпечувати швидке реагування, високу продуктивність та гнучкість. Архітектура EDA може з легкістю обробляти великі за обсягом потоки подій, завдяки асинхронній обробці подій, великі компанії можуть створювати системи, що ефективно масштабуються та адаптуються під сучасні вимоги бізнесу й інновації.

1.6.4. Мікросервісна архітектура

В останні роки з розвитком цифрових технологій та впровадженні нових змін, виникла потреба оновлювати та масштабувати системи без повного їх розгортання.

Мікросервіси – це легкі, автономні компоненти, які виконують відповідні функції в межах застосунку, спілкуючись один з одним через API. Незважаючи на незалежність один від одного, мікросервіси працюють разом, для досягнення необхідних результатів.

Мікросервіси – це архітектурний підхід, що передбачає поділ великих застосунків на менші, функціональні одиниці, здатні працювати та спілкуватись незалежно один від одного. Цей підхід виник через обмеження монолітної архітектури. Завдяки мікросервісам кожен компонент є незалежно одиницею, який може взаємодіяти з іншими. Це дозволяє розробникам досягти масштабованості, простоти та гнучкості, необхідних для створення складних програмних рішень.

Архітектура мікросервісів зосереджена на розбитті додатків за певним аспектом та функцій застосунку, прикладом застосування може бути пошук даних, логування подій, які взаємодіють один з одним. [11]

Приклад мікро-сервісної архітектури зображено на рисунку 1.8

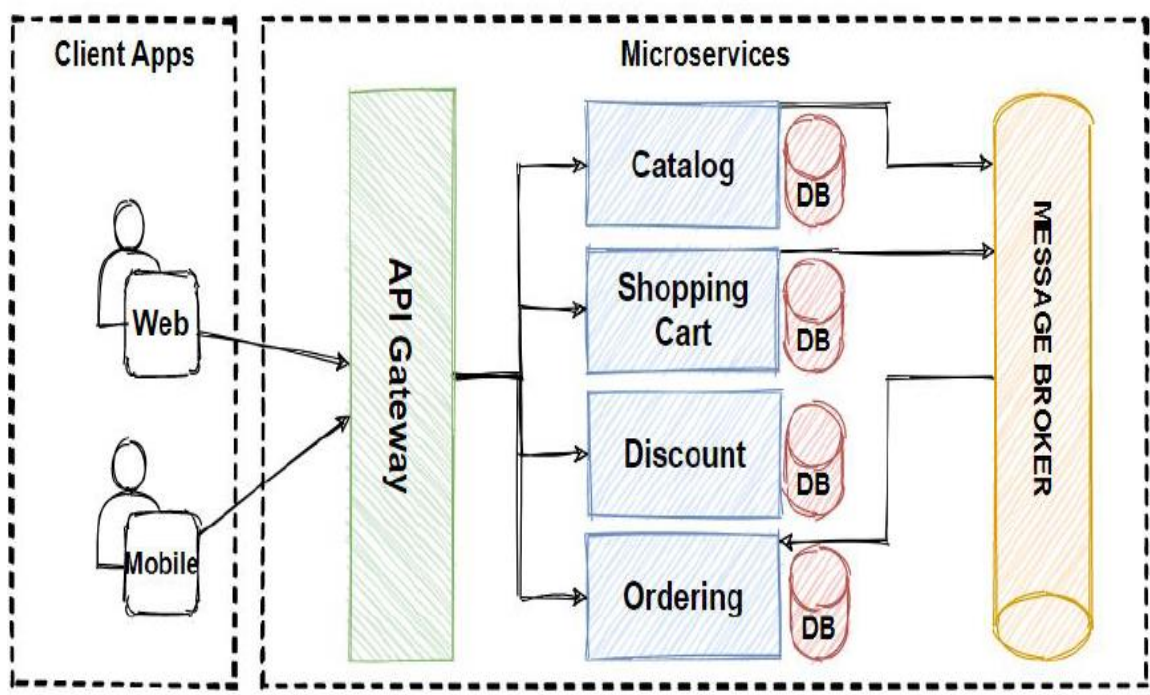


Рисунок 1.8 Схема мікросервісної архітектури

Мікросервісна архітектура має кілька ключових переваг, які роблять її популярною для розробки складних і масштабованих рішень.

- Масштабованість: завдяки тому що кожен компонент системи є незалежним, його можна розробляти, розгортувати та тестувати окремо від інших. Також це дозволяє швидко проваджувати зміни або виправляти помилки.
- Гнучкість у виборі технологій: для різних задач можна використовувати різні технології та мови програмування, оскільки мікросервіси не залежать один від одного. Прикладом може бути мікро-сервіс, який призначений для авторизації та написаний на Python, а інший написаний на Node.js, що допоможе досягти кращої продуктивності.
- Просте тестування та зменшення кількості помилок: ця архітектура дозволяє ізолювати помилки в межах одного сервісу. Тестування стає менш простим, оскільки кожен мікро-сервіс відповідає за свою задачу. Це знижує ризик помилок в продуктивному середовищі.
- Відмовостійкість: якщо один компонент системи дасть збій, то це не призведе до зупинки усієї системи, оскільки мікро-сервіси можуть працювати

автономно. Така відмовостійкість дуже важлива для великих систем із високими вимогами до стабільності.

Мікросервісна архітектура має також свої недоліки, які можуть ускладнювати її впровадження та управління.

- Між серверна комунікація: хоч і мікросервіси функціонують незалежно, вони часто взаємодіють один з одним для отримання даних. Це вимагає підтримки надійних, стандартизованих API комунікацій.

- Розподілені транзакції: транзакції передбачають послідовність дій, які потребують коректної роботи мікросервісів, це підвищує ризик загального збою через помилку в одному з елементів.

- Логування: при розгортанні мікросервісів кожен з них має власний механізм логування. Це може призвести до великої кількості логів, які є неструктурованими та складними для підтримки.

- Операційні витрати: наявність багатьох дрібних сервісів може збільшити операційні витрати, які пов'язані з розгортанням, масштабуванням та моніторингом.

Мікросервісна архітектура широко використовується в різних галузях, розглянемо основні з них. E-commerce платформи, такі як інтернет-магазини та платформи електронної комерції, застосовують мікросервісну архітектуру для забезпечення надійності роботи всіх процесів. У сфері платіжних систем і фінансових послуг фінансові компанії використовують цю архітектуру для підтримки різних платіжних операцій. Соціальні мережі, наприклад Twitter, використовують мікросервіси для забезпечення роботи різних функцій, таких як перегляд стрічки новин або відправка коментарів. Для таких систем ця архітектура дозволяє адаптувати і змінювати окремі елементи платформи.

Мікросервісна архітектура вирішує основну низьку проблем, з якими стикалися розробники у свої традиційних монолітних рішеннях. Однак розростання системи можуть створити нові виклики. Тому важливо мати власну моніторингову платформу. Ця архітектура активно використовується в сучасному світі, оскільки вона допомагає компаніям створювати гнучкі,

масштабовані та адаптивні додатки. Завдяки незалежності компонентів, ця архітектура полегшує роботу з великими обсягами даних і високим навантаженням.

1.6.5. Безсерверна архітектура

Безсерверна архітектура (Serverless architecture) – це підхід до розробки програмного забезпечення, який дозволяє створювати та запускати сервіси без управління інфраструктурою. Розробники можуть писати та розгортати код, тоді як хмарний провайдер автоматично забезпечує сервери для запуску їхніх додатків. Сервери дозволяють користувачам взаємодіяти з додатком і отримувати доступ до бізнес-логіки, але управління потребує значного часу та ресурсів. Команди повинні обслуговувати сервери, піклуватись про безпеку, створювати резервні копії на випадок збою.

Впроваджуючи serverless-архітектуру, розробники можуть передати ці зобов'язання хмарному провайдеру та зосередитись на якісному написанні коду. Одним із популярним методом використання безсерверних архітектур є використання Function as Service (FaaS), де розробники пишуть програмний код у вигляді окремих функцій. Кожна функція виконує певне завдання при спрацьовуванні події, наприклад після відправки HTTP запиту. Коли викликається функція, хмарний провайдер виконує її на своєму сервері та віддає результат. [12]

Кожна функція, яка виконується у безсерверній архітектурі, має певні характеристики, що задаються хмарним провайдером. Тривалість визначає час, який потрібен для виконання функції. Холодний старт є затримкою, яка виникає під час першого виклику функції. Обмеження на одночасне виконання вказує кількість екземплярів функції, які можуть одночасно виконуватись у хмарному провайдері. Таймаут визначає час, протягом якого провайдер дозволяє функції

виконуватись перед тим, як завершити її. Більшість хмарних провайдерів встановлюють мінімальний та максимальний таймаут.

Приклад безсерверної архітектури зображено на рисунку 1.9.

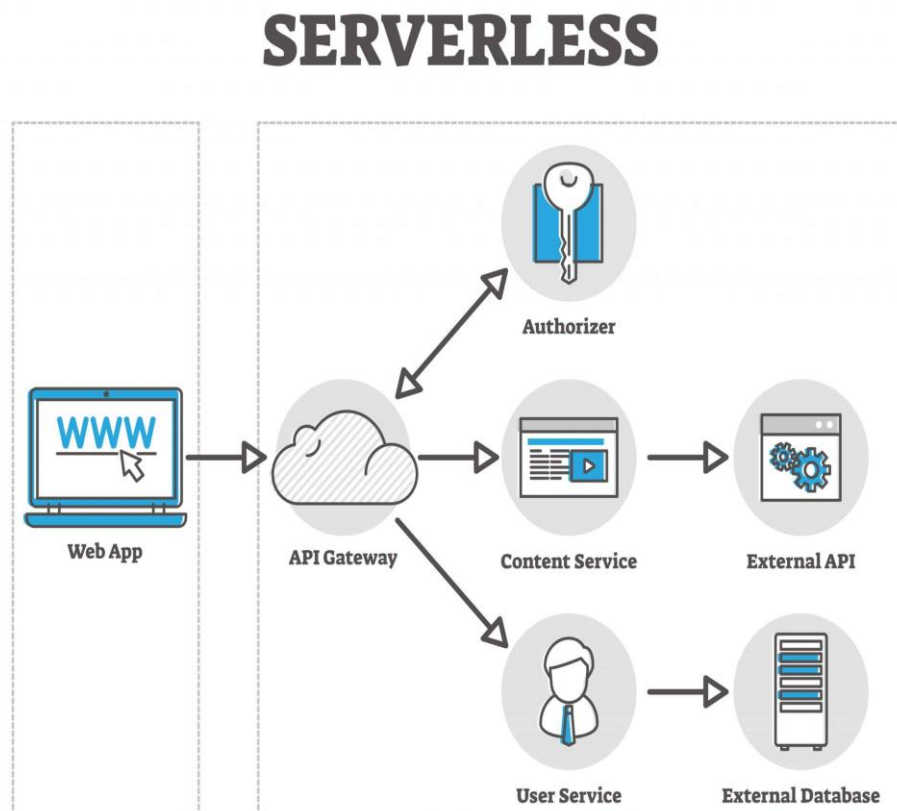


Рис 1.9 Загальна схема безсерверної архітектури

Розглянемо основні переваги serverless архітектури:

- Оплата за фактичне використання: у безсерверній архітектурі, потрібно платити тільки за виконання окремих функцій. Немає необхідності оплачувати за ресурси, які не використовуються.
- Масштабованість: безсерверні платформи дозволяють автоматично збільшувати або зменшувати використовувані ресурси залежно від навантаження.
- Швидке розгортання та оновлення: немає необхідності завантажувати код на сервери або налаштовувати бекенд, щоб випустити робочу

версію застосунку. Розробники можуть швидко завантажувати частину коду та запускати новий продукт.

- Відсутність необхідності управління серверами: безсерверні технології використовують сервери, але розробникам не потрібно ними управляти, оскільки всю інфраструктуру контролює постачальник хмарних послуг.

- Зниження затримок у виконанні коду. Хмарні провайдери можуть використовувати запускати функції на серверах, які розташовані ближче до серверу.

Розглянемо основні недоліки serverless:

- Проблема холодного старту: в архітектурі оплата відбувається лише за використання, але якщо функція викликається рідко, можлива значна втрата продуктивності. Функції, які запускаються після не тривалої активності, можуть працювати повільніше.

- Непридатність для довготривалих завдань: архітектура ефективна для короткострокових процесів, але для тривалої вона може бути не вигідна через високу вартість обчислювального часу.

- Складність процесу розробки: функції в цій архітектурі дрібніші ніж у традиційній архітектурі, потрібен додатковий час для їх налаштування.

- Залежність від постачальника послуг: безсерверні архітектури часто залежать від послуг та інструментів одного постачальника послуг, що може призвести до залежності від цього постачальника та подальшу складну міграцію до іншого провайдеру.

- Обмежений контроль та гнучкість: використання безсерверного підходу обмежує контроль над інфраструктурою, включаючи операційну систему та апаратне забезпечення.

- Складнощі моніторингу та відлагодження: через розподілену структуру безсерверних додатків моніторинг та введення логів може бути складним. Аналіз роботи додатків може бути ускладнене, особливо при спробі відтворити умови при якому виникла помилка.

Безсерверні функції часто використовують для швидкої обробки зображень та відео. Прикладом може бути соціальна мережа, яка використовує безсерверні обчислення для зміни розміру зображення або його формату. Такі додатки також використовують для обробки фінансових транзакцій в електронній комерції. Великі системи можуть автоматично аналізувати логи та виявляти аномалії, забезпечуючи моніторинг і стабільність роботи.

Безсерверна архітектура стає більш популярною, так як можна динамічно використовувати обчислювальну архітектуру хмарного додатку, зменшуючи витрати на обслуговування серверів. Успішне використання serverless архітектури залежить від розуміння цих обмежень та відповідного набору завдань. Безсерверна архітектура має свої обмеження, а саме проблеми з повільним запуском функцій та складність у розробці та моніторингу додатку.

1.6.6. Сервіс-орієнтована архітектура

Сервіс-орієнтована архітектура (SOA) – це стиль проектування програмного забезпечення, в якому компоненти додатків надають послуги іншим компонентам через доступні протоколи зв'язку мережі. Прикладом використання цієї архітектури може бути організація, яка має сервіси, що взаємодіють один з одним двома можливими способами: через обмін даними або через координацію двох чи більше сервісів. [13]

Принципи сервіс-орієнтованої архітектури:

- Повторне використання: сервіси створюються з метою їх повторного використання в різних системах.
- Протокольна взаємодія: сервіси взаємодіють між собою за допомогою стандартних мережевих протоколів HTTP.
- Відповідність бізнес-процесу: сервіс повинен моделюватися для вирішення певної задачі.

- Автономність: сервіси можуть залишатись незалежними від інших елементів системи, що сприяє гнучкості та полегшує обслуговування.
- Чітке визначення інтерфейсів: кожен сервіс повинен мати чітко визначений інтерфейс або контракт, який описує його функціональність та правила взаємодії з іншими сервісами.
- Гранулярність: сервіси повинні мати відповідний розмір і охоплення, ідеально виконуючи одну окрему бізнес-функцію на кожен сервіс. Розробники можуть використовувати декілька таких сервісів для створення композитного сервісу, який буде виконувати складні операції.

Схема сервісно-орієнтованої архітектури зображено на рисунку 1.10

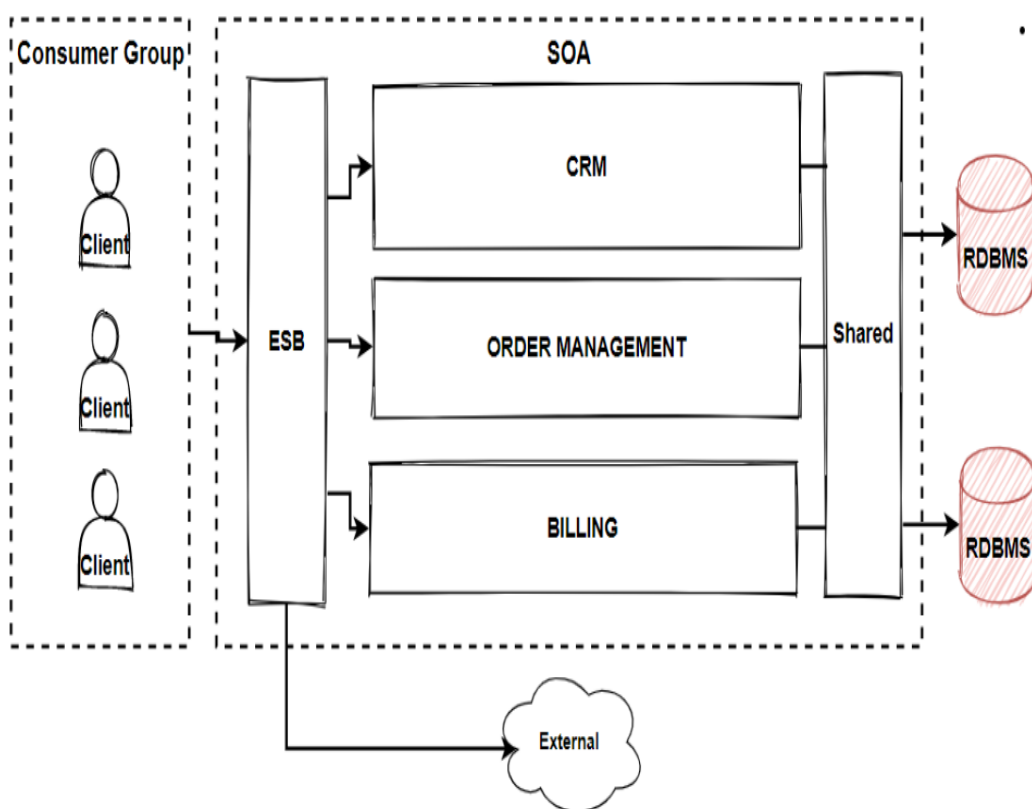


Рис 1.10 Загальна схема сервіс-орієнтованої архітектури

Сервісно-орієнтована архітектура (SOA) складається з кількох основних компонентів. Опис сервісу є базовим елементом SOA, що визначає основні функції кожного сервісу. Реалізація сервісу включає програмний код, який створює логіку для виконання певних функцій, таких як аутентифікація користувача або реєстрація. Контракт сервісу визначає умови його

використання, включаючи попередні вимоги та вартість. Інтерфейс сервісу забезпечує взаємодію між сервісами або іншими системами, дозволяючи викликати операції або обмінюватися даними.

