Міністерство освіти і науки України

Харківський національний університет радіоелектроніки

Факультет інформаційно-аналітичних технологій та менеджменту

(повна назва)

Кафедра прикладної математики

(повна назва)

КВАЛІФІКАЦІЙНА РОБОТА
Пояснювальна записка

рівень вищої освіти другий (магістерський)

Аналіз підходів у розробці масштабованих веб-застосунків

(тема)

Виконав:

здобувач _2_ року навчання, групи _САУм-24-1

Артем ХАЛТУРІН

(Власне ім’я, ПРІЗВИЩЕ)

Спеціальність

124 Системний аналіз

(код і повна назва спеціальності)

Тип програми освітньо-професійна

(освітньо-професійна або освітньо-наукова)

Освітня програма

Системний аналіз і управління

(повна назва освітньої програми)

Керівник асист. Володимир ЛУХАНІН

(посада, Власне ім’я, ПРІЗВИЩЕ)

Допускається до захисту

Завідувач кафедри ПМ ____________ Максим СИДОРОВ

(підпис) (Власне ім’я, ПРІЗВИЩЕ)

2025 р.

Харківський національний університет радіоелектроніки

Факультет інформаційно-аналітичних технологій та менеджменту
Кафедра прикладної математики
Рівень вищої освіти другий (магістерський)
Спеціальність 124 Системний аналіз
(код і повна назва)
Тип програми освітньо-професійна
(освітньо-професійна або освітньо-наукова)
Освітня програма Системний аналіз і управління
(повна назва)

ЗАТВЕРДЖУЮ:

Завідувач кафедри ____________
(підпис)
“ 10 ” листопада 2025 р.

ЗАВДАННЯ
НА КВАЛІФІКАЦІЙНУ РОБОТУ

здобувачеві Халтуріну Артему Дмитровичу
(прізвище, ім’я, по батькові)

1. Тема роботи Аналіз підходів у розробці масштабованих веб-застосунків

затверджена наказом по університету від 10 листопада 2025 р. № 1027 Ст

2. Термін подання здобувачем роботи до екзаменаційної комісії 18 грудня 2025 р.

3. Вихідні дані до роботи інфраструктура мікросервісної архітектури з використанням Docker

4. Перелік питань, що потрібно опрацювати в роботі
1. Системний аналіз предметної області
2. Вибір і обґрунтування методу розв’язання
3. Програмна реалізація
4. Результати обчислювального експерименту
5. Аналіз можливих застосувань

5. Перелік графічного матеріалу із зазначенням креслеників, схем, плакатів, комп'ютерних ілюстрацій

1. Актуальність теми роботи
2. Постановка задачі
3. Системний аналіз предметної області
4. Метод чисельного аналізу
5. Результати обчислювального експерименту

КАЛЕНДАРНИЙ ПЛАН

№	Назва етапів роботи	Терміни виконання етапів роботи	Примітка
1	Підбір та вивчення технічної літератури за темою роботи	10 – 16 листопада 2025 р.	виконано
2	Вибір та обґрунтування методу	17 – 23 листопада 2025 р.	виконано
3	Розробка алгоритму і програми	24 – 30 листопада 2025 р.	виконано
4	Проведення аналітичних досліджень та розрахунків	01 – 07 грудня 2025 р.	виконано
5	Робота над текстом пояснювальної записки	08 – 17 грудня 2025 р.	виконано
6	Представлення роботи на рецензію в ЕК	18 грудня 2025 р.	виконано

Дата видачі завдання _10_ листопада 2025 р.

Здобувач ____________________________
(підпис)

Керівник роботи __________________ асист. Володимир ЛУХАНІН
(підпис) (посада, Власне ім'я, ПРІЗВИЩЕ)

РЕФЕРАТ

Пояснювальна записка: 76 с., 7 табл., 8 рис., 3 дод., 30 джерел.

АРХІТЕКТУРА ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ, ВЕБ-ЗАСТОСУНОК, МОНОЛІТНА АРХІТЕКТУРА, МІКРОСЕРВІСНА АРХІТЕКТУРА, SERVERLESS, МОВА ПРОГРАМУВАННЯ TYPESCRIPT, NODE.JS, ФРЕЙМВОРК NESTJS, ПЛАТФОРМА DOCKER, ПРОТОКОЛ HTTP.

Об'єкт дослідження – веб-застосунки, що функціонують у високонавантажених або розподілених середовищах і потребують масштабування та високої надійності.

Мета роботи – метою є проведення системного аналізу сучасних архітектурних підходів до розробки масштабованих веб-застосунків, порівняльна оцінка їх ефективності та формування практичних рекомендацій щодо вибору оптимальних рішень для забезпечення масштабованості веб-систем.

Методи дослідження – методи системного аналізу, архітектурні підходи (моноліт, мікросервіси, serverless), засоби управління масштабуванням, балансування навантаження, оптимізації продуктивності та прийняття технічних рішень при проєктуванні веб-застосунків.

У кваліфікаційній роботі проаналізовано архітектурні патерни (моноліт, мікросервіси, serverless) та обґрунтовано ефективність мікросервісного підходу для систем зі змінним навантаженням. Розроблено програмний прототип на базі Node.js (NestJS), Docker і Redis із реалізацією горизонтального масштабування та асинхронної обробки подій. Експериментально підтверджено зростання пропускної здатності (RPS) у 2,8 рази та зниження часу відгуку (P95) з 850 до 320 мс при масштабуванні критичного сервісу до трьох екземплярів. Отримані результати рекомендовані для впровадження у сферах E-commerce та FinTech. Подальший розвиток вбачається у використанні Kubernetes та Service Mesh.

ABSTRACT

Introductory note: 76 pages, 8 tables, 7 figures, 3 appendixes, 30 sources.

SOFTWARE ARCHITECTURE, WEB APPLICATION, MONOLITHIC ARCHITECTURE, MICROSERVICE ARCHITECTURE, SERVERLESS, TYPESCRIPT PROGRAMMING LANGUAGE, NODE.JS, NESTJS FRAMEWORK, DOCKER PLATFORM, HTTP PROTOCOL.

Object of research – web applications that operate in high-load or distributed environments and require scalability and high reliability.

Purpose of work – to conduct a systematic analysis of modern architectural approaches to developing scalable web applications, compare their effectiveness, and propose practical recommendations for choosing optimal solutions to ensure the scalability of web systems.

Methods of research – system analysis methods, architectural approaches (monolith, microservices, serverless), scaling management tools, load balancing, performance optimization, and technical decision-making in web application design.

The thesis analyzes architectural patterns (monolith, microservices, serverless) and substantiates the effectiveness of the microservice approach for systems with variable loads. A software prototype based on Node.js (NestJS), Docker, and Redis was developed with the implementation of horizontal scaling and asynchronous event processing. Experimental confirmation of a 2.8-fold increase in throughput (RPS) and a reduction in response time (P95) from 850 to 320 ms when scaling a critical service to three instances. The results obtained are recommended for implementation in the areas of E-commerce and FinTech. Further development is envisaged in the use of Kubernetes orchestration and Service Mesh.

ЗМІСТ

С.

Перелік скорочень, умовних познак, одиниць і термінів 8

Вступ 10

1 Системний аналіз предметної області та постановка задач дослідження 12

1.1 Системний аналіз задачі розробки масштабованих веб-застосунків 12

1.1.1 Вербальна модель системи 12

1.1.2 Морфологічний опис системи 14

1.1.3 Функціональна модель системи 17

1.1.4 Інформаційна модель 20

1.2 Аналіз сценаріїв вирішення задачі 21

1.2.1 Постановка задачі вибору методу 21

1.2.2 Оцінювання вектора пріоритетів незадоволеностей методом аналізу ієрархій 23

1.2.3 Модель вирішення проблеми 26

1.3 Змістовна та формальна постановка задачі 28

1.3.1 Змістовна постановка задачі 28

1.3.2 Формальна постановка задачі 29

1.4 Постановка задач дослідження 30

2 Вибір та обґрунтування методу розв’язання 31

2.1 Концепція масштабованості веб-застосунків 31

2.1.1 Горизонтальна vs вертикальна масштабованість 32

2.1.2 Метрики оцінювання масштабованості 34

2.2 Архітектурні патерни для масштабованих систем 36

2.2.1 Монолітна архітектура: переваги та обмеження 36

2.2.2 Мікросервісна архітектура 39

2.2.3 Serverless підхід Serverless підхід та event-driven архітектура 42

2.2.4 Порівняльна характеристика підходів 49

Висновки за розділом 2 54

3 Програмна реалізація 56
3.1 Інструментарій для розробки масштабованихрозподілених систем 56
3.2 Алгоритм розв’язання задачі створення прототипу веб-застосунку 58
3.3 Опис програми 59
Висновки за розділом 3 61
4 Результати обчислювального експерименту та їх аналіз 63
4.1 Підготовка даних та методологія обчислювального експерименту 63
4.2 Результати експерименту 64
Висновки за розділом 4 67
Висновки 68
Перелік джерел посилання 69
Додаток А Лістинг програми 72
Додаток Б Результат тестів для базового сценарію 75
Додаток В Результат тестів для масштабованого сценарію 76

ПЕРЕЛІК СКОРОЧЕНЬ, УМОВНИХ ПОЗНАК, ОДИНИЦЬ І ТЕРМІНІВ

ACID (Atomicity, Consistency, Isolation, Durability) – набір властивостей транзакцій (атомарність, узгодженість, ізольованість, довговічність), що забезпечують надійність баз даних;

Amazon Kinesis – керований сервіс AWS для збору, обробки та аналізу потокових даних у реальному часі;

Apache Kafka – розподілена платформа для потокової передачі подій та створення конвеєрів даних;

API (Application Programming Interface) – інтерфейс прикладного програмування, набір правил для взаємодії між комп’ютерними програмами;

AWS (Amazon Web Services) – хмарна платформа від компанії Amazon, що надає широкий спектр інфраструктурних сервісів;

AWS Lambda – безсерверний (serverless) обчислювальний сервіс від AWS, що виконує код у відповідь на події;

Azure – хмарна платформа від компанії Microsoft;

Bottleneck (вузьке місце) – компонент системи з найменшою пропускною здатністю, який обмежує загальну продуктивність;

CDN (Content Delivery Network) – географічно розподілена мережа серверів для швидкої доставки контенту користувачам;

CPU (Central Processing Unit) – центральний процесор, основний обчислювальний елемент сервера;

CQRS (Command Query Responsibility Segregation) – архітектурний патерн, що розділяє моделі читання та запису даних;

CRUD (Create, Read, Update, Delete) – базові операції управління даними (створення, читання, оновлення, видалення);

DNS (Domain Name System) – система доменних імен для перетворення читабельних адрес сайтів у IP-адреси;

gRPC (Google Remote Procedure Call) – високопродуктивний фреймворк для віддаленого виклику процедур;

IoT (Internet of Things) – Інтернет речей, мережа фізичних об'єктів, підключених до Інтернету для обміну даними;

Kubernetes – платформа з відкритим кодом для автоматизації розгортання, масштабування та управління контейнеризованими застосунками;

MVP (Minimum Viable Product) – мінімально життєздатний продукт, версія застосунку з достатнім функціоналом для перших користувачів;

P95 (95th Percentile) – метрика затримки, яка показує, що 95% запитів обробляються швидше за вказане значення;

QoS (Quality of Service) – якість обслуговування, сукупність характеристик, що визначають ступінь задоволення користувача сервісом;

REST (Representational State Transfer) – архітектурний стиль взаємодії компонентів розподіленого застосунку в мережі;

RPS (Requests Per Second) – кількість запитів, які система обробляє за одну секунду;

MVC (Model-View-Controller) – архітектурний шаблон проектування, що розділяє застосунок на три взаємопов'язані частини: модель (дані), вигляд (інтерфейс користувача) та контролер (логіка керування);

TPS (Transactions Per Second) – кількість транзакцій, які система може виконати за одну секунду; TTFB (Time To First Byte) – час до отримання першого байта відповіді від сервера.

ВСТУП

Актуальність теми. Сучасний етап розвитку цифрової економіки характеризується експоненційним зростанням обсягів даних та кількості користувачів онлайн-сервісів. Провідні технологічні компанії та наукові установи (Amazon, Netflix, Google) активно досліджують проблеми забезпечення стабільності веб-систем в умовах пікових навантажень. Світові тенденції свідчать про поступовий перехід від традиційних монолітних архітектур до розподілених систем, зокрема мікросервісних та безсерверних (serverless), що дозволяє досягти вищої гнучкості та відмовостійкості.

Актуальність роботи зумовлена необхідністю вирішення протиріччя між зростаючими вимогами до продуктивності веб-застосунків та обмеженнями класичних підходів до їх проєктування. Вибір оптимального архітектурного рішення перестає бути суто технічною задачею і потребує комплексного системного аналізу для знаходження балансу між масштабованістю, складністю підтримки та вартістю інфраструктури.

Мета і завдання кваліфікаційної роботи. Метою кваліфікаційної роботи є проведення системного аналізу сучасних архітектурних підходів до розробки масштабованих веб-застосунків, порівняльна оцінка їх ефективності та формування практичних рекомендацій щодо вибору оптимальних рішень для забезпечення масштабованості веб-систем. Для досягнення поставленої мети необхідно виконати наступні завдання:

– провести огляд існуючих проблем масштабування веб-застосунків та визначити основні вимоги до архітектур масштабованих систем;

– проаналізувати сучасні архітектурні підходи, що використовуються для забезпечення масштабованості веб-застосунків;

– провести порівняльний аналіз переваг та недоліків виявлених підходів за визначеними критеріями;

– розробити спрощений прототип веб-застосунку з використанням обраного підходу;

– оцінити продуктивність та масштабованість реалізованого рішення при різному навантаженні або змінних параметрах;

– сформулювати рекомендації для побудови масштабованих веб-застосунків.

Об'єктом дослідження є веб-застосунки, що функціонують у високонавантажених або розподілених середовищах і потребують масштабування та високої надійності.

Предметом дослідження є методи системного аналізу, архітектурні підходи (моноліт, мікросервіси, serverless), засоби управління масштабуванням, балансування навантаження, оптимізації продуктивності та прийняття технічних рішень при проєктуванні веб-застосунків.

Методи дослідження. У роботі використовуються методи системного аналізу для дослідження структури та взаємозв'язків компонентів веб-застосунків, порівняльний аналіз для оцінки різних архітектурних підходів, методи моделювання, а також методи експериментального тестування продуктивності для оцінки масштабованості розробленого прототипу при різних рівнях навантаження.

Публікації. Результати, отримані у кваліфікаційній роботі, було представлено на науково-технічній конференції «Комп'ютерно-інтегровані технології автоматизації технологічних процесів на транспорті та у виробництві» (м. Харків, 7-8 листопада 2025 р.) [1].

1 СИСТЕМНИЙ АНАЛІЗ ПРЕДМЕТНОЇ ОБЛАСТІ ТА ПОСТАНОВКА ЗАДАЧ ДОСЛІДЖЕННЯ

1.1 Системний аналіз задачі розробки масштабованих веб-застосунків

1.1.1 Вербальна модель системи

Об'єктом аналізу є масштабований веб-застосунок, призначений для роботи в високонавантажених або розподілених середовищах, що потребує забезпечення високої продуктивності, надійності та можливості динамічного масштабування. Основна задача системи полягає у забезпеченні безперебійного обслуговування зростаючої кількості користувачів та обробки великих обсягів запитів при збереженні прийнятної швидкості відгуку та стабільності роботи.

Призначення системи: забезпечення ефективної обробки користувацьких запитів, управління навантаженням, горизонтального та вертикального масштабування ресурсів, а також підтримка високої доступності сервісу для вирішення бізнес-задач організацій, що потребують обробки значних потоків даних та взаємодії з великою кількістю користувачів одночасно.

Мета системи: створення архітектурного рішення, яке забезпечує оптимальний баланс між продуктивністю, масштабованістю, складністю розгортання та витратами на підтримку інфраструктури. Система має гарантувати можливість адаптації до змінних умов навантаження, підтримувати відмовостійкість компонентів та забезпечувати ефективне використання обчислювальних ресурсів через застосування сучасних архітектурних підходів, таких як мікросервісна архітектура, контейнеризація, оркестрація та хмарні технології.

Проведемо класифікацію системи.

За походженням система є штучною, оскільки створена людиною для обробки користувацьких запитів та надання веб-сервісів.

За об'єктивністю існування система є абстрактною, адже представлена як концептуальна модель та програмне забезпечення, розгорнуте на фізичній або

хмарній інфраструктурі.

За природою система належить до соціотехнічних, оскільки об'єднує технічні компоненти, такі як сервери, бази даних, балансувальники навантаження, із людським фактором – користувачами, розробниками та адміністраторами.

За централізованістю система може бути як централізованою (монолітна архітектура), так і децентралізованою (мікросервісна архітектура), залежно від обраного архітектурного підходу.

За розмірністю система є багатовимірною, оскільки має численні входи (запити користувачів), обробляє їх паралельно через різні компоненти та видає відповідні результати.

За однорідністю структурних елементів система є гетерогенною, оскільки складається з різних компонентів, таких як веб-сервери, сервери додатків, бази даних, системи кешування, черги повідомлень, кожен з яких виконує унікальну функцію.

За лінійністю система є нелінійною, оскільки її поведінка залежить від складних взаємодій між компонентами, навантаження на систему та зовнішніх факторів.

За цільовою орієнтацією система є цілеспрямованою, оскільки її мета визначається внутрішніми функціями для забезпечення масштабованості, продуктивності та надійності веб-сервісу.

За складністю система є складною, оскільки характеризується взаємозалежністю компонентів і складними алгоритмами взаємодії, такими як балансування навантаження, розподіл запитів, синхронізація даних та управління станом.

За ступенем детермінованості система є недетермінованою (стохастичною), оскільки поведінка системи залежить від непередбачуваних факторів, таких як інтенсивність користувацьких запитів, мережеві затримки та відмови окремих компонентів.

За взаємодією із зовнішнім середовищем система є відкритою, адже активно обмінюється даними з користувачами, зовнішніми API, базами даних та

іншими сервісами.

За способом організації система є ієрархічною, оскільки структурована як багаторівнева модель, де презентаційний рівень, бізнес-логіка, рівень даних та інфраструктурні компоненти підпорядковуються централізованому або розподіленому управлінню.

За способом управління система має комбіноване управління, оскільки частина функцій, таких як автомасштабування та балансування навантаження, автоматизована, а частина, наприклад налаштування архітектури та моніторинг, вимагає втручання людини.

За статичністю система є динамічною, адже її стан змінюється в реальному часі залежно від навантаження, доступності ресурсів і зовнішніх умов.

1.1.2 Морфологічний опис системи

Морфологічний опис системи «масштабований веб-застосунок» відображає її структуру, складові частини, взаємодію із зовнішнім середовищем та основні компоненти, необхідні для забезпечення функціонування. Система представляється у вигляді моделі типу «чорна скринька», де вхідні дані – це HTTP/HTTPS запити користувачів, які можуть включати різні типи операцій (читання, запис, оновлення даних), параметри запитів, файли та інші дані. На виході система генерує HTTP-відповіді з відповідним статусом, контентом (HTML, JSON, XML тощо) та метаданими. Внутрішній склад системи користувачем безпосередньо не досліджується, а увага зосереджується на її межах, які забезпечують цілісність, доступність і продуктивність. Ці межі включають точки входу (load balancer, API Gateway), які розподіляють навантаження між серверами; механізми автентифікації та авторизації, що контролюють доступ до ресурсів; а також використання кешування, CDN та оптимізації запитів для забезпечення швидкої обробки даних на всіх етапах (рис. 1.1) [3].

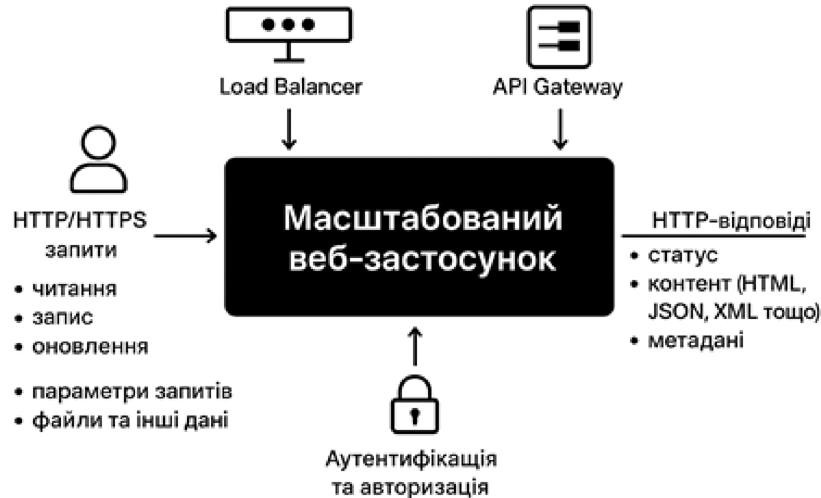

Рисунок 1.1 – Модель типу «чорна скринька»

Система «Масштабований веб-застосунок» складається з кількох ключових компонентів, які забезпечують її функціональність. Основними компонентами є:

– балансувальник навантаження (Load Balancer);

– веб-сервери (Web Servers);

– сервери додатків (Application Servers);

– база даних (Database Layer);

– система кешування (Caching Layer);

– система черг повідомлень (Message Queue);

– моніторинг та логування (Monitoring & Logging).

Розглянемо більш детально компоненти системи.

Ці компоненти є стандартними будівельними блоками для більшості сучасних розподілених систем, як-от описано у праці Мартіна Клеппмана [2].

Балансувальник навантаження розподіляє вхідні запити між доступними серверами для забезпечення рівномірного навантаження та високої доступності. Він визначає стан серверів і перенаправляє трафік лише на працездатні вузли, що підвищує відмовостійкість системи.

Веб-сервери обробляють HTTP-запити, надають статичний контент

(HTML, CSS, JavaScript, зображення) та передають динамічні запити на сервери додатків. Вони можуть бути масштабовані горизонтально для обробки більшої кількості одночасних запитів.

Сервери додатків виконують бізнес-логіку застосунку, обробляють дані користувачів, взаємодіють із базами даних та зовнішніми API. Залежно від архітектурного підходу, це може бути монолітний застосунок або набір мікросервісів.

База даних забезпечує зберігання та управління даними застосунку. Для масштабованості можуть використовуватися реплікація (read replicas), шардинг або NoSQL-рішення, які дозволяють горизонтально масштабувати дані.

Система кешування (наприклад, Redis, Memcached) зберігає часто використовувані дані в пам'яті для зменшення навантаження на базу даних та прискорення часу відгуку системи.

Система черг повідомлень (наприклад, RabbitMQ, Kafka) забезпечує асинхронну обробку задач, розвантажує основні компоненти та дозволяє масштабувати обробку повідомлень незалежно від інших частин системи.

Моніторинг та логування здійснюють збір метрик продуктивності, відстеження помилок та аналіз поведінки системи в реальному часі для своєчасного виявлення проблем і оптимізації роботи.

Межею системи «Масштабований веб-застосунок» є обмежене середовище, в якому забезпечується обробка запитів, зберігання даних і генерація відповідей відповідно до вимог продуктивності та надійності. Це середовище включає в себе всі компоненти системи, які взаємодіють між собою та з зовнішніми елементами.

До основних елементів зовнішнього середовища належать:

– користувачі – кінцеві споживачі веб-сервісу, які надсилають запити через веб-браузер або мобільні додатки;

– хмарні сервіси та інфраструктура – забезпечують обчислювальні ресурси, зберігання даних, мережеві можливості та додаткові сервіси (AWS, Azure, Google Cloud);

– розробники та DevOps-команда – відповідальні за розробку, розгортання, налаштування, моніторинг і оновлення системи, включаючи управління CI/CD-процесами;

– зовнішні API та сервіси – забезпечують інтеграцію з платіжними системами, соціальними мережами, аналітичними платформами тощо;

– постачальники технологій – надають програмне та апаратне забезпечення для підтримки системи, включаючи сервери, контейнерні платформи, системи оркестрації (Kubernetes, Docker) [4].

Обмін інформацією між системою та зовнішнім середовищем забезпечує ефективну роботу застосунку, підтримує продуктивність і відповідає вимогам масштабованості та доступності (рис. 1.2).

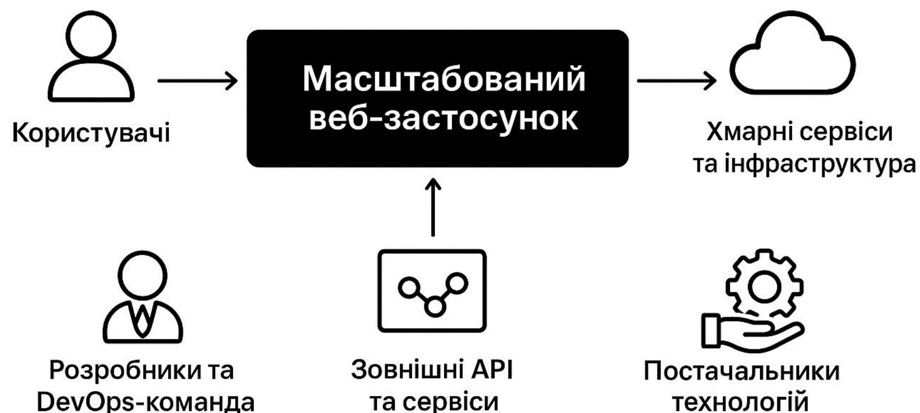

Рисунок 1.2 – Модель зовнішнього середовища системи

1.1.3 Функціональна модель системи

Метою функціональної моделі системи є аналіз процесів, які забезпечу-

ють масштабоване функціонування веб-застосунку в умовах змінного навантаження. Система спрямована на забезпечення продуктивності, відмовостійкості та ефективного використання ресурсів.

Функціональна модель системи включає такі процеси:

– прийом та маршрутизація запитів: вхідні HTTP/HTTPS запити надходять до балансувальника навантаження, який розподіляє їх між доступними веб-серверами згідно з обраним алгоритмом (round-robin, least connections, IP hash тощо);

– обробка статичного контенту: веб-сервери обслуговують статичні файли (HTML, CSS, JS, зображення) безпосередньо або через CDN для зменшення навантаження та прискорення доставки контенту;

– виконання бізнес-логіки: сервери додатків обробляють динамічні запити, виконують бізнес-правила, взаємодіють із базами даних та зовнішніми сервісами;

– управління даними: система забезпечує читання та запис даних до баз даних, використовуючи механізми реплікації, шардингу або кластеризації для підвищення продуктивності та надійності;

– кешування даних: часто використовувані дані зберігаються в системі кешування для зменшення навантаження на базу даних та прискорення відгуку системи;

– асинхронна обробка задач: важкі або тривалі операції (наприклад, обробка файлів, відправка email) виконуються асинхронно через черги повідомлень, що дозволяє не блокувати основний потік обробки запитів;

– моніторинг та автомасштабування: система постійно відстежує метрики продуктивності (CPU, пам'ять, час відгуку) та автоматично додає або видаляє ресурси залежно від поточного навантаження;

– логування та аналіз: всі запити, помилки та події логуються для подальшого аналізу, виявлення вузьких місць та оптимізації роботи системи.