Постачальник сервісу створює, підтримує та надає сервіси для використання іншими компонентами. Реєстр сервісів, або репозиторій, є каталогом доступних сервісів із детальною інформацією про них, включаючи спосіб взаємодії. Споживач сервісу, який може бути окремим додатком, системою або іншим сервісом, запитує виконання конкретного сервісу у постачальника. Контракт між постачальником і споживачем регламентує їхню взаємодію, навіть якщо вони належать до різних організацій, відділів або галузей.

Використання сервісно-орієнтованої архітектури для розробки продуктів має кілька переваг. Нижче розглянемо деякі з найбільш важливих переваг.

- Гнучкість: однією з основних переваг SOA є створення повторно використовуваних програмних компонентів. Кожен сервіс розроблений для виконання конкретної функції, яку можна використовувати повторно в багатьох системах або додатках.

- Можливість інтеграції: SOA дозволяє легко інтегруватися з іншими системами, оскільки вона має протокол для обміну даними. Завдяки цьому простіше інтегруватись з новими сервісами в наявні системи, що допомагає працювати ефективніше і знижує витрати.

- Розширюваність: SOA дозволяє розширювати бізнес-логіку кожного сервісу. Оскільки вони слабо зв'язані між собою та зміни в одному не можуть впливати на інші.

Основні недоліки SOA.

- Складність управління: архітектура передбачає велику кількість сервісів, що ускладнює їх використання.

- Залежність від протоколів обміну даних: архітектура часто покладається на протоколи REST, що можуть бути не завжди оптимальними з точки зору продуктивності або гнучкості.

– Складність тестування та аналіз роботи сервісу: у сервісах важко забезпечити комплексне тестування, адже необхідно продумати взаємодію усіх сервісів та відповідність бізнес-вимогам.

Багато організацій успішно використовують SOA для покращення бізнес-процесів і вдосконалення своїх систем. Наприклад, банки застосовують SOA для інтеграції різних сервісів, таких як обробка платежів чи управління рахунками. У сфері електронної комерції компанії покладаються на цю архітектуру для обробки замовлень, управління продуктовими запасами та покращення роботи з клієнтами.

SOA – це потужний шаблон проектування, який дозволяє створювати модульні програмні системи. Ця архітектура надає більше гнучкості, можливість масштабування та економічну ефективність. Незважаючи на те, що впровадження може бути складним, довгострокові переваги роблять її привабливим варіантом для високонавантажених систем.

1.6.7. Багатошарова архітектура

N-рівнева архітектура – її ще називають багатошаровою, оскільки програмне забезпечення створене так, щоб функції обробки, управління даними та презентаційний вигляд були фізично та розподілені за певною логікою. Це означає, що різні функції розміщуються на декількох машинах або кластерах, що забезпечує надання послуг без спільного використання ресурсів, і, таким чином, ці послуги надаються на максимальній потужності. Архітектура передбачає поділ застосунку на три різні рівні. [14]

Презентаційний рівень: це рівень інтерфейсу користувача, де відбувається взаємодія користувачів із застосунків. Він обробляє введення даних користувачем, відображає інформацію. Найпоширеніші технології, які використовують на цьому рівні, є HTML, CSS, JavaScript та фреймворки інтерфейсу користувача. Рівень бізнес-логіки: відомий як рівень логіки

застосунку, цей рівень містить основну логіку та операції застосунку. Він обробляє та управляє даними з презентаційного рівня, забезпечує дотримання бізнес-логіки та координує взаємодію між різними компонентами. Рівень доступу до даних: цей рівень відповідає за взаємодію з базами даних або зовнішніми джерелами даних. Він виконує завдання, такі як отримання, оновлення та видалення даних. Завдяки цьому рівню, інші частини застосунку не повинні турбуватись про специфіку зберігання даних. Рівень сутностей: цей рівень відповідає за представлення основних бізнес-об'єктів та логіки застосунку. Завдяки відокремленню цього рівня від інших, можна зосередитись на операціях, специфічних для домену. [14]

Схема багат шарової архітектури, зображено на рисунку 1.11

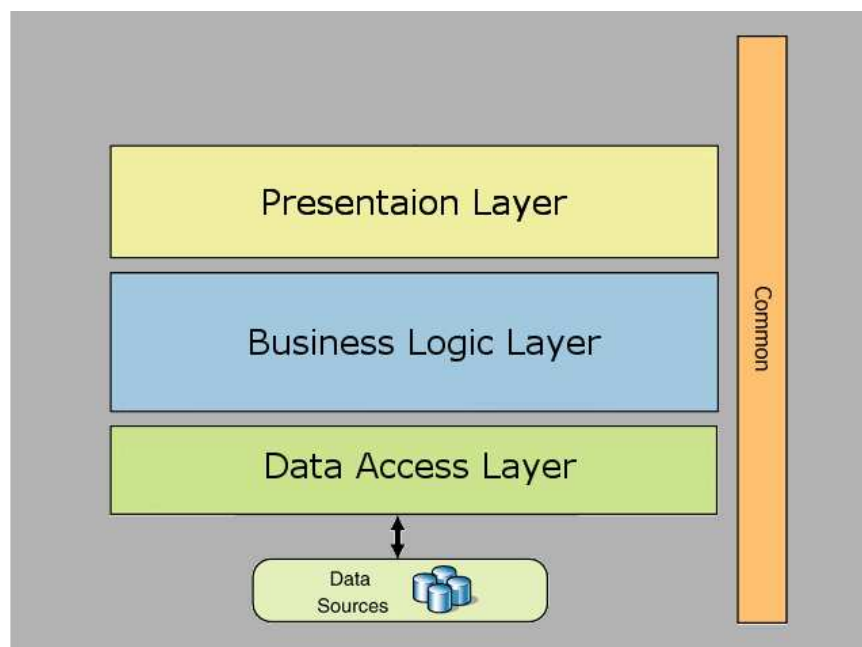


Рис 1.11 Схема багат шарової архітектури

Впровадження багат шарової архітектури забезпечує кілька переваг, що сприяють підвищенню якості та підтримки застосунку:

- Масштабованість: завдяки поділу застосунку на окремі рівні, стає можливим масштабувати кожен рівень незалежно.
- Повторне використання бізнес-логіки: природна архітектури сприяє повторному використанню коду.

- Підтримка та оновлення: завдяки чітким межах між рівнями, оновлення або виправлення конкретної частини застосунку стає більш керованим.

- Безпека: розділення обов'язків дозволяє краще керувати безпекою.

До недоліків багат шарової архітектури можна віднести наступні:

- Збільшення загальної складності: поділ на кілька рівнів додає складності, оскільки необхідно підтримувати взаємодію між ними.

- Збільшення мережевої затримки: кожен додатковий рівень додає затримку при передачі даних.

- Висока вартість: кожен рівень потребує окремих апаратних ресурсів, що може підвищити втрати на обладнання.

Багат шарова архітектура широко використовується в різних сферах, де потрібно забезпечити надійність і масштабованість та легкість обслуговування програмних систем.

Багат шарова архітектура широко використовується в різних сферах, де потрібно забезпечити надійність, масштабованість і легкість обслуговування програмних систем. Корпоративні системи: великі компанії часто використовують багат шарову архітектуру для створення корпоративних додатків. Мобільні додатки: архітектура використовується для створення мобільних додатків, де потрібно розподілити функціональні частини між клієнтським інтерфейсом та сервером. Фінансові та банківські системи: у фінансових установах, де важлива безпека та швидка обробка транзакцій, архітектура дозволяє обробляти великий обсяг транзакцій.

Багат шарова архітектура є потужним шаблоном проектування, що забезпечує модульність, масштабованість і підтримку програмних застосунків.

Розділяючи компоненти на окремі рівні, розробники можуть створювати застосунки, які краще адаптуються до змін і здатні впоратися зі зростаючими потребами користувачів. Впровадження архітектури вимагає продуманого планування та ретельної реалізації, але переваги, які вона забезпечує в плані

гнучкості та простоти обслуговування, роблять її цінним підходом для сучасної розробки програмного забезпечення.

1.7. Порівняння архітектури за ключовими характеристиками

Сучасні вимоги до програмного забезпечення такі як масштабованість, гнучкість, надійність та швидкість роботи вплинули на появу різних архітектурних шаблонів, кожен з яких пропонує унікальні рішення. Залежно від специфіки системи, проекту та його цілей, архітектура може значно вплинути на його продуктивність та економічну ефективність у довгостроковій перспективі. Коли треба оцінити архітектуру програмного забезпечення, важливо врахувати декілька ключових критеріїв, які допоможуть визначити її ефективність та відповідність потребам, розглянемо їх.

Структура – це спосіб, у якому компоненти та модулі утворюють архітектуру системи.

Масштабованість – це здатність архітектури підтримувати більше навантаження та обсягу даних без значної деградації та продуктивності.

Складність – основний чинник, який впливає на вартість розробки та обслуговування.

Критерій використання – визначає, де можна застосувати архітектуру у певній предметній області.

Модульність – критичний аспект архітектури програмного забезпечення, оскільки впливає на те, наскільки легко підтримувати, масштабувати та змінювати систему.

Основна ідея модульності полягає у розподілі системи на окремі компоненти, які виконують окремі функції.

Порівняння архітектури зображено на рисунку 1.12 та в таблиці 1.1.

Порівняння ключових особливостей архітектури

Архітектура	Структура	Масштабованість	Складність	Використання
Монолітна архітектура	Єдина кодова база	Обмежена масштабованість	Проста складність	Невеликі проекти, MVP
Сервісно-орієнтована архітектура	Набір сервісів	На рівні сервісів	Помірна	Великі Підприємства, інтеграція різних систем
Мікросервісна архітектура	Набір сервісів	Точна Масштабованість на рівні сервісів	Висока складність управління	Великі масштабовані системи
Подійно-орієнтована архітектура	Компоненти взаємодіють через події	На рівні обробників подій	Складна взаємодія подій і обробників	Великі системи реального часу, обробка подій
Безсерверна архітектура	Функції виконуються на вимогу	Розгортання окремих функцій	Низька складність управління	Системи з нерегулярним навантаженням
Багатошарова архітектура	Набір сервісів	На рівні сервісів	Проста складність	Великі системи

Monolithic Vs SOA Vs Microservices

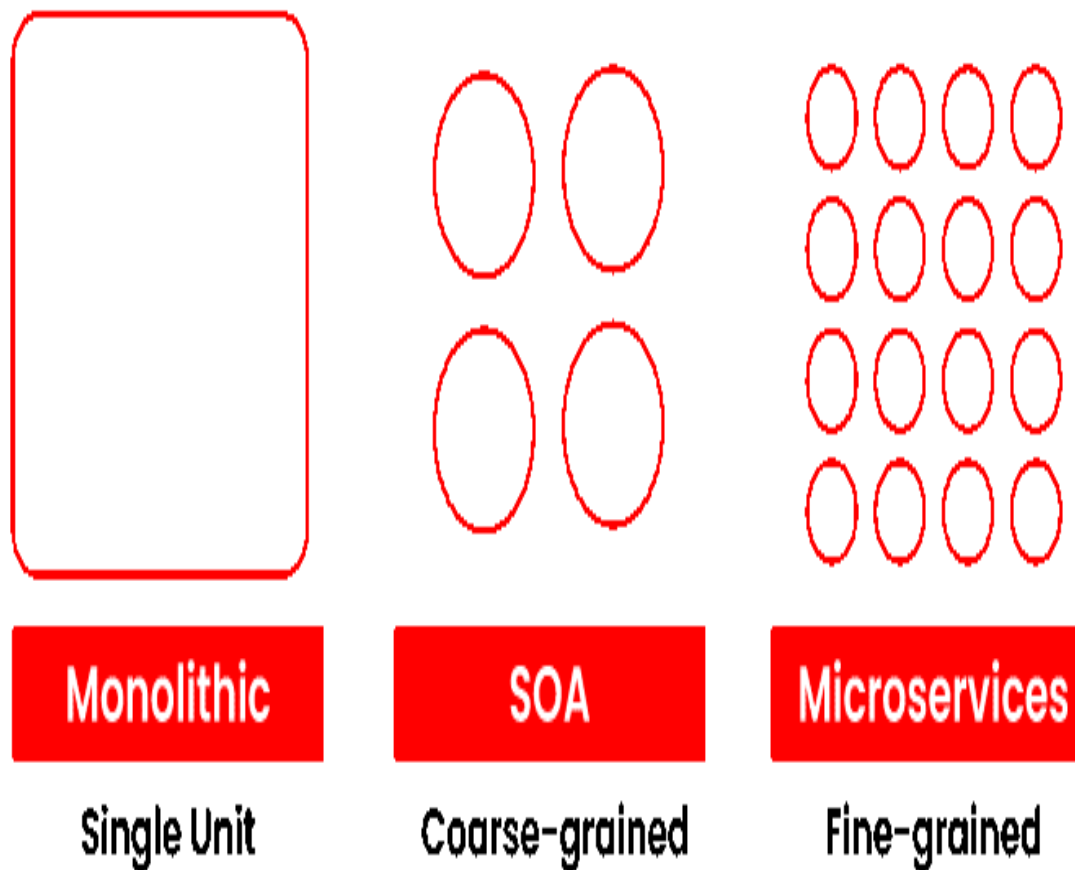


Рис 1.12 Порівняння модульності архітектури

Для вибору найкращої архітектури високонавантаженого сервісу застосуємо математичний метод TOPSIS (Technique for Order Preference by Similarity to Ideal Solution).

Метод TOPSIS – це метод багатокритеріального прийняття рішень, який базується на виборі альтернативи, що має мінімальну відстань до ідеального рішення і максимальну відстань до анти ідеального рішення.

Для визначення найкращої архітектури було проведено дослідження, засноване на опитуванні. У дослідженні взяли участь фахівець з галузі інформаційних технологій, який надав оцінку різних архітектурних підходів за трьома ключовими критеріями: масштабованість, складність впровадження та

ефективність використання. Експерт оцінив шість популярних архітектур за шкалою від 1 до 5, де 1 означає низький рівень показника, а 5 – найвищий та ваги критеріїв.

Результати опитувань представлені в таблиці 1.2

Таблиця 1.2

Результати опитувань фахівців інформаційних технологій

Архітектура	Масштабованість	Складність	Використання
Монолітна	2	1	3
Сервісно-орієнтована	4	3	5
Мікросервісна	5	5	4
Подійно-орієнтована	4	4	5
Безсерверна	3	2	4
Багатошарова	4	2	4

Ваги критеріїв визначено як:

- Масштабованість: 0.4.
- Складність: 0.3.
- Використання: 0.3.

Застосуємо формулу нормалізації (1.1)

$$r_{ij} = \frac{x_{ij}}{\sqrt{\sum_{i=1}^m x_{ij}^2}}, \quad (1.1)$$

де r_{ij} – нормалізоване значення для i -го рядка i та j -го стовпця, x_{ij} – початкове значення елемента матриці, m – кількість рядків у матриці,

Отримаємо такі нормалізовані значення, зображені в таблиці 1.3

Таблиця 1.3

Нормалізовані значення обчислень

Архітектура	Масштабованість	Складність	Використання
Монолітна	0.267	0.169	0.333
Сервісно-орієнтована	0.534	0.507	0.555
Мікросервісна	0.668	0.845	0.444
Подійно-орієнтована	0.534	0.676	0.555
Безсерверна	0.401	0.338	0.444
Багатошарова	0.534	0.338	0.444

Розрахуємо зважену матрицю за формулою (1.2)

$$v_{ij} = w_j \cdot r_{ij} \quad (1.2)$$

Таблиця 1.4

Зважена матриця обчислень

Архітектура	Масштабованість	Складність	Використання
Монолітна	0.106	0.050	0.100
Сервісно-орієнтована	0.213	0.152	0.166
Мікросервісна	0.267	0.253	0.133
Подійно-орієнтована	0.213	0.202	0.166
Безсерверна	0.160	0.101	0.133
Багатошарова	0.213	0.101	0.133

Розрахуємо ідеальне та анти ідеальне рішення за формулами (1.3) та

$$A^+ = \max(v_{ij}) = (0.267, 0.050, 0.166) \quad (1.3)$$

Розрахуємо відстань ідеального та анти ідеального рішень за формулами (1.4) та (1.5)

$$S_i^+ = \sqrt{\sum_{j=1}^n (v_{ij} - A_j^+)^2} \quad (1.4)$$

$$S_i^- = \sqrt{\sum_{j=1}^n (v_{ij} - A_j^-)^2} \quad (1.5)$$

Розрахунки розрахунків ідеального та анти ідеального рішення зображено в таблиці 1.5

Таблиця 1.5

Ідеальні та анти ідеальні рішення обчислень

Архітектура	S+	S–
Монолітна	0.210	0
Сервісно-орієнтована	0.089	0.130
Мікросервісна	0.029	0.204
Подійно-орієнтована	0.058	0.156
Безсерверна	0.148	0.065
Багатошарова	0.148	0.065

Розрахуємо результати за допомогою коефіцієнту близькості за формулою (1.6)

$$C_i = \frac{S_i^-}{S_i^+ + S_i^-} \quad (1.6)$$

Результати розрахунків зображені в таблиці 1.6

Обчислення коефіцієнтів близькості

Архітектура	Ci
Монолітна	0
Сервісно-орієнтована	0.539
Мікросервісна	0.876
Подійно-орієнтована	0.729
Безсерверна	0.304
Багатошарова	0.304

Мікросервісна архітектура показала найвищий результат, завдяки високій масштабованості та збалансованості інших критеріїв, що робить її найкращим вибором для високонавантажених систем.