Функціональна модель системи визначає всі ключові етапи обробки запитів, від їх надходження до генерації відповіді, та забезпечує виконання вимог

масштабованості, продуктивності та надійності. Це дозволяє створити веб-застосунок, здатний ефективно працювати під високим навантаженням.

До механізмів системи належать: балансувальник навантаження, який розподіляє запити між серверами; веб-сервери та сервери додатків, що обробляють запити; бази даних і системи кешування, які забезпечують швидкий доступ до даних; хмарна інфраструктура, яка підтримує масштабованість та доступність; системи моніторингу, які відстежують стан системи. Крім того, DevOps-команда забезпечує підтримку працездатності, налаштування та оптимізацію системи.

До управління системою належать правила, стандарти та політики, що забезпечують її функціонування. Основними управлінськими елементами є конфігурації автомасштабування, які визначають умови додавання або видалення ресурсів; політики безпеки, які регулюють доступ до системи; стандарти якості коду та архітектурні принципи, які забезпечують підтримуваність системи.

Виходом системи є HTTP-відповіді, які містять запитувану інформацію, відповідають вимогам продуктивності та генеруються з мінімальними затримками.

Система обробки запитів забезпечує автоматизовану роботу з даними користувача та генерує відповіді, дотримуючись принципів масштабованості та продуктивності. Вхідним елементом системи є HTTP-запит, який передається до балансувальника навантаження. На цьому етапі відбувається визначення оптимального сервера для обробки запиту та перенаправлення запиту до обраного веб-сервера. Результатом цього процесу є маршрутизований запит, готовий до обробки.

Далі запит надходить до модуля обробки бізнес-логіки, де виконується аналіз запиту, взаємодія з базою даних або системою кешування, виконання необхідних обчислень та формування відповіді. Цей процес відбувається з урахуванням поточного навантаження на систему та доступності ресурсів.

Згенерована відповідь після обробки повертається користувачу через веб-сервер (рис. 1.3).

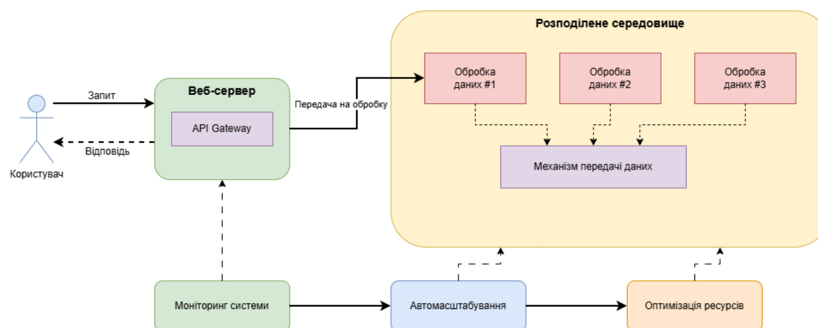

Рисунок 1.3 – Розподілена система обробки даних

Для забезпечення продуктивності та масштабованості всі процеси виконуються в розподіленому середовищі, яке гарантує надійну обробку та передачу даних. Усі дії в системі контролюються механізмами моніторингу, а також політиками автомасштабування, які забезпечують оптимальне використання ресурсів.

1.1.4 Інформаційна модель

Інформаційна модель системи «Масштабований веб-застосунок з розподіленою архітектурою» дозволяє описати структуру даних, їх взаємозв'язки та основні елементи, необхідні для забезпечення ефективної обробки запитів та масштабованості системи (рис. 1.4).

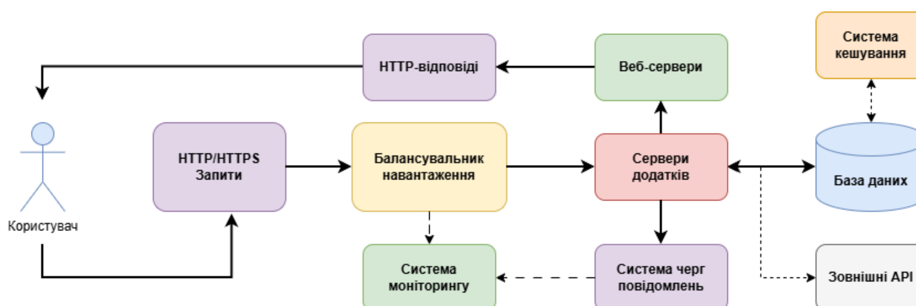

Рисунок 1.4 – Інформаційна модель масштабованого веб-застосунку

Основними компонентами інформаційної моделі є такі:

– запити користувачів, що включають HTTP/HTTPS запити з параметрами, заголовками, тілом запиту та метаданими, необхідними для обробки;

– балансувальник навантаження, який виконує аналіз доступності серверів, розподіл запитів та моніторинг стану вузлів;

– веб-сервери, що обробляють статичний контент та передають динамічні запити на сервери додатків;

– сервери додатків, які виконують бізнес-логіку, обробляють дані, взаємодіють із базами даних та зовнішніми API;

– база даних, яка зберігає структуровані дані застосунку, підтримує механізми транзакцій, реплікації та шардингу для забезпечення масштабованості;

– система кешування, що зберігає часто використовувані дані в пам'яті для прискорення доступу та зменшення навантаження на базу даних;

– система черг повідомлень, яка забезпечує асинхронну обробку задач, розвантажуючи основні компоненти системи;

– система моніторингу, що збирає метрики продуктивності, логи подій та дані про помилки для аналізу та оптимізації роботи системи;

– HTTP-відповіді, які є кінцевим результатом роботи системи, адаптованим до запиту користувача та готовим до передачі через мережу.

1.2 Аналіз сценаріїв вирішення задачі

1.2.1 Постановка задачі вибору методу

Наступним етапом системного аналізу задачі є вибір архітектурного підходу, який найбільш доцільно застосувати для створення масштабованого веб-застосунку. Для прийняття обґрунтованого рішення про вибір відповідного методу необхідно визначити перелік можливих підходів і порівняти їх за відповідними критеріями.

Для аналізу було обрано такі критерії оцінки:

– критерій 1 (К1): масштабованість – здатність системи ефективно обробляти зростаюче навантаження шляхом додавання ресурсів;

– критерій 2 (К2): складність розгортання та підтримки – трудомісткість впровадження, налаштування та супроводу архітектурного рішення;

– критерій 3 (К3): продуктивність – швидкість обробки запитів та ефективність використання ресурсів;

– критерій 4 (К4): витрати на інфраструктуру – вартість розгортання та експлуатації системи, включаючи серверні ресурси та хмарні сервіси.

Для реалізації поставленої задачі було визначено такі альтернативи:

– альтернатива 1 (А1): монолітна архітектура – традиційний підхід, де всі компоненти застосунку об'єднані в єдиний модуль;

– альтернатива 2 (А2): мікросервісна архітектура – розподілений підхід, де застосунок складається з незалежних сервісів, кожен з яких виконує окрему функцію;

– альтернатива 3 (А3): serverless архітектура – підхід на основі функцій як сервісу (FaaS), де код виконується у відповідь на події без необхідності управління серверами.

Основною метою ієрархії є забезпечення вирішення задачі, першим рівнем виступають критерії оцінки (масштабованість, складність розгортання та підтримки, продуктивність, витрати на інфраструктуру), а другим рівнем – альтернативи (монолітна архітектура, мікросервісна архітектура, serverless архітектура). Ця модель дозволяє порівняти альтернативи за ключовими характеристиками, обґрунтувати вибір найкращого методу та забезпечити ефективне впровадження системи. Використання даної ієрархії сприяє прозорості процесу прийняття рішень та полегшує комунікацію між технічними та бізнес-командами. Ієрархічна модель вибору методу для вирішення задачі побудови масштабованого веб-застосунку наведена на рис. 1.5.

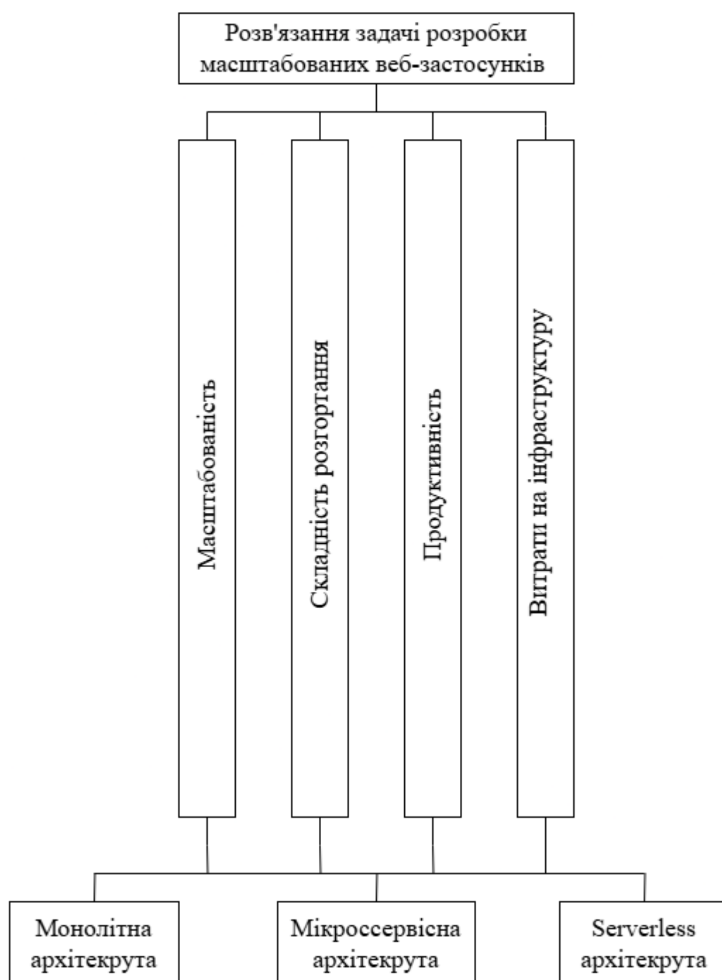

Рисунок 1.5 – Ієрархічна модель вибору методу розв'язання задачі розробки масштабованих веб-застосунків

1.2.2 Оцінювання вектора пріоритетів незадоволеностей методом аналізу ієрархій

Для аналізу ієрархії побудуємо матриці парних порівнянь моделі, а також критеріїв системи.

Матриця парних порівнянь критеріїв записана у таблиці 1.1. Останній стовпчик цієї таблиці містить результати розрахунків для вектора пріоритетів критеріїв.

Таблиця 1.1 – Матриця парних порівнянь критеріїв

Критерії оцінювання	К1	К2	К3	К4	Оцінки компонентів	Вектор пріоритетів
К1	1	7	5	3	3,20	0,55
К2	0,14	1	0,33	0,2	0,33	0,06
К3	0,2	3	1	0,5	0,76	0,13
К4	0,33	5	2	1	0,29	0,26
Усього					5,79	

Випадкова узгодженість для матриці четвертого порядку дорівнює 0,9.

За даними таблиці 1.1:

– індекс узгодженості ІУ = 0,045;

– відносна узгодженість ВУ = 0,05.

Оскільки відносна узгодженість близька до 0,1 (і менше за 0,1), то робимо висновок, що матриця парних порівнянь критеріїв побудована правильно.

Вектор локальних пріоритетів критеріїв відносно проблеми вибору дорівнює $W = (0,55;0,06;0,13;0,26)^T$.

Сформуємо матриці попарних порівнянь альтернатив за кожним критерієм (таблиці 1.2 – 1.5) та виконаємо розрахунки за ними.

Таблиця 1.2 – Матриця попарних порівнянь за першим критерієм (масштабованість)

К1	А1	А2	А3	Оцінки компонентів	Вектор пріоритетів
А1	1	0,2	0,33	0,41	0,11
А2	5	1	3	2,47	0,65
А3	3	0,33	1	1	0,24
Усього				3,88	

За даними таблиці 1.2:

– індекс узгодженості ІУ = 0,03;

– відносна узгодженість ВУ = 0,05.

Таблиця 1.3 – Матриця попарних порівнянь за другим критерієм (складність розгортання)

К1	А1	А2	А3	Оцінки компонентів	Вектор пріоритетів
А1	1	5	3	2,47	0,65
А2	0,2	1	0,33	0,41	0,11
А3	0,33	3	1	1	0,24
Усього				3,88	

За даними таблиці 1.3:

– індекс узгодженості ІУ = 0,03;

– відносна узгодженість ВУ = 0,05.

Таблиця 1.4 – Матриця попарних порівнянь за третім критерієм (продуктивність)

К1	А1	А2	А3	Оцінки компонентів	Вектор пріоритетів
А1	1	3	0,5	1,14	0,31
А2	0,3	1	0,2	0,41	0,11
А3	2	5	1	2,15	0,58
Усього				3,70	

За даними таблиці 1.4:

– індекс узгодженості ІУ = 0,04;

– відносна узгодженість ВУ = 0,07.

Таблиця 1.5 – Матриця попарних порівнянь за четвертим критерієм (витрати)

К1	А1	А2	А3	Оцінки компонентів	Вектор пріоритетів
А1	1	0,33	0,2	0,41	0,11
А2	3	1	0,5	1,14	0,31
А3	5	2	1	2,15	0,58
Усього				3,70	

За даними таблиці 1.5:

– індекс узгодженості ІУ = 0,04;

– відносна узгодженість ВУ = 0,07.