1.8. Постановка задачі дослідження

З результатів аналізу можна визначити, що недостатня масштабованість і складність використання є істотними проблемами для високо навантажених сервісів.

Розробка високонавантажених систем стала критично важливою у зв'язку з постійним збільшенням кількості користувачів і попиту на сервіси, що функціонують у реальному часі. Прикладом таких сервісів є системи моніторингу спортивних подій, соціальні мережі, фінансові платформи та сервіси e-commerce. Ці системи постійно стикаються з проблемами, пов'язаними із забезпеченнями високої швидкості та доступності при одночасному зростанні навантаження та обсягу оброблюваних даних.

Основним викликом у побудові високонавантажених систем є вибір та реалізація архітектури, здатної забезпечити стабільну роботу під великим

навантаженням, а також ефективний розподіл ресурсів. У рамках даної роботи необхідно дослідити існуючі архітектурні рішення та моделі оптимізації високонавантажених систем. Крім того, планується практична розробка моделей оптимізації для обробки великої кількості запитів у реальному часі. Результатом досліджень стане набір рекомендацій та практичних моделей оптимізації таких систем, що дозволить покращити їх продуктивність, надійність та стійкість до пікових навантажень.

Для досягнення цієї мети необхідно проаналізувати та вирішити наступні задачі:

1. Дослідити існуючі архітектури високонавантажених сервісів.
2. Розглянути моделі оптимізації високонавантаженого сервісу.
3. Застосувати математичне моделювання, для вибору кращих моделей оптимізації високонавантаженого сервісу під конкретні задачі.
4. Спроекувати архітектуру високонавантаженого сервісу.
5. Розробити моделі оптимізації високонавантаженого сервісу
6. Отримати практичні результати розробки моделей оптимізації.

Результатом роботи будуть практичні рекомендації щодо вибору архітектури та моделей оптимізації для високонавантажених систем, які можна застосувати підвищення продуктивності до стійкості до пікових навантажень.

Розроблені моделі та рекомендації сприятимуть створенню надійних, продуктивних, масштабованих систем, здатних забезпечувати стабільну роботу у реальному часі.

1.9. Висновки до розділу 1

Кожна архітектура має свої виклики, які розробникам слід враховувати при проектуванні та впровадженні високонавантажених систем, адже вони можуть значно вплинути на стабільність, масштабованість і продуктивність.

Для монолітної архітектури головною проблемою є тісна інтеграція компонентів, що ускладнює масштабування та гнучке внесення змін. Внаслідок цього система стає менш ефективною у великих масштабах, а будь-яке оновлення або збій може призвести до часу простою, знижуючи продуктивність і зручність підтримки.

У мікросервісній архітектурі одним із найбільших викликів є управління складністю розподіленою системи. При зростанні кількості сервісів стає важче забезпечити узгодженість і ефективну комунікацію між ними.

Для сервісно-орієнтованої архітектури головним викликом є ризик швидкого розростання сервісів без правильного проектування. Також, це може призвести до збільшення витрат на підтримку програмного коду, безпеки та ускладнення загальної структури системи.

У безсерверній архітектурі основним завданням є правильна конфігурація виконання функцій. Важливо також враховувати холодні старты функцій, що можуть затримувати їх перший запуск.

Для подійно-орієнтованої архітектури головною проблемою є труднощі зі збереженням послідовності повідомлень, що важливо для систем, де дані мають оброблятися у визначеному порядку.

За допомогою математичного методу TOPSIS визначено, що найкращою архітектурою для високонавантажених систем, є мікросервісна.

Розробники повинні ретельно зважувати компроміси кожної архітектури та впроваджувати оптимальні практики для забезпечення надійної роботи системи. До таких моделей може належати масштабування (горизонтальне або вертикальне), використання кешування для підвищення швидкості до даних, партиціювання на рівні СКБД для зменшення обсягу даних, а також застосування систем оркестрації сервісів.

РОЗДІЛ 2. МОДЕЛІ ОПТИМІЗАЦІЇ ВИСОКОНАВАНТАЖЕНОГО СЕРВІСУ

2.1. Оптимізаційна модель

Оптимізаційна модель – це модель, яка використовується для знаходження найкращого розв’язку проблеми з набору можливих варіантів, враховуючи певні обмеження та цілі. Оптимізаційне моделювання є потужним інструментом, яке застосовується в різних галузях, включаючи дослідження операцій, інженерію, економіку, фінанси, логістику та інші. Завдяки оптимізації розподілу ресурсів, виробничих процесів або логістики, оптимізаційне моделювання може знизити витрати та підвищити ефективність роботи в різних робочих процесах.

Крім того оптимізаційне моделювання покращує стратегічне планування та прийняття довгострокових рішень. Воно дозволяє організаціям оцінювати різні сценарії та альтернативи, допомагаючи зрозуміти потенційні наслідки перед їх впровадженням. Це може бути особливо корисним у таких галузях як, створення високонавантажених систем. [15]

Оптимізаційні моделі створені для того, щоб допомагати організаціям та окремим особам приймати обґрунтовані рішення, максимізуючи або мінімізуючи цільову функцію при дотриманні певних обмежень. Цільовими функціями виступають математичні вирази, які визначають, що потрібно максимізувати (наприклад, прибуток, дохід, ефективність) або мінімізувати (наприклад час).

В цій магістерській дисертації ми розглянемо високонавантажений сервіс, перед яким стоїть задача розробити моделі оптимізації, для цього застосуємо такі кроки моделювання:

1. Визначення проблеми: високонавантажений сервіс, хоче підвищити свою продуктивність, але постає перед вибором, які саме рішення необхідно обрати.

2. Визначення обмеження: високонавантажений сервіс, має обмеження обчислювальних ресурсів та фінанси.
3. Збирання даних: необхідно зібрати дані про оптимізаційні моделі.
4. Інтерпретування результатів досліджень: на основі аналізу моделей оптимізації високонавантаженого сервісу необхідно визначити найкраще рішення за допомогою математичних підходів. Для цього буде розглянуто різні моделі оптимізації та застосовано математичні підходи для обґрунтованого вибору оптимального варіанту, а саме: масштабування, кешування та партиціювання.

2.2. Масштабування сервісів: горизонтальне та вертикальне

Останні 20 років розробники стали свідками безпрецедентного зростання розмірів, складності та потужності програмних систем. Цей темп розвитку навряд чи сповільниться в наступні 20 років та якими будуть майбутні системи зараз важко уявити. Можна з упевненістю сказати, що все більше систем враховують можливість їх зростання, а саме більшу кількість запитів, даних і аналізу. Масштабованими системами в програмній архітектурі називають ті системи, які здатні адаптуватися до зростання.

Масштабованість дозволяє опрацьовувати зростаючий обсяг роботи, шляхом додавання ресурсів до системи. Простіше кажучи, масштабування визначає здатність програмної системи обробляти зростання в певних напрямках її функціонального призначення.

Розглянемо приклади напрямків функціонування. Кількість одночасних користувачів або зовнішніх запитів, які система може обробити, є важливим показником її ефективності. Обсяг даних, які система може ефективно обробляти та управляти ними, визначає її здатність працювати з великими масивами інформації. Здатність підтримувати стабільний, послідовний час відгуку навіть

при зростанні навантаження на запити, свідчить про надійність і стійкість системи. [16]

Масштабованість відіграє ключову роль у визначенні довгострокового успіху та стійкості сервісу. З розвитком бізнес-логіки, зростають вимоги до системи та інфраструктури.

Розглянемо основний вплив масштабованості на сучасні сервіси. По-перше, масштабована система забезпечує швидкість, надійність і доступність послуг, що значно покращує клієнтський досвід і сприяє задоволеності користувачів сервісу. По-друге, інвестування в масштабовані рішення дозволяє узгодити використання ресурсів із реальними потребами, забезпечуючи економічну ефективність і мінімізуючи надмірне виділення ресурсів. По-третє, масштабовані системи зазвичай включають резервування, розподілені ресурси та механізми аварійного відновлення, що підвищує стійкість системи та забезпечує її надійність і тривалу роботу.

Існує дві основні стратегії масштабування: горизонтальне та вертикальне.

Вертикальне масштабування – це процес збільшення потужності окремих машин у системі. Масштабування передбачає модернізацію апаратних компонентів, таких як процесор (CPU), оперативна пам'ять (RAM) та швидкість мережі, а також додавання більше ресурсів або сховища для обробки більших навантажень.

Вертикальне масштабування можна порівняти з розширенням мосту, для того щоб він зміг пропускати більше транспорту. Після того як міст буде зроблено міцнішим і ширшим, він може витримувати збільшене навантаження без необхідності додавання нових смуг.

Горизонтальне масштабування – це процес додавання більше машин до кластеру ресурсів у системі розподілу навантаження. Якщо система масштабується горизонтально, це означає додавання більше серверів до вашої існуючої інфраструктури для обробки збільшеного навантаження.

Це масштабування підвищує доступність, якщо є один екземпляр сервісу і він виходить з ладу, сервіс стає недоступним. Це називають єдиною точкою

відмови (Single Point of Failure), явище, яке потрібно уникати в будь-якій масштабованій системі. Наявність кількох реплік підвищує доступність. Якщо одна з реплік виходить з ладу, запити можуть бути направлені на іншу репліку. Система матиме зменшену потужність обробки до моменту поновлення несправного серверу, але вона залишатиметься доступною.

Схема порівнянь вертикального та горизонтального навантаження зображена на рисунку 2.1

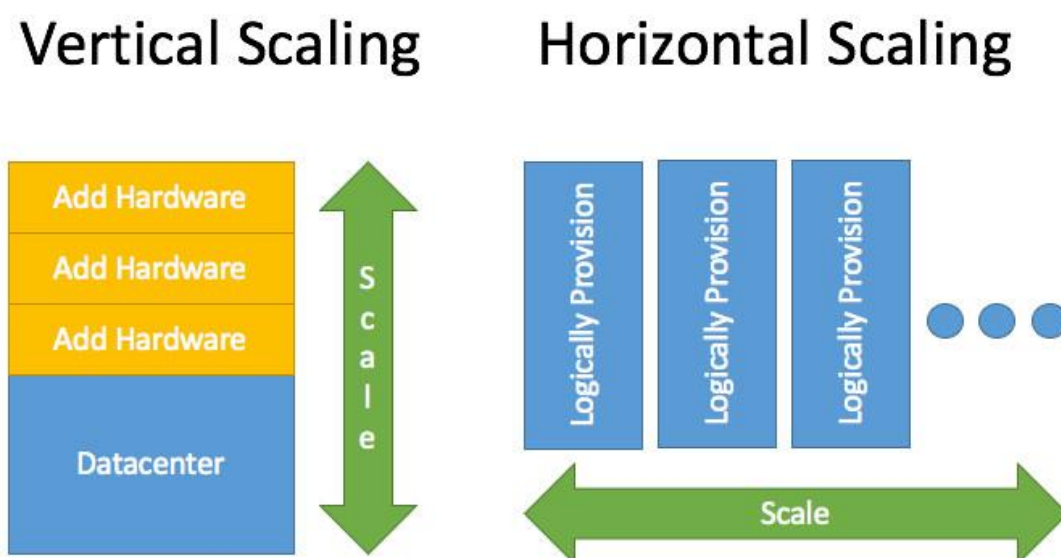


Рис 2.1 Схема вертикального та горизонтального масштабування

Загалом, масштабування є одним із ключових аспектів архітектури високонавантажених систем. Воно забезпечує здатність системи ефективно обробляти значні обсяги трафіку та навантаження, водночас зберігаючи високу продуктивність і надійність. При розробці високонавантажених систем виникають труднощі з визначенням оптимальної моделі масштабування, оскільки кожен підхід має свої переваги та обмеження. Для вирішення цієї проблеми було використано математичний підхід вагових коефіцієнтів.

На основі аналізу експертів у галузі IT-архітектури встановлено, що прийняття обґрунтованого рішення потребує врахування численних факторів, такі як: вартість, підтримка, надійність та масштабованість. Розглянемо детальний опис кожного з них:

1. Вартість – включає початкові інвестиції, витрати на обслуговування та операційні витрати, які виникають під час впровадження обраної моделі масштабування.
2. Продуктивність – визначає, наскільки швидко та ефективно система може обробляти запити.
3. Масштабованість – визначає, наскільки легко ми зможемо збільшити потужність системи для обробки більших обсягів трафіку або даних у майбутньому.
4. Надійність – оцінює, наскільки стабільною та доступною є система за умови відмови обладнання або несправності.

Оскільки думки, спеціалістів щодо цього не співпадають, спираючись на критерії визначені вище, з'ясуємо якому фактору слід приділити більше уваги, для цього застосуємо метод вагових коефіцієнтів.

Метод вагових коефіцієнтів – це оцінювання, яке полягає у наданні всім ознакам вагових коефіцієнтів. Він може використовуватись двома способами. У першому способі всім ознакам призначають вагові коефіцієнти так, щоб сума всіх коефіцієнтів дорівнювала 1, 10 або 100. У другому способі найважливішій з усіх ознак призначають ваговий коефіцієнт, який дорівнює певному фіксованому числу, а решті ознак – коефіцієнти, які дорівнюють часткам цього числа.

Узагальнену думку експертів S_i за i -ою ознакою розраховують за формулою (2.1)

$$S_i = (\sum_{j=1}^m a_{ij}) / m, \quad (2.1)$$

де a_{ij} - ваговий коефіцієнт, який призначив j -ий експерт i -ій ознаці; j - номер експерта; i - номер ознаки; m - кількість експертів, які оцінюють i -ту ознаку. Чим більшою є величина S_i , тим більш вагомою є ця ознака. [17]

У рамках дослідження було проведено опитування п'яти експертів у галузі ІТ-інфраструктури, які оцінили чотири ключові критерії для вибору оптимального критерія оцінки моделі масштабування. Для кожного критерію експерти надали вагові коефіцієнти, які відображають їх важливість у контексті

високонавантажених систем. Розрахуємо середнє значення для кожного критерію масштабування:

1. Вартість: $0.2 + 0.3 + 0.2 + 0.2 + 0.1 = 0.2$.
2. Підтримка: $0.2 + 0.2 + 0.2 + 0.2 + 0.2 = 0.2$.
3. Надійність: $0.4 + 0.3 + 0.4 + 0.4 + 0.4 = 0.38$.
4. Масштабованість: $0.2 + 0.2 + 0.2 + 0.2 + 0.3 = 0.22$.

Результати експертних оцінок представлені в таблиці 2.1

Таблиця 2.1

Експертні оцінки фахівців у галузі ІТ-інфраструктури

Критерій	Експерт 1	Експерт 2	Експерт 3	Експерт 4	Експерт 5	Середнє
Вартість	0.2	0.3	0.2	0.2	0.1	0.2
Підтримка	0.2	0.2	0.2	0.2	0.2	0.2
Надійність	0.4	0.3	0.4	0.4	0.4	0.38
Масштабованість	0.2	0.2	0.2	0.2	0.3	0.22
Сумарне значення	1.0	1.0	1.0	1.0	1.0	1.0

На основі результатів методу вагових коефіцієнтів можна зробити висновок, що критерію надійність було присвоєно найбільший пріоритет із середнім значенням 0.38, тобто найкраща модель, яка підходить для цього це горизонтальне масштабування. Це свідчить про те, що стабільність роботи системи в умовах можливих збоїв є ключовим фактором при виборі оптимальної моделі масштабування.

2.3 Кешування: Redis, Memcached, DragonflyDB

Кешування – це виділення пам'яті пристроєм або застосунку, для тимчасового зберігання даних, які ймовірно будуть використовуватись знову. Термін уперше було використано у 1967 році в публікації IBM Systems Journal для позначення поліпшення пам'яті, описаного як високошвидкісний буфер.

Структура кешу складається з ключа та значення. Ключ даних є ідентифікатором, за яким можна отримати інформацію, а значення – це фактична інформація, яка зберігається в кеші. Інформація може зберігатись або на постійному пристрої зберігання (наприклад, у базі даних) або у додатку. [18] Коли програма потребує доступу до даних, вона спершу перевіряє кеш-пам'ять, для того щоб визначити чи доступні ці дані. Якщо програма не знаходить дані в кеші, програма завантажує дані з основного сховища, наприклад з файлу чи веб-сторінки та додає їх у кеш для подальшого використання.

Прикладом цього використання можуть бути веб-браузери, які кешують інформацію, таку як зображення та скрипти. Коли користувач повторно відвідує веб-сайт, браузер швидко завантажує кешовані дані замість того, щоб завантажувати їх знову, що дозволяє прискорити завантаження сторінок.

Розглянемо основні практики використання:

- Кешування в пам'яті: додатки часто використовують пам'ять системи для кешування даних, які зберігаються на диску. Прикладом може бути СКБД, яка використовує кешування для роботи зі складними запитам.
- Кешування на стороні серверу: веб-додатки часто кешують дані на стороні серверу. Прикладом може бути веб-додаток, який кешує дані, що регулярно дістаються з бази даних.
- Кешування CDN: мережа доставки контенту зберігає кешовані дані на декількох серверах, які географічно розподілені для представлення контенту користувачам, які знаходяться найближче к ним.
- Кешування DNS: сервери доменних імен часто кешують дані для того щоб швидко перетворити імена хостів в адресу інтернет протоколу.