1.2.3 Модель вирішення проблеми

Проведений аналіз у п. 1.2.2 дозволив обрати найкращий архітектурний підхід для реалізації масштабованого веб-застосунку. Підсумкові результати оцінки альтернатив представлені в таблиці 1.6.

Таблиця 1.6 – Остаточні розрахунки

Альтернатива	Критерій				Узагальнені пріоритети
	К1(0,55)	К2 (0,06)	К3 (0,13)	К4 (0,26)	
А1	0,11	0,65	0,31	0,11	0,19
А2	0,65	0,11	0,11	0,31	0,47
А3	0,24	0,24	0,58	0,58	0,34

Узагальнені пріоритети розраховуються як:

$$A_1 : 0,11 \cdot 0,55 + 0,65 \cdot 0,06 + 0,11 \cdot 0,13 + 0,11 \cdot 0,26 = 0,19\,,$$

$$A_2 : 0,65 \cdot 0,55 + 0,11 \cdot 0,06 + 0,11 \cdot 0,13 + 0,31 \cdot 0,26 = 0,47\,,$$

$$A_3 : 0,24 \cdot 0,55 + 0,24 \cdot 0,06 + 0,58 \cdot 0,13 + 0,58 \cdot 0,26 = 0,34\,.$$

На основі отриманих даних оптимальним вибором для вирішення поставленої задачі є друга альтернатива – мікросервісна архітектура, яка отримала найвищий глобальний пріоритет (0,47).

Таке рішення є обґрунтованим, оскільки мікросервісна архітектура продемонструвала найбільшу відповідність ключовому критерію – масштабованості (К1), що є основною вимогою для створення ефективної системи, здатної обробляти зростаюче навантаження.

Мікросервісна архітектура забезпечує:

– високу масштабованість завдяки можливості незалежного масштабування окремих сервісів;

– гнучкість розробки та розгортання – різні команди можуть працювати над окремими мікросервісами незалежно;

– відмовостійкість – відмова одного сервісу не призводить до зупинки всієї системи;

– технологічну різноманітність – кожен сервіс може використовувати оптимальний стек технологій.

Хоча serverless архітектура (А3) також показала високі результати за критеріями продуктивності та витрат (0,34), а монолітна архітектура (А1) має перевагу у простоті розгортання (0,19), мікросервісний підхід найкраще відповідає всім вимогам і може бути рекомендований як оптимальний архітектурний стиль для побудови масштабованого веб-застосунку, який забезпечує надійність, продуктивність і можливість ефективного горизонтального масштабування.

1.3 Змістовна та формальна постановка задачі

1.3.1 Змістовна постановка задачі

Розробка масштабованих веб-застосунків є критично важливим завданням для організацій, що прагнуть забезпечити стабільну роботу своїх продуктів при зростаючому навантаженні та розширенні функціональності. Сучасні веб-застосунки повинні обслуговувати тисячі або мільйони користувачів одночасно, зберігаючи високу продуктивність, доступність та якість користувацького досвіду. Однак створення таких систем супроводжується низкою викликів, зокрема вибором оптимальної архітектури, технологічного стеку та підходів до організації коду.

Поставлена задача полягає в аналізі та систематизації сучасних підходів до розробки масштабованих веб-застосунків, що включає дослідження архітектурних патернів, фронтенд та бекенд технологій та методів оптимізації продуктивності. Така система має забезпечувати можливість горизонтального та вертикального масштабування, підтримувати high availability, мінімізувати технічний борг та дозволяти ефективну паралельну розробку множиною команд.

Розв'язання цієї задачі базується на комплексному вивченні існуючих архітектурних підходів – від монолітних систем до мікросервісів, serverless та event-driven архітектур [5]. Особлива увага приділяється фронтенд-архітектурам, зокрема Feature-Sliced Design, що забезпечує масштабованість та підтримуваність клієнтської частини застосунку [6]. Завдяки систематичному аналізу різних підходів можна сформувати рекомендації щодо оптимального вибору технологій та архітектурних рішень для різних типів веб-застосунків [7].

1.3.2 Формальна постановка задачі

Мета оптимізації – вибір архітектурного підходу A із множини можливих альтернатив $A \in \{A_{\text{mono}}, A_{\text{micro}}, A_{\text{serverless}}\}$, який максимізує загальну ефективність системи E при змінному навантаженні L.

Ефективність E визначається як функція продуктивності P та сукупної вартості володіння C:

$$E(A,L) = \frac{P(A,L)}{C(A,L)} \to \max,$$

де $P(A,L)$ – продуктивність – це функція, що агрегує ключові показники якості обслуговування, такі як пропускна здатність Th (кількість запитів на секунду) та час відгуку T_{res} (латентність);

$C(A,L)$ – вартість – це сукупна вартість, що включає витрати на інфраструктуру $T_{\inf ra}$ (сервери, бази даних) та витрати на розробку й підтримку C_{dev}.

При цьому мають виконуватися обмеження (SLA):

– час відгуку: середній час відгуку не повинен перевищувати порогове значення $T_{\max}$ для 95-го перцентиля запитів: $T_{res}^{p95}(A,L) \le T_{\max}$;

– рівень помилок: відсоток помилкових запитів не повинен перевищувати порогове значення $Err(A,L) : Err_{\max} : Err(A,L) \le Err_{\max}$;

– доступність: загальний час безвідмовної роботи $U(A)$ має бути не нижчим за цільовий показник $U_{\min}$ (наприклад, 99.9%): $U(A) \ge U_{\min}$.

Цільова функція полягає у виборі такої архітектури A, яка забезпечує найкраще співвідношення продуктивності до вартості при дотриманні всіх заданих обмежень щодо якості обслуговування.

1.4 Постановка задач дослідження

Метою кваліфікаційної роботи є проведення системного аналізу сучасних архітектурних підходів до розробки масштабованих веб-застосунків, порівняльна оцінка їх ефективності та формування практичних рекомендацій щодо вибору оптимальних рішень для забезпечення масштабованості веб-систем. Для досягнення поставленої мети необхідно виконати наступні завдання:

– провести огляд існуючих проблем масштабування веб-застосунків та визначити основні вимоги до архітектур масштабованих систем;

– проаналізувати сучасні архітектурні підходи, що використовуються для забезпечення масштабованості веб-застосунків;

– провести порівняльний аналіз переваг та недоліків виявлених підходів за визначеними критеріями;

– розробити спрощений прототип веб-застосунку з використанням обраного підходу;

– оцінити продуктивність та масштабованість реалізованого рішення при різному навантаженні або змінних параметрах;

– сформулювати рекомендації для побудови масштабованих веб-застосунків.

Ці задачі забезпечують досягнення основної мети дослідження – створення комплексного розуміння підходів до розробки масштабованих веб-застосунків та формування практичних рекомендацій, що можуть бути застосовані при проєктуванні реальних систем різного масштабу та складності.

2 ВИБІР ТА ОБҐРУНТУВАННЯ МЕТОДУ РОЗВ’ЯЗАННЯ

2.1 Концепція масштабованості веб-застосунків

Масштабованість веб-застосунку являє собою здатність системи ефективно адаптуватися до зростаючого навантаження шляхом додавання ресурсів без значної деградації продуктивності або необхідності фундаментальних архітектурних змін. Це критична характеристика сучасних веб-систем, яка безпосередньо впливає на користувацький досвід, операційні витрати та конкурентоспроможність продукту [8].

У контексті веб-розробки масштабованість охоплює декілька взаємопов'язаних аспектів. По-перше, це технічна здатність інфраструктури обробляти зростаючу кількість запитів, користувачів або обсяги даних. По-друге, це архітектурна гнучкість, що дозволяє розширювати функціональність без порушення існуючих компонентів. По-третє, це економічна ефективність масштабування, коли збільшення потужності не призводить до непропорційного зростання витрат.

Значення масштабованості особливо зростає в умовах непередбачуваного зростання аудиторії. Історія знає численні приклади, коли успішні продукти стикалися з критичними проблемами через недостатню підготовку до масштабування. Twitter у ранні роки свого існування регулярно стикався з простоями через неможливість обробити пікове навантаження, що негативно вплинуло на репутацію сервісу. Instagram, навпаки, з самого початку проєктував архітектуру з урахуванням швидкого зростання, що дозволило компанії масштабуватися від нуля до мільйонів користувачів протягом кількох місяців.

Масштабованість тісно пов'язана з концепцією еластичності системи. Еластичність передбачає не лише можливість нарощувати потужності, а й здатність динамічно зменшувати ресурси під час спаду навантаження, що особливо важливо для оптимізації витрат у хмарних середовищах. Сучасні веб-застосунки часто демонструють циклічні коливання навантаження – денні піки

активності, сезонні сплески або навантаження, спричинені маркетинговими кампаніями. Ефективна масштабованість дозволяє автоматично адаптуватися до цих змін [9].

Важливим аспектом є також масштабованість команди розробки. Добре спроєктована архітектура дозволяє паралельно працювати над різними компонентами системи без створення конфліктів та блокувань. Це особливо критично для великих організацій, де над одним продуктом можуть працювати десятки або сотні розробників. Мікросервісна архітектура, модульна структура фронтенду та чітке розмежування відповідальностей сприяють збільшенню швидкості розробки без втрати якості коду.

2.1.1 Горизонтальна vs вертикальна масштабованість

Існує два фундаментальні підходи до масштабування веб-застосунків, кожен з яких має свої переваги, обмеження та сценарії оптимального застосування.

Вертикальна масштабованість, також відома як "scale up", передбачає збільшення потужності існуючих серверів шляхом додавання ресурсів: процесорних ядер, оперативної пам'яті, дискового простору або покращення мережевої пропускної здатності. Цей підхід є найпростішим у реалізації, оскільки не вимагає змін в архітектурі застосунку [10]. Додавання RAM або заміна процесора на більш потужний часто може бути виконано без модифікації коду.

Основні переваги вертикального масштабування включають простоту впровадження та відсутність необхідності у складних механізмах розподілу навантаження. Застосунок продовжує працювати на одному сервері, що спрощує розробку, тестування та підтримку системи. Відсутність мережевої затримки між компонентами забезпечує високу швидкість обробки запитів. Управління даними також залишається простішим, оскільки немає необхідності в розподілених транзакціях або складних механізмах синхронізації.

Проте вертикальне масштабування має суттєві обмеження. По-перше, існує фізична межа потужності одного сервера. Навіть найпотужніші машини мають скінченну кількість процесорних ядер і обсяг пам'яті. По-друге, вартість масштабування зростає нелінійно – подвоєння потужності часто коштує значно більше, ніж вдвічі. По-третє, цей підхід створює єдину точку відмови: якщо сервер виходить з ладу, весь застосунок стає недоступним [11]. Нарешті, вертикальне масштабування часто вимагає простою системи для оновлення обладнання.

Горизонтальна масштабованість, або "scale out", передбачає додавання нових серверів або інстансів застосунку для розподілу навантаження. Замість одного потужного сервера система використовує множину менш потужних машин, які працюють паралельно. Цей підхід вимагає більш складної архітектури, але забезпечує практично необмежену масштабованість.

Ключовою перевагою горизонтального масштабування є відсутність теоретичної межі потужності. Систему можна розширювати, додаючи нові сервери, без необхідності заміни існуючих. Економічна ефективність також є вищою, оскільки можна використовувати commodity hardware – стандартні, відносно недорогі сервери. Відмовостійкість системи значно покращується: вихід з ладу одного сервера не призводить до повної недоступності застосунку, оскільки інші інстанси продовжують обробляти запити.

Горизонтальне масштабування особливо ефективне в хмарних середовищах, де нові інстанси можуть автоматично створюватися та видалятися залежно від навантаження. Amazon Web Services, Google Cloud Platform та Microsoft Azure надають інструменти для автоматичного масштабування, що дозволяє оптимізувати витрати та забезпечити стабільну продуктивність.

Однак горизонтальне масштабування супроводжується певними викликами. Застосунок повинен бути спроєктований як stateless, тобто не зберігати стан між запитами на рівні сервера, або використовувати централізоване сховище сесій. Розподілені системи вимагають механізмів балансування навантаження, які інтелектуально розподіляють запити між доступними серверами.

Управління даними стає складнішим: необхідні розподілені бази даних, механізми реплікації та забезпечення консистентності даних між вузлами.

Мережева латентність також стає фактором, оскільки компоненти системи фізично розділені та взаємодіють через мережу. Складність моніторингу та діагностики проблем зростає пропорційно кількості серверів. Розробники повинні враховувати можливість часткових відмов системи та проєктувати застосунок з урахуванням принципів розподілених систем.

На практиці оптимальна стратегія часто поєднує обидва підходи. Вертикальне масштабування може використовуватися для компонентів, які важко розподілити, наприклад баз даних або кеш-серверів, тоді як горизонтальне масштабування застосовується для stateless компонентів, таких як веб-сервери або API endpoints. Такий гібридний підхід дозволяє збалансувати простоту впровадження з можливістю практично необмеженого масштабування.

2.1.2 Метрики оцінювання масштабованості

Об'єктивна оцінка масштабованості веб-застосунку вимагає систематичного вимірювання ключових метрик продуктивності. Ці показники дозволяють не лише визначити поточний стан системи, а й прогнозувати поведінку під різними типами навантаження та ідентифікувати потенційні проблеми до того, як вони вплинуть на користувачів.

Пропускна здатність визначає кількість запитів, які система здатна обробити за одиницю часу, вимірюється в запитах за секунду (requests per second, RPS) або транзакціях за секунду (transactions per second, TPS). Ця метрика є прямим індикатором того, скільки користувачів система може обслуговувати одночасно. Для ефективного аналізу пропускної здатності необхідно вимірювати її під різним навантаженням та ідентифікувати точку, після якої додавання ресурсів не призводить до пропорційного збільшення продуктивності.

Час відповіді (response time) та латентність є критичними метриками ко-

ристувацького досвіду. Середній час відповіді показує типову продуктивність системи, проте він може маскувати проблеми, які впливають на частину користувачів. Тому важливо аналізувати перцентилі розподілу: 50-й перцентиль (медіана) показує типовий досвід, 95-й та 99-й перцентилі виявляють проблеми, які впливають на найменш щасливих користувачів. Дослідження Google показали, що затримка в 100 мілісекунд може призвести до зниження конверсії на 1%, що підкреслює критичність оптимізації часу відповіді.

Утилізація ресурсів відображає, наскільки ефективно система використовує доступні потужності. Моніторинг використання CPU, оперативної пам'яті, дискового вводу-виводу та мережевої пропускної здатності дозволяє ідентифікувати bottleneck компоненти. Ідеальна утилізація залежить від типу ресурсу: CPU зазвичай має працювати на рівні 60-70% під нормальним навантаженням, залишаючи резерв для пікових періодів, тоді як оперативна пам'ять може використовуватися більш інтенсивно завдяки механізмам кешування [12].

Коефіцієнт масштабування (scalability coefficient) показує, наскільки ефективно додавання ресурсів покращує продуктивність. Ідеальне лінійне масштабування означає, що подвоєння кількості серверів призводить до подвоєння пропускної здатності. На практиці через накладні витрати на координацію та комунікацію між вузлами коефіцієнт масштабування зазвичай менший за одиницю. Аналіз цього показника допомагає визначити, чи варто продовжувати горизонтальне масштабування, чи необхідно оптимізувати архітектуру.

Доступність (availability) вимірює відсоток часу, коли система функціонує коректно. Вона часто виражається у вигляді "дев'яток": 99,9% доступності означає не більше 43 хвилин простою на рік, тоді як 99,99% дозволяє лише 4,3 хвилини простою. Для критичних систем, таких як фінансові платформи або медичні сервіси, навіть 99,99% може бути недостатньо. Масштабована архітектура повинна забезпечувати високу доступність через механізми резервування та автоматичного відновлення.

Час до першого байту (Time to First Byte, TTFB) є особливо важливим для веб-застосунків, оскільки він безпосередньо впливає на сприйняття швидкості

сторінки користувачем. Ця метрика вимірює час від ініціювання запиту до отримання першого байту відповіді сервером. TTFB включає час мережевої затримки, час обробки запиту на сервері та час генерації відповіді.

Коефіцієнт помилок (error rate) показує відсоток запитів, які завершилися невдало. Зростання коефіцієнту помилок під навантаженням часто сигналізує про проблеми масштабованості: таймаути з'єднань, вичерпання ресурсів або проблеми з базою даних. Важливо розрізняти різні типи помилок: клієнтські помилки (4xx) зазвичай не пов'язані з масштабованістю, тоді як серверні помилки (5xx) часто вказують на проблеми з інфраструктурою.

Вартість обслуговування запиту (cost per request) є економічною метрикою, яка показує фінансову ефективність масштабування. Ідеальна система має константну або навіть зменшувану вартість обслуговування запиту при зростанні навантаження завдяки ефекту масштабу та кращій утилізації ресурсів. Зростання цієї метрики сигналізує про неефективність архітектури.

Для комплексного аналізу масштабованості необхідно проводити різні типи навантажувальних тестів. Базове навантажувальне тестування визначає продуктивність системи під типовим навантаженням. Стрес-тестування виявляє межі системи, поступово збільшуючи навантаження до точки відмови. Spike-тестування перевіряє поведінку системи при раптових сплесках трафіку. Soak-тестування, або тестування на витривалість, виявляє проблеми, які проявляються лише після тривалої роботи під навантаженням, такі як витоки пам'яті або деградація продуктивності.

2.2 Архітектурні патерни для масштабованих систем

2.2.1 Монолітна архітектура: переваги та обмеження

Монолітна архітектура представляє традиційний підхід до розробки веб-застосунків, де всі компоненти системи об’єднані в єдину кодову базу та розго-

ртаються як один неподільний блок. Незважаючи на появу альтернативних підходів, монолітна архітектура залишається релевантною для багатьох сценаріїв, особливо на ранніх стадіях розвитку продукту.

У класичному монолітному застосунку презентаційний рівень, бізнес-логіка та рівень доступу до даних тісно інтегровані в межах однієї кодової бази. Розгортання відбувається цілком: будь-яка зміна, навіть незначна, вимагає перезбірки та перезавантаження всього застосунку. Всі компоненти виконуються в єдиному процесі та поділяють спільні ресурси: пам'ять, процесорний час та підключення до бази даних.

Основною перевагою монолітної архітектури є простота розробки та розгортання на початкових етапах проєкту. Відсутність необхідності в складній інфраструктурі, оркестрації контейнерів чи налаштуванні міжсервісної комунікації дозволяє команді зосередитися на бізнес-логіці. Один репозиторій, одна збірка, один процес розгортання – це значно знижує когнітивне навантаження та operational overhead, особливо для невеликих команд.

Продуктивність монолітних застосунків у певних сценаріях може перевершувати розподілені системи. Внутрішні виклики методів відбуваються в межах одного процесу без мережевої латентності, що характерна для міжсервісної комунікації. Транзакції в базі даних залишаються простими, оскільки немає необхідності в розподілених транзакціях або eventual consistency моделях. Локальні виклики також дозволяють уникнути складностей серіалізації та десеріалізації даних.

Налагодження та трейсинг у монолітному застосунку є більш прямолінійними. Розробник може встановити breakpoint та покроково пройти через весь flow виконання коду від контролера до репозиторію. Стек-трейси помилок містять повну інформацію про ланцюжок викликів без розриву на межах сервісів. Це значно спрощує діагностику проблем та скорочує час на виправлення багів.

Тестування монолітних застосунків також має свої переваги. Інтеграційні тести можуть покривати охоплювати повний процес обробки запиту без необхідності імітувати зовнішні сервіси або налаштовувати складне тестове середо-

вище. End-to-end тести працюють з реальною системою без додаткових абстракцій. Для невеликих та середніх проєктів це забезпечує достатню впевненість у коректності системи.

Однак монолітна архітектура демонструє суттєві обмеження при масштабуванні. Перша проблема полягає в неможливості незалежного масштабування окремих компонентів. Якщо один модуль системи, наприклад обробка зображень, вимагає значних CPU-ресурсів, необхідно масштабувати весь застосунок, включаючи компоненти, які не потребують додаткової потужності. Це призводить до неефективного використання ресурсів та зростання витрат.

Розгортання монолітного застосунку стає ризикованішим зі збільшенням його розміру. Кожне оновлення впливає на всю систему, що підвищує ймовірність регресій та несподіваних побічних ефектів. Час розгортання зростає пропорційно розміру застосунку: компіляція, збірка, запуск можуть займати десятки хвилин, що уповільнює цикл розробки та ускладнює впровадження практик continuous deployment.

Технологічна гнучкість у монолітній архітектурі обмежена. Вибір технологічного стеку здійснюється на початку проєкту, і його зміна згодом вимагає значних зусиль. Використання різних мов програмування або фреймворків для різних частин системи практично неможливе. Це особливо проблематично для довгострокових проєктів, де окремі компоненти могли б отримати користь від специфічних технологій.

Масштабування команди розробки також стикається з викликами в монолітній архітектурі. Зі зростанням кількості розробників зростає ймовірність конфліктів при злитті коду, тісного зв’язку між модулями та порушення boundaries між компонентами. Onboarding нових членів команди стає складнішим, оскільки необхідно розуміти всю систему цілком. Паралельна робота над різними функціями ускладнюється через спільну кодову базу.

Відновлення після збоїв у монолітних системах також має свої особливості. Якщо один компонент викликає критичну помилку або споживає надмірно багато ресурсів, це може призвести до падіння всього застосунку. Відсутність

ізоляції означає, що проблеми поширюються швидко та впливають на всі функції системи одночасно.

Незважаючи на обмеження, монолітна архітектура залишається оптимальним вибором для багатьох сценаріїв. Стартапи на етапі product-market fit, внутрішні корпоративні системи з передбачуваним навантаженням, або проєкти з невеликими командами можуть значно виграти від простоти монолітного підходу. Amazon та Netflix розпочинали з монолітних архітектур та мігрували до мікросервісів лише після досягнення масштабу, що виправдовував цю складність [13].

Сучасні підходи до монолітної архітектури включають концепцію modular monolith – структурованого монолітного застосунку з чітко визначеними boundaries між модулями, слабким зв'язком та високою cohesion. Такий підхід зберігає переваги монолітної архітектури, водночас закладаючи фундамент для потенційної міграції до мікросервісів у майбутньому, якщо це стане необхідним.

2.2.2 Мікросервісна архітектура

Мікросервісна архітектура представляє радикально відмінний підхід до побудови веб-застосунків, розбиваючи монолітну систему на набір невеликих, незалежних сервісів, кожен з яких відповідає за конкретну бізнес-функцію [14]. Цей архітектурний стиль набув широкого поширення в останнє десятиліття завдяки компаніям-піонерам, таким як Netflix, Amazon та Spotify, які продемонстрували його ефективність для систем надвеликого масштабу.

Фундаментальним принципом мікросервісної архітектури є декомпозиція застосунку на сервіси, організовані навколо бізнес-можливостей [15]. Кожен мікросервіс є повністю автономним та включає власну бізнес-логіку, базу даних та інтерфейси взаємодії. Наприклад, в e-commerce платформі можуть існувати окремі сервіси для управління каталогом товарів, обробки замовлень, управлін-

ня користувачами, платіжних операцій та рекомендаційної системи [16].

Незалежність розгортання є однією з ключових переваг мікросервісів. Кожен сервіс має власний життєвий цикл розробки та може оновлюватися без впливу на інші частини системи. Це дозволяє командам практикувати continuous deployment з високою частотою релізів. Netflix, наприклад, здійснює тисячі розгортань на день завдяки мікросервісній архітектурі. Можливість швидко випускати зміни в продакшн значно прискорює цикл зворотного зв'язку та дозволяє організаціям швидше реагувати на потреби ринку.

Технологічна гетерогенність стає реальністю в мікросервісному середовищі. Різні сервіси можуть бути написані різними мовами програмування, використовувати різні бази даних та фреймворки залежно від специфіки завдання. Сервіс обробки зображень може використовувати Python з бібліотеками машинного навчання, тоді як high-throughput API буде реалізований на Go для максимальної продуктивності. Сервіси з складною бізнес-логікою можуть використовувати Java або C#, а швидкі прототипи – Node.js або Ruby.

Масштабованість у мікросервісній архітектурі стає більш гранульованою та ефективною. Замість масштабування всього застосунку можна масштабувати лише ті сервіси, які відчувають підвищене навантаження. Якщо сервіс пошуку отримує в десять разів більше запитів, ніж сервіс адміністрування, можна запустити додаткові інстанси лише сервісу пошуку. Це дозволяє оптимізувати використання ресурсів та знижувати операційні витрати.

Відмовостійкість системи покращується завдяки ізоляції сервісів. Збій одного мікросервісу не обов'язково призводить до недоступності всього застосунку. Шаблони проєктування, такі як Circuit Breaker, Bulkhead та Timeout, дозволяють ізолювати проблеми та забезпечити graceful degradation. Якщо сервіс рекомендацій недоступний, користувачі все ще можуть переглядати товари та робити замовлення, лише втрачаючи персоналізовані рекомендації.

Організаційна структура команд природно відображається в мікросервісній архітектурі через закон Конвея, який стверджує, що архітектура системи відображає комунікаційну структуру організації [17]. Кожна команда може воло-

діти одним або кількома мікросервісами та нести повну відповідальність за їх розробку, тестування, розгортання та підтримку. Це сприяє формуванню cross-functional команд з високою автономією та швидкістю прийняття рішень.

Однак мікросервісна архітектура супроводжується значною складністю, яка може перевищити переваги для невеликих або середніх проєктів. Розподілені системи за своєю природою складніші за монолітні. Мережеві виклики між сервісами додають латентність та можливість відмов, які необхідно обробляти на рівні коду. Гарантування консистентності даних між сервісами вимагає складних паттернів, таких як Saga pattern або Event Sourcing.

Operational complexity зростає експоненційно з кількістю сервісів. Замість одного монолітного застосунку команда повинна управляти десятками або сотнями окремих сервісів. Це вимагає досконалої інфраструктури: оркестрація контейнерів через Kubernetes, service mesh для управління міжсервісною комунікацією, централізоване логування та distributed tracing для діагностики проблем [18]. Організації повинні інвестувати в DevOps практики та інструменти автоматизації для ефективного управління цією складністю.

Тестування мікросервісних систем представляє унікальні виклики. Модульні тести окремих сервісів залишаються простими, але інтеграційне тестування вимагає координації множини сервісів. Contract testing з використанням інструментів, таких як Pact, допомагає забезпечити сумісність між сервісами без необхідності запускати всю систему. End-to-end тести стають повільнішими та крихкішими через залежність від багатьох компонентів.

Міжсервісна комунікація вимагає ретельного проєктування. Синхронна комунікація через REST або gRPC є простішою для розуміння, але створює щільний зв'язок між сервісами та знижує відмовостійкість. Асинхронна комунікація через черги повідомлень або event streams забезпечує кращу масштабованість та decoupling, але ускладнює debugging та вимагає обробки eventual consistency.