Розглянемо основні переваги кешування:

- Швидкий доступ до даних: значно скорочує час доступу до даних.
- Зменшення використання ресурсів: зменшує навантаження на основне джерело даних, знижуючи використання CPU. Це робить систему більш ефективною та масштабованою.
- Економія коштів: завдяки зменшенню використання ресурсів і підвищенню продуктивності, кешування допомагає знизити на підтримку основного джерела даних.
- Продуктивність: знижена затримка доступу до даних допомагає програмам виконувати завдання швидше.

До основних популярних технологій кешування високонавантажених сервісів можна віднести наступні: Redis, Memcached та DragonflyDB.

Redis – це програмне забезпечення з відкритим вихідним кодом для збереження даних у пам'яті, яке вирізняється своєю швидкістю та універсальністю. Вона може використовуватись як база даних, кеш та брокер повідомлень, забезпечуючи підтримку різноманітних структур даних, такі як: рядки, списки, множини, хеші. Redis часто застосовують у програмах, які потребують високої продуктивності та низької затримки доступу до даних, наприклад: аналіз у реальному часі, кешування даних та черги повідомлень. Її простота, швидкодія та вибір функцій зробили Redis популярним вибором серед розробників.

Memcached – це високошвидкісна, розподілена система кешування в оперативній пам'яті, яка використовується для прискорення динамічних веб-додатків шляхом зберігання даних у пам'яті. Вона функціонує як сховище «ключ-значення», де доступ до даних здійснюється за унікальними ключами.

DragonflyDB – це сучасне сховище даних у пам'яті, повністю сумісне з API Redis та Memcached. Воно розроблене для ефективного використання сучасного та хмарного обладнання, забезпечуючи значне підвищення продуктивності та зниження витрат у порівнянні з традиційними рішеннями.

Загальна схема кешування на прикладі Redis зображена на рис. 2.2

How Redis is typically used

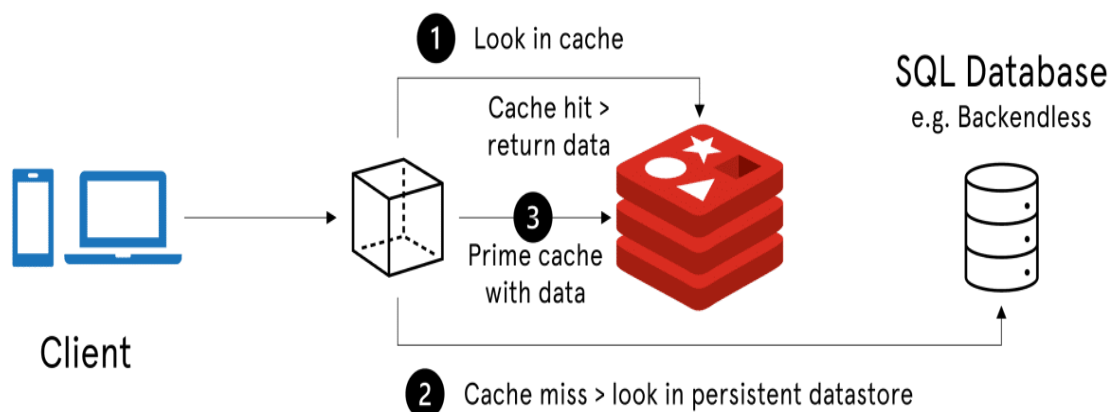


Рис 2.2 Загальна схема кешування Redis

Кешування є потужною технологією, яке забезпечує значні переваги для сучасних високонавантажених систем обробки. Воно дозволяє прискорити доступ до інформації, зменшити затримки у виконанні операцій та знизити навантаження на основі джерела даних. Проте вибір оптимального рішення залежить від специфічних вимог системи, таких як: швидкість, масштабованість, функціональність, вартість впровадження та відмовостійкість.

Застосуємо математичний метод аналізу ієрархій для порівняння Redis, Memcached та DragonflyDB, враховуючи технічні та економічні аспекти.

Метод аналізу ієрархій (АНР) – це базовий підхід до прийняття рішень, розроблений для поєднання раціональних та інтуїтивних аспектів, з метою вибору найкращого варіанту серед кількох альтернатив, оцінених за різними критеріями. У цьому процесі особа, яка приймає рішення, здійснює прості парні порівняння, які використовуються для визначення загальних пріоритетів та ранжування альтернатив. АНР допускає певну послідовність у судженнях та надає засоби покращення цієї послідовності. [19]

Алгоритм методу аналізу ієрархій:

1. Побудова ієрархій.
2. Парні порівняння елементів ієрархії та розрахунок локальних пріоритетів.

3. Розрахунок глобальних пріоритетів елементів ієрархії.

Метод аналізу ієрархій при побудові єдиної шкали для різних компонентів проблеми використовує міру ступеня впливу кожного фактора одного рівня на фактори верхнього рівня. Т. Сааті запропонував шкалу відносної важливості (значимості, переваги) представлену в таблиці 2.2

Таблиця 2.2

Шкала відношень (ступені значущості дій)

Ступінь значимості	Визначення
1	Однакова значимість (об'єкти рівноцінні)
3	Слабка значимість (один об'єкт дещо переважає іншого)
5	Істотна значимість (один об'єкт краще від іншого)
7	Очевидна значимість (один об'єкт значно краще від іншого)
9	Абсолютна значимість (один об'єкт абсолютно кращий)
2, 4, 6, 8	Проміжні судження про об'єкти

При використанні шкали , експерт, порівнюючи два об'єкти в змісті досягнення цілі, розташованої на вищому рівні ієрархії, потрібно поставити у відповідність цьому порівнянню число в інтервалі від 1 до 9 або обернене до нього.

Застосуємо метод аналізу ієрархій для вибору найкращої технології кешування для високонавантаженого сервісу:

Розглянемо три технології кешування (Redis, Memcached, DragonflyDB).

1. Побудова ієрархії: визначимо мету, критерії та альтернативи порівняння. Метою є вибір найкращої технології кешування для високонавантаженого сервісу. До критеріїв згідно з опитуванням ІТ-експертів слід віднести наступні: продуктивність, зручність використання та надійність. Серед альтернатив, які ми коротко дослідили вище відносяться: Redis, Memcached, DragonflyDB. Представимо це у вигляді ієрархічної структури, яка зображена на рисунку 1.4

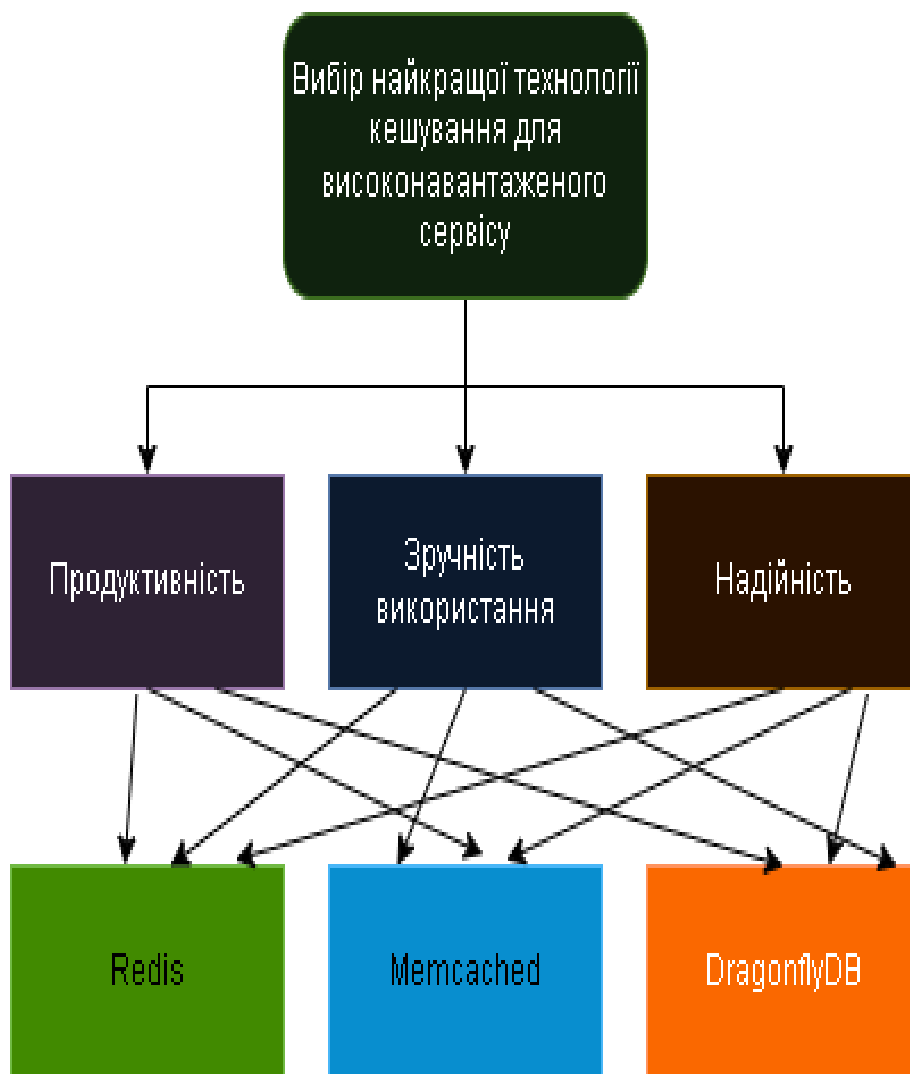


Рис 1.4 Ієрархічна структура вибору технології кешування

2. Парні порівняння елементів ієрархії, розрахунок локальних пріоритетів елементів ієрархії та оцінювання узгодженості експертних оцінок: побудуємо множину матриць попарних порівнянь (якщо елемент E1 домінує над елементом E2, то комірка матриці, що відповідає рядкові E1 і стовпчику E2

заповнюється цілим числом, а комірка, що відповідає E2 і E1 заповнюється оберненим до нього числом. Якщо елементи рівноцінні, то в симетричних комірках матриці попарних порівнянь ставляться одиниці). Скориставшись методом середнього геометричного визначаємо середнє геометричне по кожному рядку матриці парних порівнянь за формулою (2.2)

$$\omega_i = \sqrt[n]{a_{i1} \cdot a_{i2} \cdot \dots \cdot a_{im}} \quad (2.2)$$

Складемо елементи цього стовпця за формулою (2.3)

$$r = \omega_1 + \omega_2 + \omega_3 + \dots + \omega_n \quad (2.3)$$

Ділимо кожен елемент з елементів на отриману суму за формулою (2.4)

$$W_i = \frac{\omega_i}{r}, \quad (i = 1, 2, 3, \dots, n) \quad (2.4)$$

Для розрахунку $\lambda_{\max}$ визначається сума по кожному колонку матриці, яка множиться на відповідну компоненту вектору пріоритетів. Умовно це можна представити в наступному вигляді за формулою (2.5)

$$\begin{aligned} \lambda_{\max} = & (a_{11} + a_{21} + \dots + a_{n1}) * W_1 + (a_{12} + a_{22} + \dots + a_{n2}) \\ & * W_2 + (a_{13} + a_{23} + \dots + a_{n3}) * W_3 + \dots \\ & + (a_{1n} + a_{2n} + \dots + a_{nn}) * W_n \end{aligned} \quad (2.5)$$

Однорідність проведених оцінок визначається шляхом визначення відношення однорідності (ВО) за формулою (2.6)

$$BO = \frac{IO}{CO} \leq 20\%, \quad (2.6)$$

де IO - індекс узгодженості, CO - величина відповідна середньої випадкової узгодженості матриці такого порядку (таблиця 2.3).

Індекс узгодженості може бути визначений за такою формулою (2.7)

$$IC = \frac{\lambda_{\max} - n}{n - 1} \quad (2.7)$$

Таблиця 2.3

Відношення однорідності

Розмір матриці	1	2	3	4	5	6	7
Середнє значення оцінки однорідності	0	0	0.58	0.90	1.12	1.24	1.32

Отже, отримаємо такі результати розрахунків:

Таблиця 2.4

Обчислення за важливістю критеріїв

Критерій	Продуктивність	Зручність використання	Надійність	w	W	$\lambda_{\max}$	IC	BO
Продуктивність	1	3	5	2.466	0.637	3.037	0.0185	0.0319
Зручність використання	0.333	1.0	3	0.999	0.258			
Надійність	0.2	0.333	1	0.405	0.104			
Сума	1.533	4.333	9	3.871	1			

Обчислення за критерієм продуктивність

Альтернатива	Redis	Memcached	DragonflyDB	w	W	λ max	IC	BO
Redis	1	3	2	1.817	0.539	3.0089	0.0005	0.00086
Memcached	0.333	1	0.5	0.550	0.163			
DragonflyDB	0.5	2	1	1.0	0.296			
Сума	1.833	6.0	3.5	3.367	1.0			

Таблиця 2.6

Обчислення за критерієм зручність використання

Альтернатива	Redis	Memcached	DragonflyDB	w	W	λ max	IC	BO
Redis	1	4	6	2.884	0.691	3.052	0.026	0.044
Memcached	0.25	1	3	0.908	0.217			
DragonflyDB	0.166	0.333	1	0.380	0.091			
Сума	1.416	5.333	10	4.173	1.0			

Таблиця 2.7

Обчислення за критерієм надійність

Альтернатива	Redis	Memcached	DragonflyDB	w	W	λ max	IC	BO
Redis	1	0.25	0.167	0.346	0.089	2.714	-0.143	-0.247
Memcached	4	1	0.5	1.259	0.323			
DragonflyDB	6	2	1	2.289	0.587			
Сума	11	3.25	1.167	3.894	1			

3. Розрахунок глобальних пріоритетів елементів ієрархії: отримані значення вектору пріоритетів (W) являють собою систему локальних критеріїв, на основі яких розраховується глобальний пріоритет альтернативи по кожному варіанту за формулою 2.8.

$$Z_j = \sum_{i=1}^m W_j(i) \cdot W(i), \quad (2.8)$$

де $W_j(i)$ - пріоритет j - ой альтернативи по i - ому критерію, $W(i)$ - пріоритет або значимість i - ого критерію.

Таблиця 2.8

Обчислення глобальних пріоритетів

Альтернатива	Продуктивність	Зручність використання	Надійність	Глобальний пріоритет
Redis	0.539	0.691	0.089	0.531
Memcached	0.163	0.217	0.323	0.194
DragonflyDB	0.296	0.091	0.587	0.275

За результатами розрахунків, визначено, що кешування за допомогою технології «Redis» є найкращою.

2.4. Партиціювання на рівні СКБД PostgreSQL

У сучасних високонавантажених системах ефективність бази даних (БД) відіграє ключову роль в процесах збереження і обробки інформації. Однією з найбільш поширених технологій для організації і управління даними є реляційні системи керування базами даних (РСКБД), а основним засобом звернення до СКБД є SQL-запити. Зі збільшенням обсягів даних в базі даних і, головне, складності структури бази даних та підвищенні вимог щодо первинної обробки даних на рівні СКБД зростає складність SQL-запитів до БД. Як наслідок,

швидкість доступу до даних може суттєво зменшуватись та впливати на загальне функціонування системи. Тривалі запити не тільки споживають значну кількість системних ресурсів, що може спричинити перевантаження сервісів, але також здатні викликати тимчасове блокування таблиць та, навіть, призвести до втрати або пошкодження даних на рівні додатків. Популярним шляхом вирішення зазначеної проблеми є підвищення вимог до апаратної частини, але це не завжди дає очікуваний результат.

Загалом, на поточний момент визначено основні проблеми, які зустрічаються при роботі з великими обсягами даних в БД:

- Відсутність індексів або їх неефективне використання. Індеси дозволяють прискорювати пошук по таблицях, але значна складність структур даних і запитів може призводити до необхідності створення надмірної кількості індексів, що призводить до розростання бази даних та збільшенням вимог до обчислювальних ресурсів.

- Значне перевищення обсягів даних у БД над обсягами вибірки даних у конкретних запитах. Партиціювання таблиць дозволяє розділити великі таблиці на менші частини за певним критерієм, наприклад, за датою або за діапазонами значень, що дозволяє СКБД виконувати запити над окремими партиціями БД, а не БД у цілому.

- Нераціональна або недостатня конфігурація серверів бази даних. Параметри СКБД, такі як розмір буферів, кешування даних в оперативній пам'яті, які не відповідають потребам конфігурації БД та складності запитів, призводять до уповільнення обробки запитів.

- Невідповідність апаратної частини СКБД обсягам даних та складності запитів. Недостатня оперативна пам'ять, перевантаження процесорів через численні одночасні запити може призвести до уповільнення виконання операцій. В таких випадках краще використовувати шардування. Це дозволить розподілити дані між кількома серверами для горизонтального масштабування, для підвищення продуктивності та балансування навантаження.

– Недостатня або надмірна нормалізація даних БД. Якщо база даних має не оптимізовану структуру, це може викликати затримки у виконанні запитів. Надмірна кількість зв'язків між таблицями або використання не відповідних типів даних негативно впливають на продуктивність SQL.

Отже, на поточний момент є багато способів збільшення швидкості виконання запитів – це використання індексів, оптимізованих структур даних, шардування та партиціювання даних, алгоритми з'єднання та обмеження вибірки. Ці засоби частково впроваджені в СКБД їх розробниками і також можуть впроваджуватись проектантами БД. Для створення ефективних SQL-запитів найчастіше використовуваним способом є створення декількох варіантів запиту та отримання плану їх виконання для подальшого аналізу та визначення оптимальних варіантів, а також правильне використання вбудованих інструментів.

Дослідження в області оптимізації бази даних підкреслюють важливість використання партиціювання для високонавантажених продуктивних систем.

Автор Танаєма П. та інші стверджують, що продуктивність бази даних є одним із основних компонентів підтримки стійкості системи. У дослідженні вони провели проектування та оцінку таблиць розділів, яка потім почала використовуватись у продуктовому середовищі. Результати показали, що таблиці з партиціями мають кращу продуктивність, але наголосили, що вставка працює погано порівняно зі звичайними таблицями.

Науковець Беднарчук П. показав, що поширеною проблемою доступу до БД є уповільнення звітів у базах даних з часом. На початку система працює швидко та ефективно, але з часом вона сповільнюється і без впровадження відповідних рішень система може перестати відповідати вимогам. Дослідник довів, що завдяки партиціюванню можливо досягти однакової продуктивності системи за весь період експлуатації, не зважаючи на збільшення обсягів накопичених даних та ускладнення запитів до БД.

Автор Мустафа С. та інші вважають, що багато факторів впливають на рішення щодо розділення даних на партиції. Це розмір бази даних, тип даних,

тип запитів та частота запитів. Дослідження показали, що абсолютного ультиматуму не існує, а конкретне рішення потрібно вибирати емпіричним шляхом.

З огляду на дослідження різних науковців можна зробити висновок, що партиціювання є одним із ключових методів оптимізації продуктивності баз даних, особливо в умовах високого навантаження та роботи з великими обсягами інформації.

Розподілення таблиць SQL – це процес розподілу даних таблиці на кілька таблиць. Це дозволяє покращити продуктивність запитів під час сканування даних з одного або декількох розділів, а не при читанні чи пошуку по всій таблиці (або її індексу). Якщо фільтр запиту охоплює одну або декілька партицій, багато з них виключається зі сканування. Загальна схема партиціювання таблиці зображена на рис 2.5

В PostgreSQL є три типи декларативного партиціювання таблиці: розбиття по діапазону значень, розподіл списку, розподілення хешу, це зображено на рис 2.6 Розбиття за діапазоном дозволяє ділити таблицю на частини на основі діапазонів значень одного або кількох стовпців. За списком на основі конкретних значень стовпця, де кожна партія містить записи, що відповідають певному набору значень. Розбиття за хешем дозволяє розподілити значення на основі хеш функції стовпця. В нашому випадку ми використовуємо розбиття даних за діапазоном. [20]

За допомогою експерименту визначимо вплив кількості партицій таблиці на продуктивність операції вибору даних.

Проведемо експеримент, який досліджує вплив кількості партицій таблиці на продуктивність операції вибору даних. Було вибрано реляційну базу даних PostgreSQL. Характеристики комп'ютера, на якому проводяться експерименти, описані у таблиці 2.9.

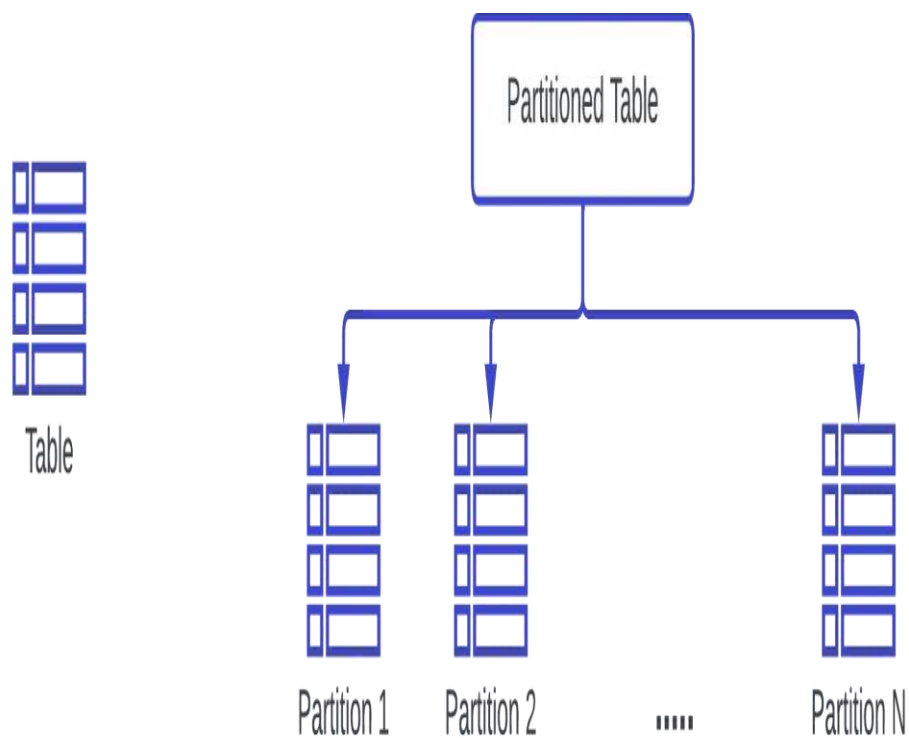


Рис 2.5 Загальна схема партиціювання таблиці

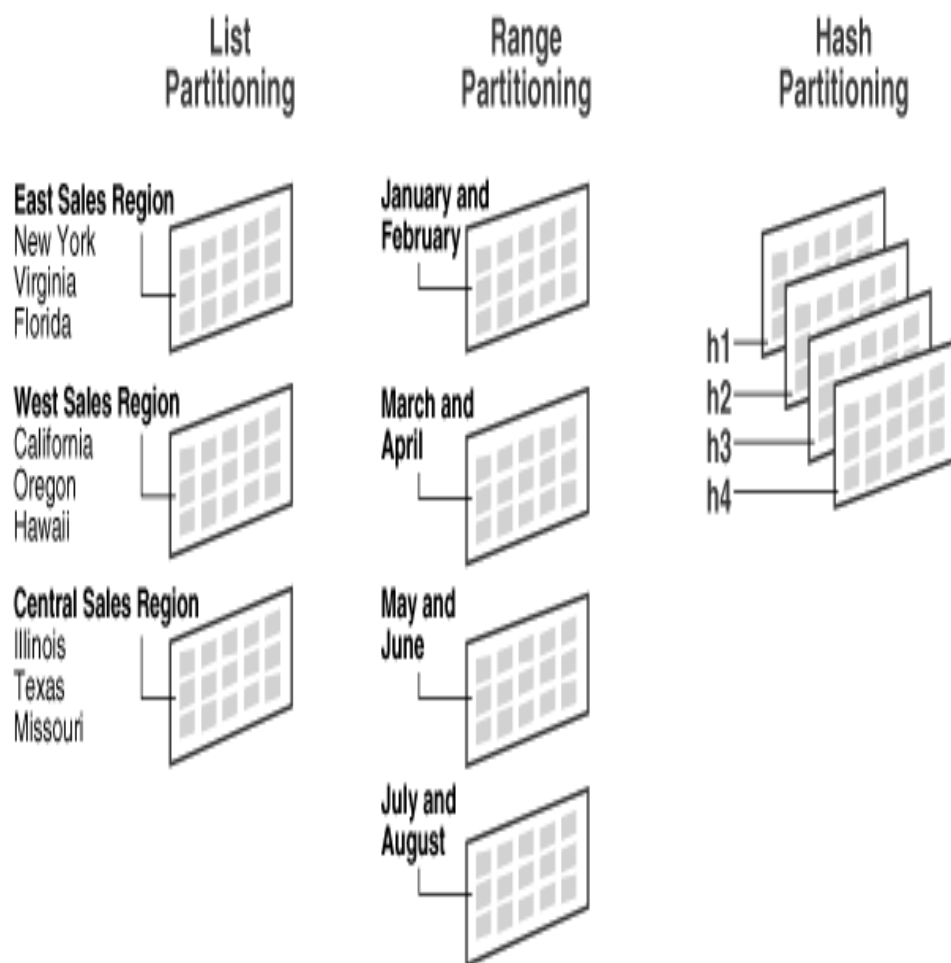


Рис 2.6 Типи партиціювання таблиці

Опис комп'ютера, що використовується

Характеристика	Значення
Процесор	Десяти ядерний 2021 M1 PRO
Оперативна пам'ять	16 ГБ
Операційна система	MacOS Ventura 13.0
Диск	SSD 256 gb, запис та читання до 3,3 ГБ/с

Було згенеровано таблицю користувачів, яка імітує систему з 60 мільйонами записів та має рівномірний розподіл по 5 мільйонів за датою реєстрації протягом 2023 року. У ході виконання досліджень застосуємо діапазон значень у якості логічного розділення даних для поля дата реєстрації. За допомогою команди EXPLAIN ANALYZE ми можемо побачити час виконання запитів з використанням партицій.

Хід виконання:

1. Створимо таблицю користувачів:

```
CREATE TABLE users_no_partition (
  id INT,
  first_name VARCHAR(25) NOT NULL,
  date_joined DATE NOT NULL
) PARTITION BY RANGE(date_joined).
```

2. Заповнимо 60 млн. користувачів з датою реєстрації 2023 року:

```
INSERT INTO users_no_partition (
  id, first_name, date_joined) VALUES(GENERATE_SERIES(1, 5000000),
  'Bob',(TIMESTAMP '2023-01-01' + RANDOM() * (TIMESTAMP '2023-01-31' -
  TIMESTAMP '2023-01-01'))::DATE.
```

3. Створимо партиції для таблиці, з різним діапазоном значень для поля дата реєстрація користувача. Приклад запиту з квартальним діапазоном:

```
CREATE TABLE users PARTITION of users for
VALUES FROM ('2023-01-01') to ('2023-04-01').
```

4. Виконуємо запит з пошуком користувачів, які зареєструвались у грудні, не змінюючи дати пошуку для кожного діапазону розбиття таблиці:

```
SELECT COUNT(*) FROM users_no_partition WHERE date_joined >= '2023-12-01' AND date_joined <= '2023-12-31'.
```

5. Повторюємо запит пошуку для різних діапазонів розбиття: місяць, 3 місяці, квартал, півроку.

Результати виконання запитів представлені в таблиці 2.10 та на рисунку 2.7

Таблиця 2.10

Час виконання запиту вибору користувачів з різним діапазоном розбиття партицій

Діапазон розбиття	Значення
Відсутній	1500 мілісекунд
Тиждень	470 мілісекунд
Місяць	160 мілісекунд
3 Місяці	636 мілісекунд
Квартал	811 мілісекунд
Півроку	1266 мілісекунд

Партиціювання за місяцями забезпечило найкращу продуктивність, скоротивши час виконання на 89,3% порівняно з відсутністю партицій. Партиція за 3 місяці скоротила час на 57,6%, а за квартал на 45,9%. Найгірший результат дала партиція за півроку, яка скоротила час виконання лише на 15,6%, що вказує на значне зниження ефективності при великих інтервалах партиціювання. Розбиття таблиць на партиції за тижнями є повільнішим порівняно з розбиттям за місяцями, що зумовлено збільшенням кількості партицій і, як наслідок, зростанням витрат на їх обробку.

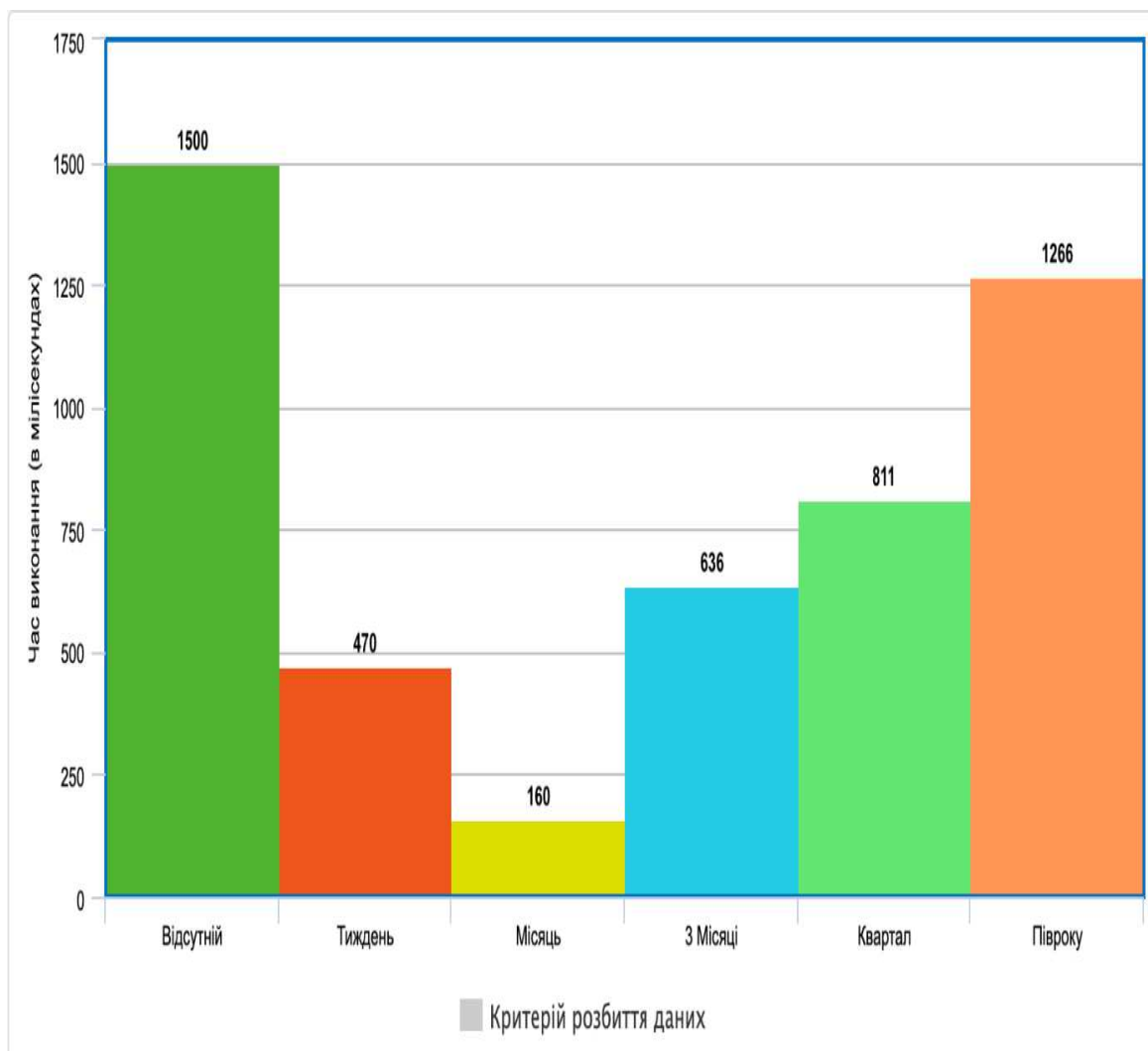


Рис 2.7 Діаграма виконання запитів з критеріями розбиття даних і відповідним часом виконання

Спочатку система може функціонувати швидко та ефективно, але зі збільшенням даних її швидкодія погіршується. У ході дослідження було проведено аналіз ефективності партиціювання таблиць для підвищення швидкості виконання SQL-запитів. Результати експериментів на прикладі великої таблиці користувачів показали, що партиціювання за короткі інтервали, зокрема за місяцями, забезпечує найбільше скорочення часу виконання запитів. Для досягнення найкращих результатів зменшення часу виконання SQL необхідно експериментувати з критерієм розподілу даних. Таким чином, дослідження підтвердило, що правильний вибір інтервалу партиціювання може

суттєво покращити швидкість доступу до даних і підвищити загальну ефективність бази даних.[20]

2.5. Висновки до розділу 2

У цьому розділі було запропоновано концепцію комплексної оптимізації високонавантажених сервісів, який поєднує три ключові підходи: масштабування, кешування та партиціювання даних. На основі проведених досліджень та аналізу існуючих рішень, розроблено модель, яка інтегрує ці підходи у єдину систему.

Спочатку система працює швидко та ефективно, але зі збільшенням користувачів продуктивність сервісу починає знижуватися. Було визначено, що оптимальна стратегія масштабування залежить від специфіки сервісу та вимог до його продуктивності. Метод вагових коефіцієнтів дозволив об'єктивно оцінити ефективність кожного підходу.

Для оптимізації операцій з даними розглянуто технології кешування, такі як Redis, Memcached та DragonflyDB. Метод аналізу ієрархій допоміг обґрунтувати вибір технології на основі ключових критеріїв, враховуючи продуктивність, надійність та зручність використання.

Партиціювання баз даних, зокрема в PostgreSQL є ефективним методом для підвищення продуктивності запитів у системах із великим обсягом даних. Експериментальний аналіз підтвердив суттєве скорочення часу виконання запитів за рахунок партиціювання даних, що зменшує навантаження на БД.

Об'єднання підходів масштабування, кешування та партиціювання створює ефективну екосистему для забезпечення продуктивності високонавантажених систем. Застосування таких моделей оптимізації вимагає ретельного планування, але дозволяє досягти значного підвищення стабільності, продуктивності та масштабованості сервісів, що особливо важливо в умовах постійного зростання обсягу даних та навантаження на систему. Це забезпечує

можливість адаптації сервісів до нових викликів, зберігаючи їхню ефективність і надійність.

У наступному розділі розробимо моделі оптимізації високонавантажених сервісів. Основна увага буде приділена створенню ефективних моделей масштабування, кешування з використанням технології Redis та використанню мікросервісної архітектури.

РОЗДІЛ 3. РОЗРОБКА МОДЕЛЕЙ ОПТИМІЗАЦІЇ ВИСОКОНАВАНТАЖЕНОГО СЕРВІСУ

3.1 Проектування та реалізація мікросервісної архітектури сервісу

Архітектура програмного забезпечення показує взаємодії компонентів інтерфейсу з внутрішніми процесами програми. Головна ідея архітектури полягає в спрощенні процесу сприйняття системи. Архітектура програмного забезпечення дозволяє створити чітку структуру, за якою зручно працювати розробникам.