Data management у мікросервісній архітектурі базується на принципі database per service – кожен сервіс має власну базу даних, яка не доступна на-

пряму іншим сервісам. Це забезпечує автономність та дозволяє обирати оптимальну технологію зберігання даних для кожного сервісу. Проте це ускладнює виконання запитів, які потребують даних з множини сервісів, та робить неможливими традиційні ACID транзакції на рівні бази даних.

API Gateway часто використовується як єдина точка входу для клієнтських застосунків, що дозволяє абстрагувати складність мікросервісної архітектури. Gateway виконує маршрутизацію запитів, агрегацію відповідей від множини сервісів, автентифікацію, rate limiting та інші cross-cutting concerns. Проте сам API Gateway може стати bottleneck або single point of failure, якщо не спроєктований належним чином.

Стратегія міграції від монолітної до мікросервісної архітектури вимагає обережності. Strangler Fig pattern рекомендує поступово виділяти функціональність у окремі сервіси, поступово "задушуючи" монолітний застосунок [19]. Починати варто з периферійних функцій або нових фіч, поступово переміщуючи критичні компоненти. Повна переписування системи з нуля рідко виправдовує себе та несе значні ризики.

2.2.3 Serverless підхід та event-driven архітектура

Serverless архітектура представляє еволюцію хмарних обчислень, де розробники повністю абстрагуються від управління інфраструктурою та фокусуються виключно на бізнес-логіці. Незважаючи на назву, сервери існують, але відповідальність за їх підтримку, масштабування та забезпечення доступності повністю лежить на постачальнику хмарних послуг.

Функції як сервіс (Function as a Service, FaaS) є основою serverless підходу. AWS Lambda, Google Cloud Functions, Azure Functions та інші платформи дозволяють розгортати невеликі, сфокусовані функції, які виконуються у відповідь на події [20]. Код виконується в ephemeral контейнерах, які автоматично створюються при надходженні запиту та знищуються після завершення оброб-

ки. Розробник платить лише за фактичний час виконання коду, вимірюваний у мілісекундах, та кількість викликів.

Автоматичне масштабування є природною властивістю serverless платформ. Якщо застосунок отримує один запит на хвилину або тисячу запитів на секунду, платформа автоматично створює необхідну кількість інстансів функцій для обробки навантаження. Розробникам не потрібно налаштовувати auto-scaling групи, load balancers чи прогнозувати потреби в ресурсах. Це особливо цінно для застосунків з нерівномірним або непередбачуваним навантаженням.

Економічна модель serverless є привабливою для багатьох сценаріїв. Відсутність витрат на холості ресурси означає, що застосунок, який не отримує запити, не генерує витрат. Це різко контрастує з традиційними підходами, де сервери працюють цілодобово незалежно від навантаження. Для застосунків з періодичним використанням, таких як scheduled jobs, webhooks або backend для мобільних застосунків, serverless може забезпечити значну економію.

Event-driven природа serverless архітектури добре поєднується з сучасними асинхронними паттернами. Функції можуть автоматично виконуватися у відповідь на різноманітні події: HTTP запити, зміни в базі даних, повідомлення в чергах, завантаження файлів у сховище або scheduled triggers. Це дозволяє природно моделювати складні бізнес-процеси як ланцюжки подій [21].

Швидкість розробки та time to market значно покращуються в serverless середовищі. Розробники можуть зосередитися на написанні бізнес-логіки без необхідності налаштовувати сервери, мережі, load balancers чи системи моніторингу. Багато рутинних завдань, такі як logging, metrics та tracing, надаються платформою out of the box. Це особливо цінно для невеликих команд або стартапів з обмеженими ресурсами.

Polyglot programming природно підтримується в serverless архітектурі. Різні функції можуть бути написані різними мовами програмування залежно від завдання. Python може використовуватися для data processing, Node.js для API endpoints, Go для high-performance обробки, а Java для інтеграції з legacy системами. Кожна функція є незалежною одиницею, що дозволяє обирати оптималь-

ний інструмент для кожної задачі.

Проте serverless архітектура має суттєві обмеження та trade-offs. Cold start latency є однією з найбільш відомих проблем. Коли функція не викликалася протягом певного часу, контейнер знищується, і наступний виклик вимагає часу на створення нового контейнера та ініціалізацію runtime. Для Java або .NET функцій холодний старт може тривати кілька секунд, що неприйнятно для interactive застосунків. Різні стратегії, такі як provisioned concurrency або warming functions, можуть пом'якшити цю проблему, але за додаткову плату.

Обмеження виконання також є характерними для serverless платформ. AWS Lambda, наприклад, має максимальний час виконання 15 хвилин, обмеження на розмір deployment package та memory footprint. Це робить serverless непридатним для довготривалих обчислень, batch processing великих обсягів даних або memory-intensive операцій. Для таких завдань потрібні альтернативні підходи або гібридні архітектури.

Vendor lock-in є значною проблемою для serverless застосунків. Код часто тісно інтегрований з специфічними сервісами хмарного провайдера: AWS DynamoDB, S3, SNS або їх еквівалентів в інших хмарах. Міграція між провайдерами може вимагати значного переписування коду. Фреймворки, такі як Serverless Framework або AWS SAM, намагаються створити абстракції, але повна портабельність залишається недосяжною.

Налагодження та локальна розробка є складнішими в безсерверному середовищі. Неможливо запустити повну копію робочого оточення локально, оскільки багато сервісів існують лише в хмарі. Емулятори та локальні засоби тестування допомагають, але не завжди точно відтворюють поведінку робочого середовища. Розподілене трасування та якісне логування стають критично важливими для діагностики проблем.

Управління станом у функціях без стана вимагає зовнішніх систем зберігання. Бази даних, кеші або сховища об'єктів використовуються для персистентності даних між викликами. Це створює додаткові залежності та потенційні вузькі місця. Управління пулом з'єднань із базою даних стає проблематичним,

оскільки кожна функція створює власні підключення, що може призвести до вичерпання лімітів з'єднань.

Складність архітектури може зрости з кількістю функцій. Застосунок може складатися з сотень незалежних функцій зі складними залежностями між ними. Управління версіями, розгортаннями та налаштуваннями для множини функцій вимагає досконалих інструментів та процесів. Підхід «Інфраструктура як код» (через Terraform, CloudFormation або Serverless Framework) стає обов'язковим для підтримки контролю.

Моніторинг та спостережуваність вимагають нових підходів. Традиційні метрики, такі як використання ЦП або пам'яті, менш релевантні в безсерверному контексті. Важливішими стають метрики на рівні бізнес-логіки: кількість викликів, тривалість виконання, частота помилок, частота холодних стартів. X-Ray від AWS або подібні інструменти допомагають візуалізувати потік виконання через множину функцій.

Безсерверний підхід оптимальний для певних сценаріїв: серверна частина API (API-бекенди) для мобільних застосунків, обробка веб-хуків, заплановані завдання, конвеєри трансформації даних, серверна частина для IoT (IoT-бекенди) або обробка подій. Для монолітних застосунків з постійним високим навантаженням традиційні підходи можуть бути більш економічно ефективними. Гібридні архітектури, які поєднують безсерверні функції для змінних навантажень з традиційними обчисленнями для базового трафіку, часто забезпечують оптимальний баланс.

Подіє-орієнтована архітектура (EDA) представляє парадигму проєктування систем, де потік виконання визначається подіями – значущими змінами стану, які відбуваються в системі або її оточенні. Цей підхід радикально відрізняється від традиційної моделі "запит-відповідь" та забезпечує високий рівень слабкого зв'язування (роз'єднання), масштабованості та реактивності.

Фундаментальною концепцією EDA є подія – незмінний факт про те, що щось відбулося в певний момент часу. Події носять декларативний характер і описують зміну стану без спроб змінити майбутнє. Наприклад, "OrderPlaced",

"PaymentReceived", "InventoryUpdated" є подіями, які фіксують факти, що вже відбулися. На відміну від команд, які є імперативними інструкціями виконати дію, події є незмінними записами історії системи.

Архітектура базується на трьох основних компонентах: виробники подій, споживачі подій та брокер подій. Виробники генерують події у відповідь на зміни стану або зовнішні тригери. Споживачі підписуються на події, які їх цікавлять, та реагують на них асинхронно. Брокер подій виступає посередником, забезпечуючи надійну доставку подій від виробників до споживачів без прямого зв'язку між ними.

Слабке зв'язування є найбільш значущою перевагою подіє-орієнтованої архітектури. Виробники не знають і не потребують знати про споживачів своїх подій. Нові споживачі можуть додаватися без модифікації виробників. Це дозволяє системі еволюціонувати без ризику порушити існуючу функціональність. Якщо потрібно додати новий функціонал, наприклад відправку сповіщення електронною поштою при створенні замовлення, достатньо створити нового споживача для події OrderPlaced без зміни коду, що створює замовлення.

Масштабованість природно вбудована в подіє-орієнтовані системи. Споживачі можуть обробляти події паралельно, і їх кількість може динамічно змінюватися залежно від навантаження. Якщо обробка подій уповільнюється, можна додати більше споживачів для розподілу навантаження. Брокери подій, такі як Apache Kafka або Amazon Kinesis, спроєктовані для горизонтального масштабування та можуть обробляти мільйони подій на секунду.

Асинхронність обробки дозволяє системам бути більш реактивними. Виробники не блокуються в очікуванні на завершення обробки події споживачами. Після публікації події виробник може негайно повернути відповідь користувачу, а фактична обробка відбуватиметься в фоновому режимі. Це особливо цінно для операцій, які вимагають значного часу: відправка електронних листів, генерація звітів або інтеграція з зовнішніми системами.

Часове роз'єднання означає, що виробники та споживачі не повинні бути доступними одночасно. Події зберігаються в брокері подій, і споживачі можуть

обробити їх пізніше, коли стануть доступними. Це значно покращує відмовостійкість системи. Якщо сервіс обробки платежів тимчасово недоступний, події PaymentInitiated накопичуються в черзі та будуть оброблені після відновлення сервісу.

Event Sourcing є потужним патерном, що природно поєднується з EDA. Замість зберігання лише поточного стану системи, Event Sourcing зберігає повну історію всіх подій, які призвели до цього стану. Поточний стан може бути відтворений шляхом відтворення всіх подій від початку. Це забезпечує повний журнал аудиту, можливість налагодження з подорожжю в часі та природну підтримку функціональності скасувати/повторити.

CQRS (Command Query Responsibility Segregation) часто використовується разом з подіє-орієнтованою архітектурою для оптимізації операцій читання та запису. Команди модифікують стан та генерують події, тоді як запити читають з денормалізованих моделей читання, оптимізованих для специфічних сценаріїв використання. Події синхронізують моделі запису та читання, дозволяючи кожній стороні використовувати оптимальну технологію зберігання [22].

Складні бізнес-процеси природно моделюються через події хореографії. Замість центрального оркестратора, який координує всі кроки процесу, кожен сервіс реагує на релевантні події та публікує нові події після завершення своєї роботи. Наприклад, процес оформлення замовлення може включати ланцюжок подій: OrderPlaced → PaymentProcessed → InventoryReserved → ShippingScheduled → OrderCompleted.

Проте подіє-орієнтована архітектура вносить значну складність у систему. Узгодженість у кінцевому підсумку стає нормою: після публікації події потрібен час, перш ніж усі споживачі оновлять свій стан. Це вимагає ретельного проєктування користувацького досвіду для обробки ситуацій, коли дані ще не синхронізовані. Користувачі повинні бачити індикатори обробки або отримувати нотифікації про завершення операцій.

Налагодження та усунення несправностей стають складнішими через асинхронну природу системи. Трасування причинно-наслідкових зв'язків між

подіями вимагає ідентифікаторів кореляції та складного розподіленого трасування. Коли щось йде не так, важко визначити, яка саме ланка в ланцюжку обробки подій відповідальна за проблему. Якісне централізоване логування та інструменти візуалізації подій стають критично важливими.

Управління схемами подій є важливим аспектом довгострокової підтримки системи. Структура подій може еволюціонувати з часом, але старі події залишаються в системі. Реєстри схем, такі як Confluent Schema Registry, допомагають керувати версіями схем та забезпечувати зворотну та пряму сумісність. Принципи еволюції схем повинні дотримуватися для уникнення критичних змін.

Порядок обробки подій може бути критичним для деяких сценаріїв використання. Розподілені брокери подій не завжди гарантують суворий порядок між розділами. Якщо порядок важливий, необхідно проєктувати стратегію розділення так, щоб пов'язані події потрапляли в один розділ. В якості альтернативи, споживачі повинні бути спроєктовані як ідемпотентні та здатні обробляти події в довільному порядку.

Ідемпотентність обробки подій є критичною вимогою. Брокери подій зазвичай гарантують доставку щонайменше один раз, що означає можливість дублювання подій. Споживачі повинні бути спроєктовані так, щоб повторна обробка тієї ж події не призводила до небажаних побічних ефектів. Це може досягатися через усунення дублікатів на основі ідентифікатора події або через природно ідемпотентні операції.

Черги недоставлених повідомлень (або "мертвих листів") необхідні для обробки "отруйних" подій, які постійно викликають помилки при обробці. Замість нескінченних спроб повторення, такі події переміщуються у спеціальну чергу для подальшого ручного аналізу. Це запобігає блокуванню обробки інших подій та дозволяє системі продовжувати функціонувати навіть при наявності проблемних випадків.

Моніторинг подіє-орієнтованих систем вимагає специфічних метрик: затримка події (між генерацією та обробкою), пропускна здатність по топікам, рі-

вень збоїв споживачів, глибина черг недоставлених повідомлень. Сповіщення повинно налаштовуватися на аномалії в цих метриках, оскільки проблеми в асинхронних системах можуть не проявлятися негайно.