Для розробки програмного забезпечення найчастіше застосовують шаблони. Вони дозволяють вирішувати однотипні завдання швидшим способом, що позитивно впливає на час розробки та спрощує багато процесів. Для розробки сервісу пошуку роботи використаємо шаблон мікросервісної архітектури, зобразимо це на діаграмі розміщення.

Діаграма розміщення програмного забезпечення (Software deployment diagram) – це тип діаграми, що використовується в області програмного забезпечення для відображення того, як програмні компоненти взаємодіють між собою та з апаратним забезпеченням в процесі розгортання системи.

Ця діаграма показує фізичну структуру системи, її компоненти, а також залежності між цими компонентами. Кожен компонент системи може бути представлений у вигляді віртуальної машини або фізичного сервера. Діаграма розміщення дозволяє визначити, на яких компонентах буде запущена система, які компоненти будуть взаємодіяти між собою, а також які сервіси будуть доступні користувачам. Результат діаграми розміщення зображено на рисунку 3.1

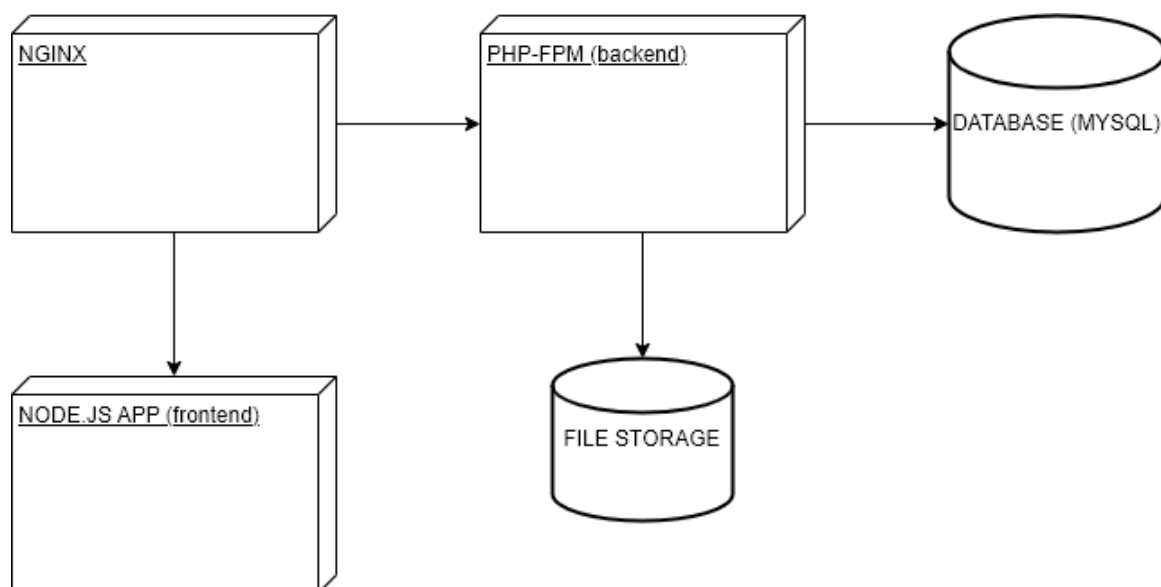


Рис 3.1 Діаграма розміщення мікросервісної архітектури сервісу пошуку роботи

Діаграма надає чітке уявлення про те, як компоненти (фронтенд, бекенд, база даних, файлове сховище) взаємодіють між собою. Фронтенд є інтерфейсом користувача і початковою точкою взаємодії. Ці запити передаються до бекенду у вигляді HTTP-запитів через API. Фронтенд також отримує результати від бекенду, форматуючи їх для відображення.

Бекенд приймає запити від фронтенду і відповідає за їх обробку. Він виконує перевірку введених даних, реалізує бізнес-логіку (наприклад, перевіряє фільтри для пошуку), звертається до бази даних для отримання необхідної інформації, а також може зберігати або отримувати файли з файлового сховища.

База даних відповідає за зберігання структурованих даних, таких як профілі користувачів, вакансії, резюме тощо. Бекенд звертається до бази даних для виконання CRUD-операцій (створення, читання, оновлення, видалення). База даних використовує механізми оптимізації, такі як індекси, щоб забезпечити швидкий доступ до даних.

Файлове сховище використовується для зберігання великих об'єктів, таких як резюме у форматі PDF, фотографії компаній або інші документи. Коли бекенд потребує доступу до таких файлів, він надсилає запити до сховища, отримує

посилання для завантаження або завантажує нові файли. Потім ці посилання можуть бути передані фронтенду для прямого доступу до файлів.

Перед початком розробки моделей оптимізації високонавантаженого сервісу необхідно обрати відповідні мови програмування.

Для розробки backend-частини було обрано Laravel – потужний і гнучкий фреймворк для веб-розробки на основі PHP. Laravel відомий своєю модульністю, масштабованістю та зрозумілим синтаксисом, що забезпечує швидку розробку складних веб-додатків. Основні можливості Laravel включають вбудовану систему маршрутизації, ORM Eloquent, шаблонізатор Blade, підтримку сесій, аутентифікації користувачів, а також велику кількість розширень і пакетів.

Фреймворк дозволяє конфігурувати проект через спеціальні конфігураційні файли, що забезпечує гнучке налаштування під вимоги конкретного проекту. Laravel ідеально підходить для створення додатків будь-якої складності: від простих блогів до багатофункціональних систем управління контентом чи платформ електронної комерції. Серед ключових переваг Laravel – стабільність, зручність для розробників та активна спільнота, яка постійно розширює можливості фреймворку, додаючи нові функції й пакети. Фреймворк підтримує шаблон MVC і має високоякісну документацію, що дозволяє швидко освоїти його та приступити до розробки.

Сучасні веб-додатки здебільшого реалізуються у форматі SPA (Single Page Application) – односторінкових додатків. SPA працюють, завантажуючи всі необхідні ресурси (HTML, CSS, JavaScript) під час першого відкриття сторінки, а подальша взаємодія здійснюється за допомогою AJAX-запитів без перезавантаження сторінки. Це забезпечує більш інтерактивний і зручний користувацький інтерфейс, пришвидшує завантаження сторінок і підвищує ефективність додатку. SPA широко використовуються в онлайн-магазинах, соціальних мережах і складних веб-платформах.

Для реалізації frontend-частини обрано Vue.js – прогресивну JavaScript-бібліотеку для створення користувацьких інтерфейсів. Vue.js є ідеальним вибором для розробки SPA та інших динамічних додатків завдяки

компонентному підходу, інтуїтивно зрозумілому синтаксису та легкості у вивченні. Ця бібліотека дозволяє створювати як прості, так і складні інтерфейси, забезпечуючи інтерактивність і гнучкість у реалізації функціоналу.

Крім того, Vue.js має багату екосистему з великою кількістю інструментів та доповнень, що спрощує і пришвидшує процес розробки. Завдяки популярності та активній спільноті Vue.js забезпечує можливість створювати динамічні додатки з високим рівнем взаємодії з користувачем. Таким чином, Laravel та Vue.js у поєднанні формують потужну основу для розробки сучасного, ефективного та зручного веб-додатку.

3.2 Кешування сервісу з використанням технології Redis

Кешування – це один із найефективніших способів оптимізації роботи сервісу, який дозволяє зменшити кількість запитів до бази даних і збільшити швидкість відповіді на запити. Laravel має вбудовану підтримку Redis, що робить інтеграцію кешування швидкою і простою. Розробимо реалізацію кешування сервісу з використанням Redis у Laravel.

Laravel уже має підтримку Redis за замовчуванням. У файлі `config/database.php` секція `redis` повинна містити такі налаштування, які зображені на рисунку 3.2

```
'redis' => [
    'client' => env('REDIS_CLIENT', 'phpredis'),

    'options' => [
        'cluster' => env('REDIS_CLUSTER', 'redis'),
        'prefix' => env('REDIS_PREFIX', Str::slug(env('APP_NAME', 'laravel'), '_').'_database_'),
    ],

    'default' => [
        'url' => env('REDIS_URL'),
        'host' => env('REDIS_HOST', '127.0.0.1'),
        'username' => env('REDIS_USERNAME'),
        'password' => env('REDIS_PASSWORD'),
        'port' => env('REDIS_PORT', '6379'),
        'database' => env('REDIS_DB', '0'),
    ],

    'cache' => [
        'url' => env('REDIS_URL'),
        'host' => env('REDIS_HOST', '127.0.0.1'),
        'username' => env('REDIS_USERNAME'),
        'password' => env('REDIS_PASSWORD'),
        'port' => env('REDIS_PORT', '6379'),
        'database' => env('REDIS_CACHE_DB', '1'),
    ],
],
```

Рис 3.2 Налаштування кешування Redis

У контролері реалізуємо кешування всіх вакансій із використанням методу `Cache::remember`. Метод `Cache::remember` перевіряє наявність даних у кеші під ключем `all_vacancies`. Якщо дані є, вони повертаються з кешу. Якщо даних немає, виконується запит до бази даних, який отримує всі вакансії разом із пов'язаними категоріями (`category`) і компаніями (`company`). Результат запиту зберігається у Redis на 3600 секунд (1 година). Програмний код зображений на рисунку 3.3

```
public function index()
{
    // Кешуємо всі вакансії
    $vacancies = Cache::remember('all_vacancies', 3600, function () {
        return Vacancy::with(['category', 'company'])->get();
    });

    return response()->json($vacancies);
}
```

Рис 3.3 Програмний код кешування отримання вакансій

Запити до бази на отримання локацій можуть бути повторюваними, тому доцільно кешувати результат такого запиту, щоб зменшити кількість звернень до бази даних і прискорити час відповіді. Програмний код зображений на рисунку 3.4

```
public function locations()
{
    // Кешуємо унікальні локації вакансій
    $locations = Cache::remember('vacancy_locations', 3600, function () {
        return Vacancy::select('location')
            ->orderBy('location', 'asc')
            ->distinct()
            ->get()
            ->map(function ($value) {
                return $value->location;
            })->push("Yci");
    });

    return response()->json($locations);
}
```

Рис 3.4 Програмний код кешування отримання локацій

Реалізація кешування з використанням Redis значно покращує продуктивність сервісу. Кешування дозволяє зменшити кількість запитів до бази даних, забезпечити швидкий доступ до даних і підтримувати високу швидкодію навіть при значному навантаженні.

3.3. Контейнеризація та оркестрація сервісу з використанням Docker та Kubernetes для масштабування сервісів

Контейнеризація – це метод розгортання програмного забезпечення, який упаковує код програми разом із усіма необхідними залежностями (бібліотеками, файлами конфігурації) у незалежний блок, що називається контейнером. На відміну від традиційної віртуалізації, яка віртуалізує всю операційну систему, контейнери спільно використовують ядро операційної системи. Завдяки цьому вони є легкими та портативними, що дозволяє запускати їх стабільно на будь-якій інфраструктурі. Контейнеризація сервісів відбувається за допомогою Docker.

Docker – це відкрита платформа для розробки, доставки та запуску додатків. Вона дозволяє відокремлювати ваші додатки від інфраструктури, що дає змогу швидко розгортати програмне забезпечення. Завдяки Docker ви можете управляти інфраструктурою так само, як керуєте своїми сервісами. Використовуючи методології Docker для доставки, тестування та розгортання коду, можна суттєво скоротити затримку між написанням коду і його запуском у виробничому середовищі.

Kubernetes – це портативна, розширювана, відкрита платформа для управління робочими навантаженнями та сервісами, яка забезпечує підтримку декларативної конфігурації та автоматизації. Платформа має велику й швидко зростаючу екосистему. Сервіси, підтримка та інструменти Kubernetes широко доступні, що робить її популярним вибором для організацій, які прагнуть масштабованого та ефективного управління сервісами.

Контейнери – це чудовий спосіб упакувати та запускати ваші додатки. Однак у виробничому середовищі важливо керувати контейнерами, які запускають додатки, та забезпечувати відсутність простоїв. Наприклад, якщо контейнер виходить з ладу, інший контейнер повинен автоматично запуститися, щоб підтримувати роботу. Саме тут на допомогу приходять платформи оркестрації контейнерів, такі як Kubernetes, які автоматизують завдання розгортання, масштабування, моніторингу та відновлення контейнерів. Загальна схема оркестрації зображена на рисунку 3.5

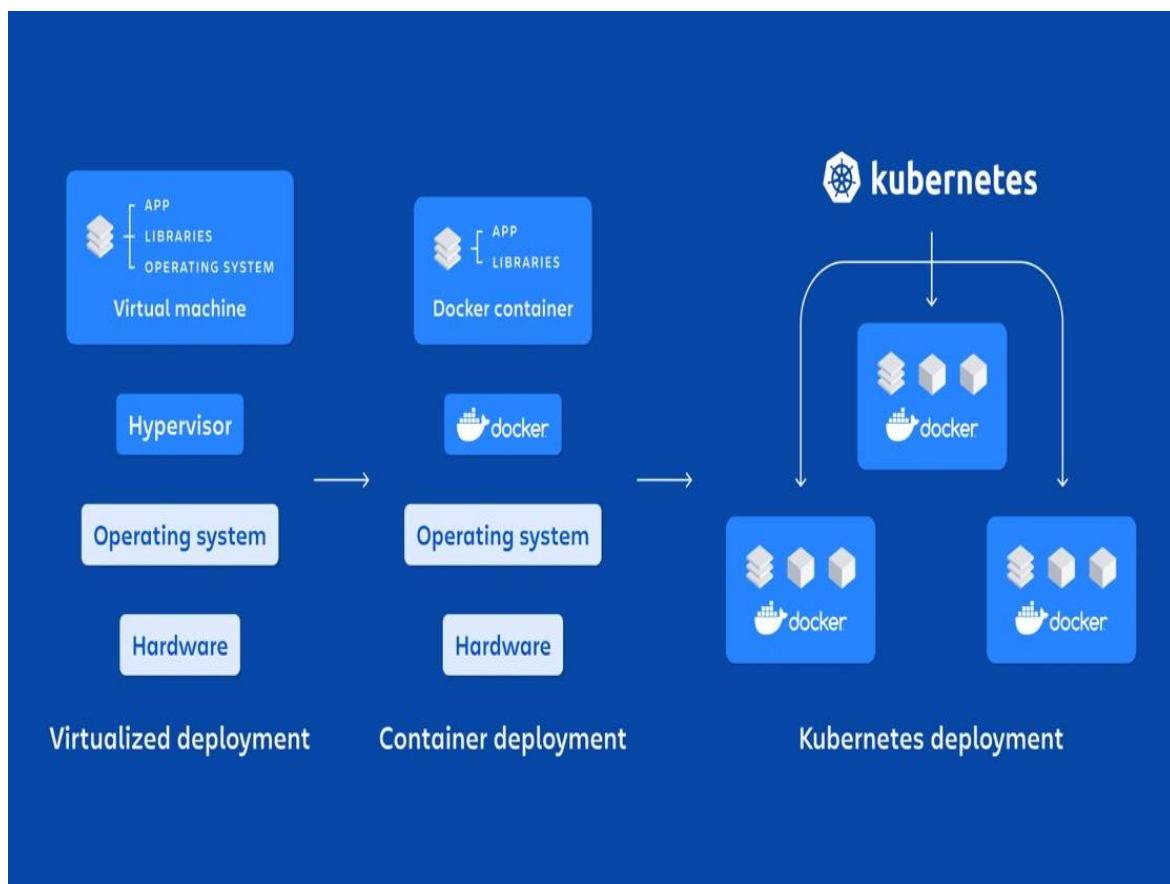


Рис 3.5 Загальна схема оркестрації

У сучасних високонавантажених сервісах ключову роль відіграє ефективне розгортання та управління сервісами, які забезпечують основну логіку додатків. Зростаюча популярність мікросервісної архітектури, контейнеризації та оркестрації спонукає розробників використовувати інструменти, які забезпечують гнучкість, масштабованість та надійність програмного забезпечення. Для цього застосуємо модель оптимізації

масштабування з використанням Dockerfile та Kubernetes. Найменшою одиницею в Kubernetes є под. Под (Pod) — це одиниця розгортання в Kubernetes, яка представляє один або декілька контейнерів, що спільно використовують мережевий простір і сховище. Под є абстракцією над контейнерами, що забезпечує їхню гнучкість, портативність та можливість оркестрації.

Для того щоб помістити наш сервіс в Kubernetes, необхідно створити Dockerfile, який містить інструкції для створення контейнеру.

Створимо Dockerfile для контейнеризації backend сервісу, який зображений на рисунку 3.6

```
FROM php:8.1-fpm-alpine
RUN curl -sSLf \
    -o /usr/local/bin/install-php-extensions \
    https://github.com/mlocati/docker-php-extension-installer/releases/latest/download/install-php-extensions && \
    chmod +x /usr/local/bin/install-php-extensions && \
    install-php-extensions gd bcmath zip redis pdo_pgsql opcache
RUN apk update && apk add --no-cache supervisor nginx nano
WORKDIR /var/www/app
COPY .docker/php.ini /usr/local/etc/php/php.ini
COPY .docker/supervisord.conf /etc/supervisord.conf
COPY ./
RUN cp .env.example .env
#Add composer
RUN php -r "copy('https://getcomposer.org/installer', 'composer-setup.php');" && \
    php composer-setup.php && \
    php -r "unlink('composer-setup.php');" && \
    mv composer.phar /usr/local/bin/composer
RUN composer install -d /var/www/app
RUN chmod -R 777 /var/www/app/storage
RUN chmod -R 777 /var/www/app/bootstrap
RUN php /var/www/app/artisan storage:link
#Start Supervisor
CMD ["/usr/bin/supervisord", "-n"]
```

Рис 3.6 Dockerfile backend сервісу

Цей Dockerfile створює контейнер на базі образу php:8.1-fpm-alpine для Laravel додатку. Використовується інструмент install-php-extensions для встановлення розширень gd, bcmath, zip, redis, pdo_pgsql і opcache. Додатково встановлюються supervisor, nginx і nano через apk. Робоча директорія

налаштовується як `/var/www/app`. Копіюються конфігураційні файли `php.ini` і `supervisord.conf`, а також програмний код додатку. Налаштовується середовище Laravel шляхом копіювання `.env`, встановлення залежностей через Composer, встановлення прав доступу до `storage` і `bootstrap`, а також створення символічного посилання для `storage`. Контейнер запускається через Supervisor для паралельного керування процесами PHP-FPM і Nginx.