Подіє-орієнтована архітектура оптимальна для систем з високими вимогами до масштабованості, слабкого зв'язування та реактивності. Платформи електронної комерції, системи Інтернету речей (IoT), аналітика в реальному часі, фінансові торгові системи та застосунки для спільної роботи значно виграють від цього підходу. Проте для простих CRUD-застосунків або систем з вимогами суворої узгодженості традиційні синхронні підходи можуть бути більш прийнятними.

2.2.4 Порівняльна характеристика підходів

Всебічне розуміння архітектурних патернів (або шаблонів) вимагає систематичного порівняння їх характеристик, переваг та обмежень у контексті конкретних бізнес-вимог та технічних обмежень. Вибір архітектури є стратегічним рішенням, яке визначає траєкторію розвитку системи на роки вперед.

Складність розробки та підтримки. Монолітна архітектура забезпечує найнижчий поріг входу та простоту розробки на початкових етапах. Єдина кодова база, прямі виклики методів та відсутність розподіленої природи спрощують адаптацію нових розробників та прискорюють початкову розробку. Проте зі зростанням розміру кодової бази складність зростає нелінійно, і в певний момент монолітний застосунок стає важким для розуміння та модифікації.

Мікросервісна архітектура має високу початкову складність, але забезпечує кращу довгострокову підтримуваність завдяки чіткому розмежуванню відповідальності. Кожен сервіс залишається достатньо невеликим для повного розуміння командою. Проте операційні накладні витрати значно зростають: необхідність управляти десятками сервісів, їх взаємодією, версіонуванням та конвеєрами розгортання вимагає зрілих DevOps-практик.

Безсерверна архітектура мінімізує операційний тягар, перекладаючи його на хмарного провайдера, але вводить нові виклики у вигляді прив'язки до постачальника, складнощів налагодження та необхідності розуміння специфічних обмежень платформи. Подіє-орієнтований підхід додає концептуальну складність через асинхронну природу та узгодженість у кінцевому підсумку, що вимагає зміни мислення розробників.

Масштабованість та продуктивність. Монолітна архітектура обмежена в масштабуванні: можливе лише вертикальне масштабування або горизонтальне масштабування всього застосунку, що неефективно, якщо вузьке місце локалізоване в одному компоненті. Проте для застосунків з низьким та середнім трафіком монолітна архітектура може демонструвати кращу продуктивність завдяки відсутності мережевих накладних витрат.

Мікросервіси забезпечують гранульовану масштабованість – кожен сервіс масштабується незалежно відповідно до своїх потреб. Це оптимізує використання ресурсів та витрати. Проте міжсервісна комунікація додає латентність, і для сценаріїв з високою пропускною здатністю та низькою затримкою це може стати проблемою. Кешування, асинхронна комунікація та оптимізація мережевих викликів стають критично важливими.

Безсерверна архітектура забезпечує автоматичне масштабування від нуля до практично необмеженої кількості одночасних виконань. Це ідеально для нерівномірних або непередбачуваних навантажень. Проте холодні старти та обмеження часу виконання обмежують її застосування для чутливих до затримки або довготривалих операцій. Подіє-орієнтовані системи природно масштабуються через паралельну обробку подій, але вимагають ретельного проєктування стратегій розділення.

Вартість володіння. Монолітна архітектура має низькі початкові витрати: мінімальна інфраструктура, простий конвеєр розгортання, не потрібні складні інструменти моніторингу. Проте при зростанні масштабу витрати на недовикористані ресурси зростають, оскільки неможливо масштабувати лише вузькі місця компонентів.

Мікросервіси мають високі операційні витрати через складність інфраструктури: платформи оркестрації (Kubernetes), сервісні сітки (service meshes), централізоване логування, розподілене трасування. Проте при правильній реалізації вони забезпечують кращу ефективність використання ресурсів при великому масштабі.

Безсерверний підхід має найбільш гнучку модель витрат: оплата лише за фактичне використання без витрат на ресурси, що простоюють. Для низькооб'ємних застосунків це може означати майже нульові витрати, проте при високооб'ємних традиційні підходи можуть бути економічнішими.

Відмовостійкість та надійність. Монолітні системи мають єдину точку відмови: збій будь-якого компонента може обрушити весь застосунок. Проте простіша природа спрощує розуміння режимів відмови. Мікросервіси забезпечують кращу ізоляцію збоїв: проблема в одному сервісі не обов'язково впливає на інші, особливо при правильному використанні переривачів (circuit breakers) та резервних механізмів.

Безсерверні платформи забезпечують високу доступність одразу: керована інфраструктура, автоматичне перемикання на резерв, розгортання у кількох зонах доступності. Проте збої у провайдера можуть вплинути на всі безсерверні функції одночасно. Подіє-орієнтовані системи природно стійкі завдяки асинхронній природі: тимчасові збої можуть бути поглинуті чергами подій, а обробка продовжується після відновлення.

Швидкість розробки та час виходу на ринок. Монолітна архітектура забезпечує найшвидшу розробку MVP: проста структура, відсутність необхідності в складній інфраструктурі, просте тестування. Це робить її оптимальним вибором для стартапів на етапі пошуку відповідності продукту ринку (product-market fit).

Мікросервіси уповільнюють початкову розробку через необхідність налаштування інфраструктури, але прискорюють подальшу розробку завдяки паралельній розробці незалежними командами та частими, незалежними розгортаннями. Безсерверний підхід може забезпечити найшвидший час виходу на

ринок для певних типів застосунків завдяки мінімальним витратам на налаштування та зосередженню на бізнес-логіці.

Технологічна гнучкість. Монолітна архітектура прив'язана до початкового вибору стеку технологій. Міграція на іншу технологію вимагає повного переписування. Мікросервіси дозволяють багатомовну архітектуру: різні сервіси можуть використовувати різні технології, оптимальні для їх конкретних потреб. Нові технології можуть поступово впроваджуватися без переписування всієї системи.

Безсерверний підхід також підтримує багатомовне програмування, проте з обмеженнями опцій середовища виконання, що підтримуються платформою. Подіє-орієнтована архітектура є технологічно-незалежною на концептуальному рівні та може бути реалізована з різними стеками технологій.

Тестування та якість коду. Монолітні застосунки легше тестувати наскрізно (end-to-end): весь стек застосунку може бути запущений локально. Проте модульні тести (unit tests) можуть бути складнішими через тісний зв'язок між компонентами. Мікросервіси мають відмінну ізоляцію модульних тестів, але складніше інтеграційне та наскрізне тестування: необхідно оркеструвати множину сервісів або широко використовувати "заглушки" (test doubles).

Безсерверні функції легко модульно тестувати, але інтеграційне тестування вимагає хмарного середовища або локальних емуляторів, які не завжди точно відтворюють поведінку в робочому середовищі. Подіє-орієнтовані системи вимагають тестування сценаріїв узгодженості в кінцевому підсумку та уважного розгляду порядку подій та ідемпотентності.

Організаційна відповідність. Монолітна архітектура оптимальна для невеликих, спільно розташованих команд, де комунікаційні накладні витрати низькі. Мікросервіси природно узгоджуються з масштабуванням організації через володіння сервісами командами та чітких меж. Це дозволяє великим організаціям розподілити роботу між багатьма автономними командами.

Безсерверний підхід знижує організаційну складність через керовану інфраструктуру, дозволяючи навіть маленьким командам досягати значного мас-

штабу. Подіє-орієнтована архітектура вимагає сильної міжкомандної координації для еволюції схем та контрактів подій, але забезпечує слабке зв'язування на рівні реалізації.

Сценарії застосування. Монолітна архітектура оптимальна для MVP, внутрішніх інструментів, застосунків з низьким та середнім трафіком, або коли досвід команди обмежений. Мікросервіси виправдані для великомасштабних застосунків, систем, що вимагають незалежної масштабованості компонентів, організацій з багатьма командами, або коли різні частини системи мають значно різні нефункціональні вимоги.

Безсерверний підхід ідеальний для обробки подій, запланованих завдань, API зі змінними патернами трафіку, застосунків, що вимагають нульових витрат на підтримку. Подіє-орієнтована архітектура оптимальна для систем реального часу, IoT-платформ, систем, що вимагають журналів аудиту, застосунків зі складними бізнес-процесами, що вимагають високого рівня роз'єднання.

Міграційні шляхи. Важливо розуміти, що архітектурний вибір не є остаточним. Типовий еволюційний шлях починається з монолітної архітектури для швидкого MVP, поступово виділяються окремі обмежені контексти у мікросервіси при досягненні певного масштабу, подіє-орієнтовані патерни впроваджуються для роз'єднання та асинхронної обробки, а безсерверні функції використовуються для специфічних сценаріїв використання, такі як фонові завдання або обробники подій.

Кожна архітектура має свою "найкращу точку застосування" – сценарії, де вона проявляє максимальну ефективність. Успішні системи часто використовують гібридні підходи, поєднуючи різні патерни: монолітне ядро для тісно пов'язаної бізнес-логіки, мікросервіси для незалежно масштабованих компонентів, безсерверні функції для обробки подій, та подіє-орієнтовану комунікацію для роз'єднання. Ключ до успіху – розуміння компромісів та вибір архітектурних патернів, що узгоджується з реальними бізнес-потребами, можливостями команди та траєкторією росту, а не з ажіотажем або трендами (табл. 1.7).

Таблиця 1.7 – Порівняння архітектурних патернів

Характеристика	Монолітна	Мікросервісна	Безсерверна	Подіє-орієнтована
Складність	Низька	Висока постійно	Середня	Висока концептуально
Масштабованість	Обмежена	Гранульована	Автоматична	Природна
Вартість	Низька	Висока	За використання	Середня
Відмово стійкість	Єдина точка відмови	Ізольовані збої	Висока доступність	Стійка через черги
Швидкість MVP	Найшвидша	Повільна	Швидка	Середня
Тестування	Легке E2E, складне unit	Легке unit, складне E2E	Складне інтеграційне	Складна узгодженість
Команди	Невеликі	Великі, автономні	Малі/середні	Потрібна координація

Висновки за розділом 2

У другому розділі було проведено комплексний аналіз методів проєктування масштабованих веб-систем та обґрунтування вибору архітектурного рішення.

У підрозділі 2.1 розглянуто концепцію масштабованості як здатність системи адаптуватися до зростаючого навантаження. Проаналізовано фундаментальні відмінності між вертикальним («scale up») та горизонтальним («scale out») масштабуванням, визначено їхні переваги та сценарії застосування. Також виділено ключові метрики ефективності, такі як пропускна здатність, латентність та коефіцієнт помилок, які дозволяють об’єктивно оцінювати стан системи.

У підрозділі 2.2 здійснено детальне порівняння архітектурних патернів:

монолітної, мікросервісної, безсерверної (Serverless) та подіє-орієнтованої архітектури. Визначено, що монолітний підхід забезпечує швидкий старт та простоту розробки, тоді як мікросервіси та Serverless надають необхідну гнучкість та відмовостійкість для високонавантажених систем. Окрему увагу приділено компромісам, пов'язаним зі складністю інфраструктури та вартістю підтримки кожного з підходів.

Таким чином, теоретичний аналіз підтвердив, що вибір методу реалізації залежить від специфічних вимог проєкту, етапу його життєвого циклу та прогнозованого навантаження. Для забезпечення довгострокової ефективності та надійності системи найперспективнішим є використання гнучких архітектурних патернів, що дозволяють поєднувати незалежне масштабування компонентів з оптимізацією витрат ресурсів.

3 ПРОГРАМНА РЕАЛІЗАЦІЯ

3.1 Інструментарій для розробки масштабованих розподілених систем

Для практичної реалізації та дослідження підходів до масштабування було обрано стек технологій на базі Node.js (NestJS) та контейнеризації Docker. Docker виступає ключовим засобом управління середовищем виконання, що дозволяє ізолювати компоненти системи, забезпечити їхню переносність та автоматизувати розгортання як у монолітному, так і в мікросервісному режимах [23]. Його використання суттєво підвищує ефективність управління ресурсами, забезпечуючи повторюваність експериментів та гнучкість конфігурації, що є ключовими вимогами при розробці високонавантажених систем.

Обраний технологічний стек демонструє низку функціональних можливостей, які роблять його оптимальним для вирішення задач масштабування:

а) контейнеризація (Docker): технологія, що дозволяє пакувати застосунок та його залежності у стандартизовані одиниці (контейнери), гарантуючи ідентичність середовища на етапах розробки, тестування та експлуатації;

б) модульність (NestJS): архітектурний фреймворк, що сприяє створенню слабко зв'язаних модулів, які легко виділяються в окремі мікросервіси завдяки використанню патерну Dependency Injection [24];

в) оркестрація (Docker Compose): інструмент для визначення та запуску багатоконтейнерних застосунків, який дозволяє декларативно описувати інфраструктуру та легко масштабувати окремі сервіси за допомогою параметра replicas;

г) типізація (TypeScript): використання суворої типізації підвищує надійність коду та спрощує підтримку розподілених систем, зменшуючи кількість помилок на етапі компіляції [25];

д) асинхронна комунікація (Redis): інтеграція з брокером повідомлень за-

безпечує реалізацію подіє-орієнтованої архітектури (Event-Driven), що є критично важливим для розв'язання (decoupling) сервісів;

е) продуктивність: неблокуюча модель вводу-виводу Node.js дозволяє ефективно обробляти велику кількість одночасних з'єднань, що є критичним для I/O-intensive задач [26].