Далі необхідно створити `yaml` файли для розгортання контейнеру в кластері Kubernetes.

Deployment – це ресурс Kubernetes, що відповідає за автоматизацію створення, оновлення, масштабування та управління життєвим циклом подів. Він дозволяє розгорнути додатки із заданою кількістю реплік, забезпечуючи їх доступність і стабільність. Deployment використовує стратегію оновлення для плавного впровадження нових версій без переривання роботи сервісу.

Створимо `yaml` файл, який визначає розгортання Laravel сервісу з використанням образу `laravel-app:v2`. Deployment створює один под, який прослуховує порт 9000 через `php-fpm`. Підключаються томи для конфігурації Nginx, Supervisor і зберігання логів. Стратегія оновлення `RollingUpdate` забезпечує поступове оновлення сервісу, де одночасно може бути недоступним лише один под. Контейнер отримує змінні середовища з `ConfigMap backend-config`, що дозволяє легко змінювати налаштування без створення нового образу. Результат конфігурації зображений на рисунку 3.7

```

apiVersion: apps/v1
kind: Deployment
metadata:
  name: backend
spec:
  selector:
    matchLabels:
      app: backend
  replicas: 1
  minReadySeconds: 15
  strategy:
    type: RollingUpdate
    rollingUpdate:
      maxUnavailable: 1
      maxSurge: 1
  template:
    metadata:
      labels:
        app: backend
    spec:
      terminationGracePeriodSeconds: 60
      volumes:
        - name: nginx-config-volume
          configMap:
            name: backend-config-nginx
        - name: backend-supervisor
          configMap:
            name: backend-config-supervisor
        - name: shared-data
          emptyDir: {}

```

Рисунок 3.7 Конфігурація Deployment backend сервісу

Service — це ресурс Kubernetes, що забезпечує доступ до подів та балансування навантаження між ними. Він дозволяє створювати стабільну мережеву точку доступу для подів, незалежно від їхньої кількості чи IP-адрес. Service підтримує маршрутизацію трафіку на основі міток (labels), що пов'язують його з відповідними подами. Backend-service забезпечує доступ до бекенд сервісу через внутрішній порт 80. Він направляє трафік до подів, які мають мітку app: backend. Це дозволяє підключати бекенд до інших компонентів кластера, таких як фронтенд або бази даних. Service працює як внутрішній рівень маршрутизації в кластері, гарантує стабільність з'єднань і спрощує управління мережею. Результат конфігурації зображений на рисунку 3.8

```

apiVersion: v1
kind: Service
metadata:
  name: backend-service
spec:
  ports:
    - targetPort: 80
      port: 80
  selector:
    app: backend

```

Рис 3.8 Конфігурація Service backend

Ingress Nginx – це контролер, який реалізує ресурс Ingress у Kubernetes для маршрутизації HTTP(S)-трафіку до внутрішніх сервісів кластера. Він працює як веб-сервер і балансувальник навантаження, що приймає запити від клієнтів і спрямовує їх до відповідних сервісів або подів залежно від визначених правил. Ingress дозволяє використовувати доменні імена, маршрутизацію за шляхами, SSL/HTTPS та інші додаткові функції для побудови складних архітектур. Контролер Ingress Nginx інтегрується в кластер Kubernetes, перетворюючи правила Ingress у конфігурацію Nginx і виконуючи їх у реальному часі. Створимо файл, який визначає Ingress Nginx конфігурацію. Ingress налаштовує маршрутизацію для двох доменів.

Для домену magistracy.example усі запити перенаправляються до backend-service, що відповідає за обробку бекенд-запитів. Для домену magistracy-balancing.example запити перенаправляються до nginx-service. Обидва сервіси прослуховують трафік на порту 80.

Ingress використовує клас nginx, що вказує на використання Ingress Nginx як контролера. Додані анотації оптимізують маршрутизацію: переписування шляхів (rewrite-target) дозволяє зберігати правильний формат URL, таймаути (proxy-read-timeout і proxy-send-timeout) забезпечують стабільну роботу

довготривалих запитів, а обмеження розміру тіла запиту (proxy-body-size) дозволяє обробляти великі запити до 15 мегабайт. Також активований редірект між версіями з www і без www, що спрощує роботу з доменами. Результат конфігурації зображений на рисунку 3.9

```

metadata:
  annotations:
    nginx.ingress.kubernetes.io/rewrite-target: /$1
    nginx.ingress.kubernetes.io/proxy-read-timeout: "3600"
    nginx.ingress.kubernetes.io/proxy-send-timeout: "3600"
    nginx.ingress.kubernetes.io/proxy-body-size: 15m
    nginx.ingress.kubernetes.io/from-to-www-redirect: "true"
spec:
  ingressClassName: nginx
  rules:
  - host: magistracy.example
    http:
      paths:
      - path: /?(.*)
        pathType: ImplementationSpecific
        backend:
          service:
            name: backend-service
            port:
              number: 80
  - host: magistracy-balancing.example
    http:
      paths:
      - path: /?(.*)
        pathType: ImplementationSpecific
        backend:
          service:
            name: nginx-service
            port:
              number: 80

```

Рисунок 3.9 Конфігурація Ingress Nginx

Цей файл забезпечує гнучку маршрутизацію запитів, дозволяючи відокремлювати функції бекенду й додаткових сервісів за допомогою окремих доменів і правил. Налаштування також включають оптимізації для продуктивності та стабільності, що робить його зручним для використання в сучасному кластері Kubernetes.

Kubernetes Dashboard – це веб-інтерфейс для управління кластерами Kubernetes, який спрощує візуалізацію, моніторинг і управління ресурсами кластера. Dashboard дозволяє працювати з ключовими ресурсами, такими як поди, сервіси, Deployment, ConfigMap та інші, через інтуїтивний графічний

інтерфейс. Інструмент особливо корисний для швидкого доступу до стану кластерів і підтримки їхньої роботи в реальному часі.

На рисунку 3.10 видно процес масштабування Deployment для ресурсу backend. У спеціальному вікні можна змінити кількість реплік подів, які будуть запуснені в кластері. Це відповідає команді `kubectl scale -n default deployment backend --replicas=1`, що задає одну репліку для цього Deployment. Праворуч показано список активних Deployment із зазначенням їхнього статусу (зелені індикатори), кількості подів і часу створення.

Масштабування через Dashboard дозволяє легко та швидко змінювати кількість реплік без використання командного рядка. Це зручно для моніторингу, аналізу стану подів і миттєвого реагування на зміни у навантаженні. Використання Dashboard демонструє ефективність і зручність інструментів Kubernetes для управління сервісами та забезпечення їхньої масштабованості.

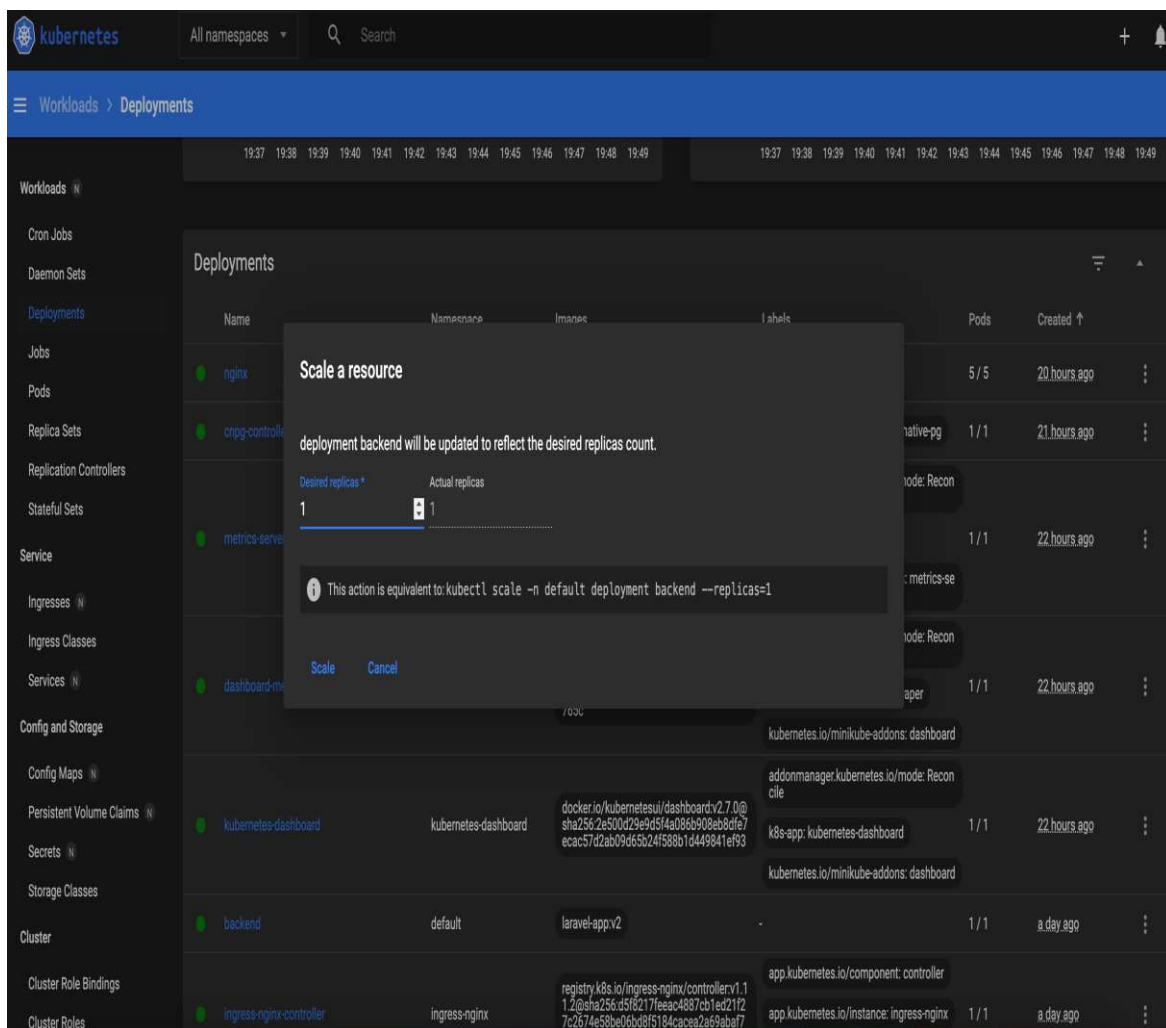


Рис 3.10 Процес масштабування сервісу backend

На рисунку 3.11 представлено Kubernetes Dashboard, який надає детальну інформацію про конкретний Pod з іменем backend-564b844fb9-5nzss. Це дозволяє відслідковувати стан та продуктивність Pod у реальному часі.

На панелі відображено використання ресурсів Pod, включаючи графіки споживання CPU та пам'яті. Використання CPU мінімальне, а пам'ять використовується стабільно без пікових навантажень. Метадані Pod показують його назву, простір імен (namespace) default, час роботи (20 годин) і мітки, такі як app: backend, які використовуються для зв'язування з іншими ресурсами, наприклад, сервісами.

Под працює на вузлі minikube зі статусом Running, що свідчить про успішну і стабільну роботу. Він має внутрішню IP-адресу 10.244.0.60, а клас якості обслуговування (QoS Class) встановлено як BestEffort, що означає мінімальні вимоги до ресурсів. Перезапусків не було, що свідчить про відсутність проблем із контейнером.

Цей рівень моніторингу допомагає відстежувати стабільність Pod, аналізувати продуктивність додатків і оптимізувати використання ресурсів. Dashboard значно спрощує управління Kubernetes-кластером, забезпечуючи надійність і ефективність роботи додатків.

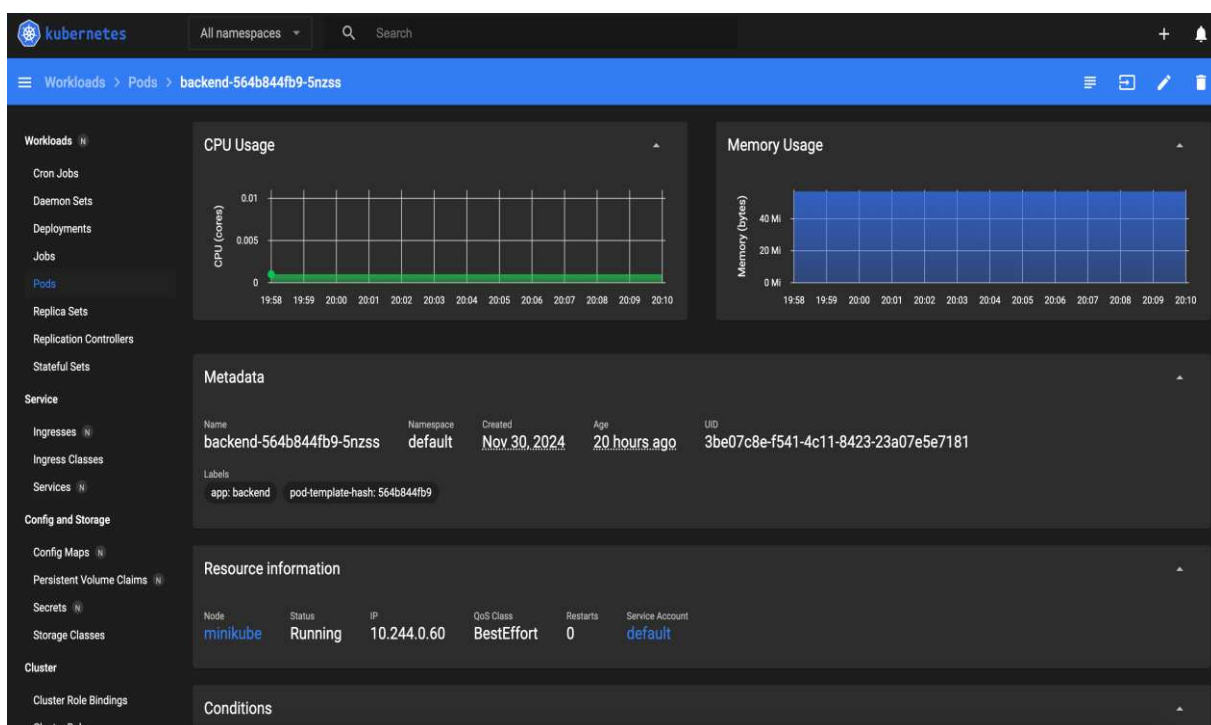


Рис 3.11 Моніторинг сервісу backend

На рисунку 3.12 показано Kubernetes Dashboard із деталями про ConfigMap під назвою backend-config. Цей ConfigMap використовується для зберігання конфігурацій сервісу у вигляді пар ключ-значення. Дані ConfigMap містять параметри для Laravel сервісу, такі як режим налагодження, середовище розробки, ключ шифрування, назва додатка, налаштування підключення до бази даних PostgreSQL, параметри логування, кешування, черг і JWT. ConfigMap дозволяє централізовано керувати конфігурацією сервісу, що спрощує внесення змін без потреби перезбирати образи Docker або змінювати код додатка. Використання ConfigMap у Kubernetes забезпечує гнучкість і спрощує управління сервісами. Dashboard надає зручний доступ до перегляду і редагування цих даних.

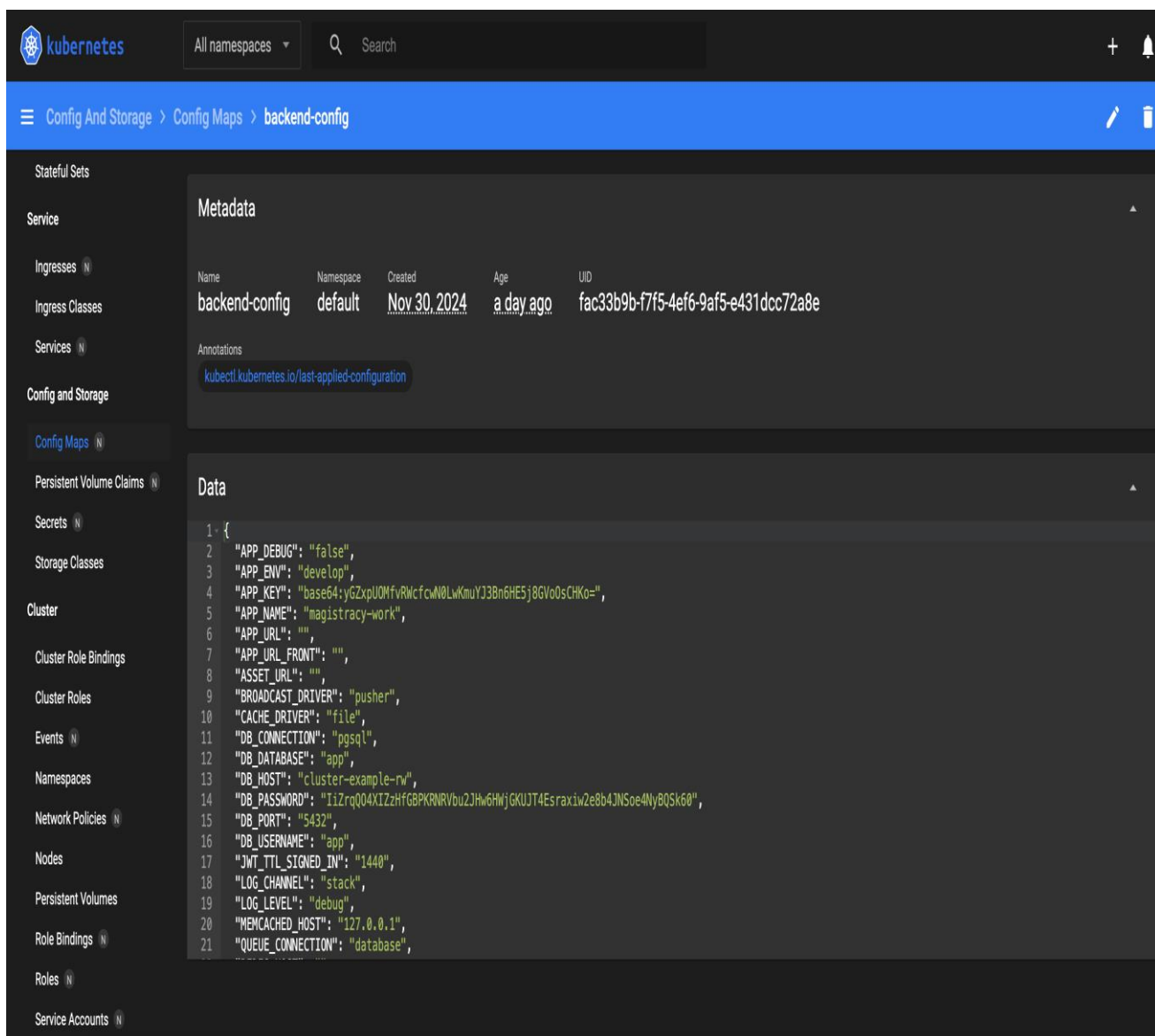


Рис 3.12 Configmap сервісу backend

Контейнеризація з використанням Docker забезпечує ізоляцію додатків, незалежність від середовища виконання і портативність. Kubernetes як оркестратор додає можливості автоматичного управління, масштабування та підтримки високої доступності. Інтеграція Kubernetes Dashboard значно спрощує управління кластером і моніторинг стану додатків у реальному часі.

Додаткова перевага Kubernetes полягає у впровадженні моделі оптимізації, яка забезпечує ефективний розподіл ресурсів, масштабування у відповідь на зміни навантаження, адаптацію інфраструктури до потреб додатків і зменшення експлуатаційних витрат. Ця модель сприяє створенню динамічних, надійних і масштабованих сервісів, що відповідають сучасним вимогам хмарних і мікросервісної архітектури. Оптимізація та автоматизація оркестрації є ключовими елементами у створенні стабільної та економічно ефективної інфраструктури для сервісів будь-якого масштабу.

3.4 Висновки до розділу 3

Впровадження сучасних технологій, таких як Docker, Kubernetes, разом із додатковими інструментами кешування, є ефективним підходом для створення високонавантажених сервісів. Контейнеризація за допомогою Docker дозволяє ізолювати середовище виконання кожного сервісу, забезпечуючи їхню незалежність і відтворюваність, що спрощує як розгортання, так і масштабування сервісів. Kubernetes, у свою чергу, надає потужні інструменти для оркестрації контейнерів, забезпечуючи автоматичне балансування навантаження, моніторинг стану сервісів, а також швидке відновлення у разі збоїв.

Використання мікросервісної архітектури дозволяє розділити систему на незалежні модулі, кожен із яких виконує конкретну бізнес-логіку. Такий підхід спрощує масштабування окремих компонентів відповідно до реальних потреб, покращує розподіл обчислювальних ресурсів і полегшує обслуговування

сервісів. Крім того, мікросервіси дозволяють командам працювати паралельно над різними частинами проекту, що значно прискорює процес розробки.

Інтеграція Redis для кешування додає значну перевагу у прискоренні роботи сервісів. Завдяки кешуванню даних у пам'яті, зменшується кількість звернень до бази даних, що не лише покращує час відгуку, але й дозволяє базі даних обробляти більше критично важливих запитів. Цей підхід також дозволяє знизити витрати на апаратне забезпечення, адже обробка даних у пам'яті є значно швидшою та менш ресурсоємною.

Таким чином, комплексне використання таких технологій, як Docker, Kubernetes, Redis, у поєднанні з мікросервісною архітектурою, створює основу для високопродуктивних, масштабованих та відмовостійких сервісів. Ці рішення дозволяють організаціям ефективно обробляти великі обсяги трафіку, швидко адаптуватися до змінних умов і гарантувати стабільну роботу навіть у найскладніших сценаріях.

ВИСНОВКИ

Підсумовуючи результати магістерської роботи, можна зробити висновок, що розроблені моделі для покращення ефективності високонавантажених систем, відкривають нові можливості для створення продуктивних, стабільних та масштабованих сервісів. У роботі детально розглянуто основні підходи до проектування, зокрема монолітну, сервісно-орієнтовану, мікросервісну, безсерверну, багатошарову та подійно-орієнтовану архітектуру, їхні плюси та мінуси, а також проблеми, які виникають при їх розробці та експлуатації.

Особлива увага приділяється викликам, таким як зменшення ефективності, надмірна завантаженість, нестабільність роботи та складність управління. Для вирішення цих проблем описано сучасні моделі та інструменти, серед яких: мікросервісна архітектура, кешування за допомогою Redis, контейнеризація з використанням Docker та оркестрація контейнерів Kubernetes. Ці технології дозволяють ізолювати середовище виконання сервісів, забезпечуючи їхню незалежність, автоматичне балансування навантаження, моніторинг стану сервісів та швидке відновлення у разі збоїв.

У підсумку, дослідження підтверджує значущість мікросервісної архітектури, сучасних технологій оркестрації, а також підходів до оптимізації систем для розробки високонавантажених сервісів. Розроблені моделі оптимізації, сприяють створенню гнучких, продуктивних і стійких до збоїв системи, здатних адаптуватися до зростання обсягів даних і навантаження.

Отримані результати надають корисні рекомендації для розробників і архітекторів, сприяючи створенню надійних та продуктивних інформаційних систем.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Introduction to High Load: What Is It? *Geniusee*. URL: <https://geniusee.com/single-blog/introduction-to-high-load-what-is-it> (Last accessed: 08.11.2024).
2. Woods E. Software Architecture in a Changing World. *IEEE Software*. 2016. Vol. 33, No. 6. P. 94–97.
3. El Murabet A., Abtoy A. Methodologies of the Validation of Software Architectures. *Journal of Computing Theories and Applications*. 2023. Vol. 1, No. 2. P. 29–36.
4. Гольдінер Д. І. Розробка архітектури програмного забезпечення для моделювання систем масового обслуговування під імплементацію мовою програмування Go. *Вісник Національного технічного університету «ХПІ»*. 2024. No 1 (11). С. 85–90.
5. Albada O. Architecture in Practice: Challenges and Opportunities in the Age of Digital Transformation. *Department of Software Engineering, Palestine Technical University-Kadoorie (PTUK), Tulkarm, Palestine*. 2023. November. P. 1–1
6. Скляренко О.В., Савченко Я.О., Литвиненко Л.О., Сушинський О.Є. Архітектурні підходи до розробки масштабованих веб-застосунків. *Cybersecurity Education Science Technique*. 2024. Vol. 4, No. 24. P. 341–350.
7. 10 Common Web Application Performance Problems and How to Solve Them. *Stackify*. URL: <https://stackify.com/web-application-problems/> (Last accessed: 10.11.2024).
8. Software Architecture Resources. *IEEE Computer Society*. URL: <https://www.computer.org/resources/software-architecture> (Last accessed: 15.11.2024).
9. What is Monolithic Architecture? *Integrate.io*. URL: <https://www.integrate.io/glossary/what-is-monolithic-architecture/> (Last accessed: 16.11.2024).

10. Event-Driven Architecture. AWS. URL: <https://aws.amazon.com/ru/event-driven-architecture/> (Last accessed: 17.11.2024).
11. Microservices Architecture. *Middleware.io*. URL: <https://middleware.io/blog/microservices-architecture/> (Last accessed: 18.11.2024).
12. Serverless Architecture. *Datadog*. URL: <https://www.datadoghq.com/knowledge-center/serverless-architecture/> (Last accessed: 19.11.2024).
13. Erl T. Service-Oriented Architecture: Analysis and Design for Services and Microservices. Boston: Prentice Hall, 2016. 864 p.
14. N-Tier Architecture. *Stackify*. URL: <https://stackify.com/n-tier-architecture/> (Last accessed: 13.11.2024).
15. Optimization Model. *IBM*. URL: <https://www.ibm.com/topics/optimization-model> (Last accessed: 20.11.2024).
16. Gorton I. Foundations of Scalable Systems. New York: O'Reilly Media, 2022. 352 p.
17. Попередній перегляд матеріалів лекції. *Studfile.net*. URL: <https://studfile.net/preview/7352154/> (дата звернення: 23.11.2024).
18. Abbot M. L., Fisher M. T. The Art of Scalability: Scalable Web Architecture, Processes, and Organizations for the Modern Enterprise. Addison-Wesley, 2015. 624 p.
19. Saaty T. L. The Analytic Hierarchy Process: Planning, Priority Setting, Resource Allocation. New York: McGraw-Hill, 1980. 287 p.
20. Чопенко М.С., Гайда А.Ю. Підвищення швидкості виконання SQL-запитів шляхом використання партиціювань на рівні СКБД. *Матеріали V Всеукраїнської науково-практичної інтернет-конференції «Інформаційні технології: моделі, алгоритми, системи»* (м. Миколаїв, 2024 р.). Національний університет кораблебудування імені адмірала Макарова. 2024. С. 168–173.

ДОДАТОК А КОД ПРОГРАМИ

Нижче наведені файли конфігурації Kubernetes:

backend-deployment.yaml

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: backend
spec:
  selector:
    matchLabels:
      app: backend
  replicas: 1
  minReadySeconds: 15
  strategy:
    type: RollingUpdate
    rollingUpdate:
      maxUnavailable: 1
      maxSurge: 1
  template:
    metadata:
      labels:
        app: backend
    spec:
      terminationGracePeriodSeconds: 60
      volumes:
        - name: nginx-config-volume
          configMap:
            name: backend-config-nginx
        - name: backend-supervisor
          configMap:
            name: backend-config-supervisor
        - name: shared-data
          emptyDir: {}
      containers:
        - image: laravel-app:v2
          imagePullPolicy: Never # this is addition
          name: backend-fpm-test
          ports:
            - containerPort: 9000
          volumeMounts:
            - name: nginx-config-volume
              mountPath: /etc/nginx/nginx.conf
              subPath: nginx.conf
```

- name: shared-data
 - mountPath: /var/www/app/storage/logs
 - name: backend-supervisor
 - mountPath: /etc/supervisor.d/backend-worker.conf
 - subPath: backend-worker.conf
- envFrom:
- configMapRef:
 - name: backend-config

backend-service.yaml

```

apiVersion: v1
kind: Service
metadata:
  name: backend-service
spec:
  ports:
    - targetPort: 80
      port: 80
  selector:
    app: backend

```

nginx-deployment.yaml

```

apiVersion: apps/v1
kind: Deployment
metadata:
  name: nginx
spec:
  selector:
    matchLabels:
      app: nginx
  replicas: 5
  minReadySeconds: 15
  strategy:
    type: RollingUpdate
    rollingUpdate:
      maxUnavailable: 1
      maxSurge: 1
  template:
    metadata:
      labels:
        app: nginx
    spec:
      containers:
        - image: nginxdemos/hello

```



```

name: nginx-demo
ports:
  - containerPort: 9000

```

nginx-service.yaml

```

apiVersion: v1
kind: Service
metadata:
  name: nginx-service
spec:
  ports:
    - targetPort: 80
      port: 80
  selector:
    app: nginx

```

backend-config.yaml

```

apiVersion: v1
kind: ConfigMap
metadata:
  name: backend-config
data:
  APP_NAME: "magistracy-work"
  APP_ENV: "develop"
  APP_DEBUG: "false"
  APP_URL: ""
  APP_URL_FRONT: ""
  APP_KEY: "base64:yGZxpUOMfvRWcfwN0LwKmuYJ3Bn6HE5j8GV0OsCHKo="
  ASSET_URL: ""

```

LOG_CHANNEL: "stack"

LOG_LEVEL: "debug"

BROADCAST_DRIVER: "pusher"

CACHE_DRIVER: "file"

QUEUE_CONNECTION: "database"

SESSION_DRIVER: "file"

SESSION_LIFETIME: "120"

MEMCACHED_HOST: "127.0.0.1"

JWT_TTL_SIGNED_IN: "1440"

DB_CONNECTION: "pgsql"

DB_HOST: "cluster-example-rw"

DB_PORT: "5432"

DB_DATABASE: "app"

DB_USERNAME: "app"

DB_PASSWORD:

"liZrqQO4XIZzHfGBPKRNRVbu2JHw6HWjGKUJT4Esraxiw2e8b4JNSoe4NyBQSk60"

REDIS_HOST: ""

REDIS_PORT: ""

backend-config-nginx.yaml

kind: ConfigMap

apiVersion: v1

metadata:

name: backend-config-nginx

data:

nginx.conf: |

```
events {  
  
}  
  
http {  
    server {  
        listen 80 default_server;  
  
        client_max_body_size 0;  
  
        root /var/www/app/public;  
        index index.html index.htm index.php;  
  
        location / {  
            try_files $uri $uri/ /index.php?$query_string;  
        }  
  
        location ~ \.php$ {  
            fastcgi_pass 127.0.0.1:9000;  
            fastcgi_param SCRIPT_FILENAME $realpath_root$fastcgi_script_name;  
            include fastcgi_params;  
        }  
  
        location /storage/ {  
            add_header 'Access-Control-Allow-Origin' '*';  
        }  
    }  
}
```

backend-config-supervisor.yaml

apiVersion: v1

kind: ConfigMap

metadata:

name: backend-config-supervisor

data:

backend-worker.conf: |-

[program:php-fpm]

command=/usr/local/sbin/php-fpm -F

autostart=true

autorestart=unexpected

exitcodes=0

stdout_logfile=/dev/stdout

stdout_logfile_maxbytes=0

stderr_logfile=/dev/stderr

stderr_logfile_maxbytes=0

startretries=0

[program:nginx]

command=/usr/sbin/nginx -g "daemon off;"

autorestart=true

stdout_logfile=/dev/stdout

stdout_logfile_maxbytes=0

stderr_logfile=/dev/stderr

stderr_logfile_maxbytes=0

startretries=0

database.yaml

apiVersion: postgresql.cnpg.io/v1

kind: Cluster

metadata:

name: cluster-example

```
spec:
  instances: 1
```

```
storage:
  size: 1Gi
```

ingress-app.yaml

```
apiVersion: networking.k8s.io/v1
kind: Ingress
metadata:
  name: ingress-magistracy-service
  annotations:
    nginx.ingress.kubernetes.io/rewrite-target: /$1
    nginx.ingress.kubernetes.io/proxy-read-timeout: "3600"
    nginx.ingress.kubernetes.io/proxy-send-timeout: "3600"
    nginx.ingress.kubernetes.io/proxy-body-size: 15m
    nginx.ingress.kubernetes.io/from-to-www-redirect: "true"
spec:
  ingressClassName: nginx
  rules:
    - host: magistracy.example
      http:
        paths:
          - path: /?(.*)
            pathType: ImplementationSpecific
            backend:
              service:
                name: backend-service
                port:
                  number: 80
    - host: magistracy-balancing.example
```

```

http:
  paths:
    - path: /?(.*)
      pathType: ImplementationSpecific
  backend:
    service:
      name: nginx-service
    port:
      number: 80

```

VacancyController.php

```
<?php
```

```

namespace App\Http\Controllers\Api;

use App\Http\Controllers\Controller;
use App\Http\Requests\ResumeFeedbackRequest;
use App\Models\FeedbackVacancy;
use App\Models\Resume;
use App\Models\Vacancy;
use Illuminate\Support\Facades\Auth;
use Spatie\QueryBuilder\AllowedFilter;
use Spatie\QueryBuilder\QueryBuilder;

```

```

class VacancyController extends Controller
{
    public function __construct()
    {
        $this->middleware('auth:sanctum')->except(['index','get','locations']);
    }

```

```

public function index()
{
    // Кешуємо всі вакансії
    $vacancies = Cache::remember('all_vacancies', 3600, function () {
        return Vacancy::with(['category', 'company'])->get();
    });

    return response()->json($vacancies);
}

public function get($id)
{
    return Vacancy::where('id', $id)->with(['category', 'company'])->firstOrFail();
}

public function locations()
{
    // Кешуємо унікальні локації вакансій
    $locations = Cache::remember('vacancy_locations', 3600, function () {
        return Vacancy::select('location')
            ->orderBy('location', 'asc')
            ->distinct()
            ->get()
            ->map(function ($value) {
                return $value->location;
            })->push("Yci");
    });

    return response()->json($locations);
}

public function feedback(ResumeFeedbackRequest $request)
{
    $feedback = $request->validated();
}

```

```

Resume::where([
  'id' => $feedback['resume_id'],
  'user_id' => Auth::user()->id
])->firstOrFail();

$feedbackVacancy = FeedbackVacancy::where([
  'resume_id' => $feedback['resume_id'],
  'vacancy_id' => $feedback['vacancy_id']
])->first();

if(!$feedbackVacancy) {
  FeedbackVacancy::create($feedback);
}

return FeedbackVacancy::where('id', $feedbackVacancy->id)->first();
}

public function myFeedbacks()
{
  $resume = Auth::user()->resume ? Auth::user()->resume->id: null;

  if($resume) {
    return FeedbackVacancy::where('resume_id', $resume)->with('vacancy','vacancy.company')-
>get();
  }

  return [];
}
}

```