Docker значно полегшує процес моделювання складних архітектур. Він спрощує розгортання таких елементів, як бази даних, брокери повідомлень та самі веб-сервіси в єдиній віртуальній мережі. Це дозволяє ефективно імітувати виробниче середовище на локальній машині без значних накладних витрат.

Однією з головних переваг є можливість динамічного масштабування. Завдяки конфігурації Docker Compose, легко змінюється кількість екземплярів (реплік) сервісу, що дозволяє проводити порівняльний аналіз продуктивності між одноекземплярною (монолітною) та багатоекземплярною (мікросервісною) архітектурами.

Прозорість мережевої взаємодії є ще однією важливою рисою. Внутрішній DNS Docker дозволяє сервісам комунікувати за назвами контейнерів, абстрагуючись від конкретних IP-адрес, що спрощує конфігурацію розподіленої системи.

Інтеграція з інструментами моніторингу робить цей стек універсальним для досліджень. Легкість підключення зовнішніх систем збору метрик дозволяє точно вимірювати вплив архітектурних рішень на пропускну здатність та час відгуку.

Використання NestJS у поєднанні з Docker є потужним підходом для створення масштабованих архітектур. Це дозволяє забезпечити ізоляцію бізнес-логіки через розгортання окремих сервісів, налаштування механізмів балансування навантаження та відповідних черг повідомлень для мінімізації затримок.

Автоматизація розгортання інфраструктури через docker-compose.yml гарантує швидке відновлення середовища, що є критично важливим для ітеративного процесу розробки та тестування.

3.2 Алгоритм розв’язання задачі створення прототипу веб-застосунку

Розробка масштабованого прототипу є комплексним завданням, яке вимагає багатоетапного підходу, що забезпечує не лише функціональність, але й можливість проведення навантажувального тестування для порівняння архітектур. Нижче детально описано ключові етапи, які реалізують створення такої системи.

Крок 1. Декомпозиція предметної області та проєктування сервісів. Створення архітектури розпочинається з виділення обмежених контекстів (Bounded Contexts). Це включає розподіл функціональності на незалежні сервіси: сервіс продуктів (Products Service), сервіс замовлень (Orders Service) та сервіс сповіщень (Notification Service). Кожен сервіс проєктується з власною зоною відповідальності. Це забезпечує можливість незалежного розгортання та масштабування.

Крок 2. Налаштування контейнеризації та оркестрації. Створення Dockerfile для кожного сервісу забезпечує ізоляцію середовища виконання (Node.js runtime). У файлі docker-compose.yml описується взаємодія між сервісами, та брокером повідомлень (Redis). Визначаються мережеві політики (networks) для забезпечення внутрішньої комунікації та прокидання портів для зовнішнього доступу. Ключовим моментом є налаштування блоку deploy для сервісу, що підлягає масштабуванню, що дозволяє запускати декілька реплік одного контейнера.

Крок 3. Реалізація механізмів комунікації (Синхронна та Асинхронна). Для забезпечення взаємодії компонентів реалізуються два типи зв'язку. Синхронна взаємодія (REST API) використовується для операцій читання даних, де важлива негайна відповідь. Асинхронна взаємодія реалізується через Redis Pub/Sub для операцій, що можуть бути виконані у фоновому режимі (наприклад, обробка замовлення та відправка сповіщень). Це дозволяє знизити зв'язність системи (coupling) та підвищити відмовостійкість.

Крок 4. Впровадження механізмів балансування та масштабування.

Налаштовується вхідна точка (Load Balancer або внутрішній механізм Docker Round-Robin), яка розподіляє вхідний трафік між доступними екземплярами сервісу. Це дозволяє системі обробляти більшу кількість запитів шляхом горизонтального додавання нових інстансів (Horizontal Scaling) без зміни програмного коду клієнта. Для перевірки ефективності налаштовуються скрипти навантажувального тестування (k6), які імітують поведінку великої кількості користувачів.

Виконання описаних етапів забезпечує створення прототипу, який дозволяє експериментально підтвердити переваги мікросервісного підходу. Комплексний підхід до управління контейнерами, комунікацією та даними дозволяє мінімізувати вузькі місця (bottlenecks) та забезпечити стабільність системи під навантаженням.

3.3 Опис програми

У цьому розділі представлено детальний опис розробленого програмного коду та конфігураційних файлів, які забезпечують створення, розгортання та тестування масштабованого веб-застосунку. Об’єктом розробки є прототип розподіленої системи електронної комерції, спроєктований для дослідження механізмів масштабування та відмовостійкості під високим навантаженням [27]. Основна бізнес-логіка застосунку декомпозована на функціональні домени: перегляд каталогу товарів, за що відповідає сервіс «products-service», обробка замовлень, яку виконує «orders-service», та асинхронне інформування користувачів через «notification-service». Така архітектура дозволяє ізолювати процеси та незалежно масштабувати кожен компонент залежно від навантаження.

Лістинг коду інфркструктури та головного сервісу наведено в Додатку А.

Основні файли інфраструктури:

а) файл docker-compose.yml – цей файл містить декларативний опис інфраструктури та визначає топологію системи. Він включає:

1) опис сервісів (products-service, orders-service, notification-service);

2) налаштування інфраструктурних компонентів (Redis);

3) конфігурацію мереж та томів для персистентності даних;

4) параметри масштабування (replicas).

б) скрипт load-test.js – файл сценарію для інструменту k6, що використовується для генерації синтетичного навантаження. Він визначає:

1) етапи тестування зі зростанням кількості віртуальних користувачів;

2) порогові значення (thresholds) для успішності тесту (час відгуку, відсоток помилок);

3) цільові ендпоінти для HTTP-запитів.

Компоненти програмної реалізації (NestJS):

а) Сервіс продуктів (Products Module):

1) products.controller.ts – обробляє вхідні HTTP-запити, виступає точкою входу для навантажувального тестування;

2) products.service.ts – містить бізнес-логіку отримання даних, імітує затримки бази даних;

б) Сервіс замовлень (Orders Module):

1) orders.service.ts – реалізує логіку створення замовлення та публікації подій у Redis;

2) взаємодія з Redis Client – забезпечує асинхронну передачу повідомлень;

в) Сервіс сповіщень (Notifications Module):

1) app.controller.ts – підписується на події з черги повідомлень (Event Pattern);

2) обробник подій – виконує фонову роботу, демонструючи відмовостійкість системи [28].

Опис ключових механізмів:

а) механізм Service Discovery – забезпечується внутрішнім DNS Docker, дозволяючи сервісам знаходити один одного за іменами хостів;

б) механізм реплікації – реалізований на рівні Docker Compose, дозволяє запускати N ідентичних копій products-service для розподілу навантаження;

в) механізм Pub/Sub – використовує Redis для розв'язання залежності між створенням замовлення та його подальшою обробкою.

Висновки за розділом 3

У третьому розділі виконано практичну реалізацію програмного комплексу для дослідження ефективності масштабування веб-застосунків.

У підрозділі 3.1 обґрунтовано вибір технологічного стеку на базі Node.js (NestJS), Docker та Redis. Визначено, що використання контейнеризації та оркестрації дозволяє ефективно моделювати як монолітну, так і мікросервісну архітектуру в межах єдиного експериментального середовища. Застосування Redis забезпечує реалізацію асинхронної взаємодії, що є критичним для побудови слабкозв'язаних розподілених систем.

У підрозділі 3.2 розроблено покроковий алгоритм створення масштабованого прототипу. Алгоритм охоплює декомпозицію предметної області на обмежені контексти, налаштування контейнерного середовища, реалізацію гібридної моделі комунікації (синхронна/асинхронна) та впровадження механізмів балансування навантаження. Такий підхід забезпечує гнучкість системи та її готовність до динамічного горизонтального масштабування.

У підрозділі 3.3 наведено детальний опис програмної реалізації компонентів системи електронної комерції. Описано структуру конфігураційних файлів інфраструктури (Docker Compose) та бізнес-логіку основних сервісів. Представлено реалізацію ключових механізмів розподілених систем: Service Discovery, реплікації інстансів та патерну Pub/Sub. Також описано інструментарій для навантажувального тестування (k6), необхідний для верифікації результатів.

Таким чином, розроблений програмний прототип є повнофункціональ-

ною платформою для проведення експериментів. Він дозволяє на практиці перевірити теоретичні положення щодо переваг горизонтального масштабування та оцінити вплив архітектурних рішень на продуктивність системи під навантаженням.

4 РЕЗУЛЬТАТИ ОБЧИСЛЮВАЛЬНОГО ЕКСПЕРИМЕНТУ ТА ЇХ АНАЛІЗ

4.1 Підготовка даних та методологія обчислювального експерименту

Метою даного обчислювального експерименту є підтвердження ефективності горизонтального масштабування веб-застосунків в рамках мікросервісної архітектури порівняно з монолітним підходом. Основна увага приділена перевірці таких ключових характеристик:

– пропускна здатність системи (Throughput/RPS) при збільшенні навантаження;

– стабільність часу відгуку (Latency) при зростанні кількості паралельних запитів;

– здатність системи до ізоляції збоїв (Fault Tolerance).

Для забезпечення достовірності експериментальних результатів було розроблено сценарії навантажувального тестування, які імітують реальну поведінку користувачів:

– сценарій «Читання»: інтенсивні запити на отримання списку продуктів (Read-heavy load) [29];

– сценарій «Запис»: створення замовлень з асинхронною обробкою (Write-heavy load).

Для проведення експерименту було використано наступні інфраструктурні рішення:

– k6: інструмент для генерації навантаження та збору метрик продуктивності;

– Docker Compose: для емуляції монолітного (1 інстанс) та кластерного (3 інстанси) середовищ;

– NestJS Monorepo: для організації кодової бази досліджуваних сервісів.

Тестування реалізовано через послідовність наступних кроків:

а) Запуск Базового Сценарію (імітація моноліту):

1) розгортання products-service у кількості 1 екземпляра;

2) подача навантаження від 50 до 100 віртуальних користувачів;

б) Запуск Масштабованого Сценарію (мікросервісний кластер):

1) масштабування products-service до 3 екземплярів;

2) подача аналогічного навантаження;

в) Оцінка якості результатів на основі таких критеріїв:

1) RPS – кількість успішних запитів за секунду;

2) P95 Latency – час, за який обробляється 95% запитів;

3) Error Rate – відсоток невдалих запитів;

4.2 Результати експерименту

Для підтвердження ефективності горизонтального масштабування було проведено серію стрес-тестів з імітацією високого обчислювального навантаження (CPU-bound tasks). Тестування проводилося при одночасному підключенні 200 віртуальних користувачів (VU).

Отримані результати підтвердили (дод. Б, В) гіпотезу про лінійну залежність пропускної здатності від кількості вузлів у кластері мікросервісів.

На рисунку 4.1 наведено порівняння середньої пропускної здатності (RPS). У базовому сценарії (1 інстанс) система змогла обробити 19,76 запитів за секунду, досягнувши межі своїх обчислювальних можливостей. Після масштабування сервісу продуктів до 3-х екземплярів, пропускна здатність зросла до 58,78 запитів за секунду. Коефіцієнт масштабування склав 2,97, що свідчить про майже ідеальну ефективність розподілу навантаження без суттєвих накладних витрат на оркестрацію.

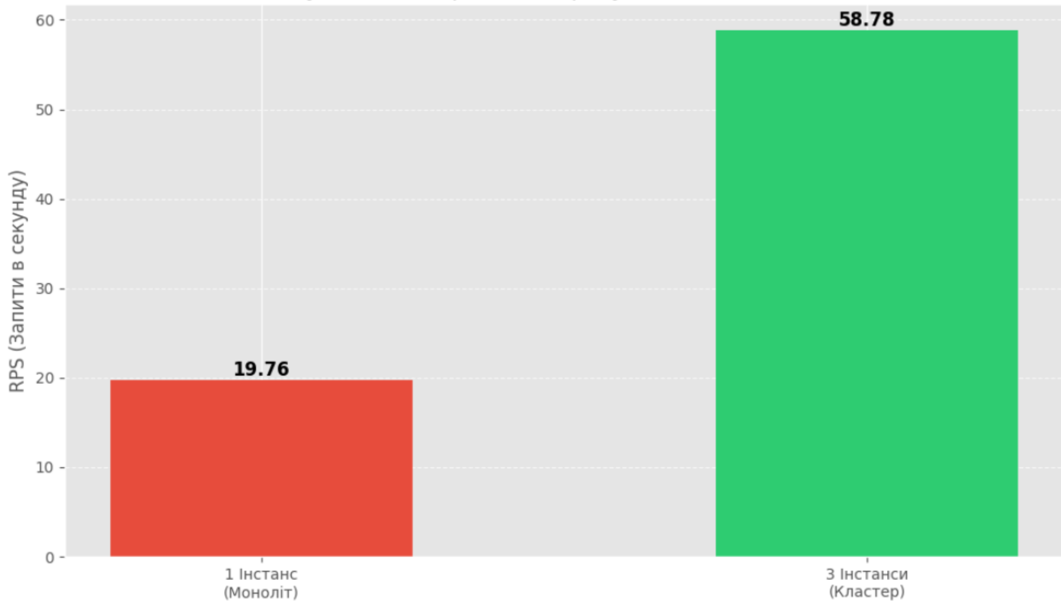

Рисунок 4.1 – Порівняння пропускної здатності (RPS)

На рисунку 4.2 відображено динаміку часу відгуку (95-й перцентиль). В умовах нестачі ресурсів (1 інстанс) користувачі стикалися зі значними затримками - до 9,95 с. Завдяки розподілу запитів між трьома контейнерами, час очікування скоротився втричі - до 3,29 с, що значно підвищило якість обслуговування (QoS).

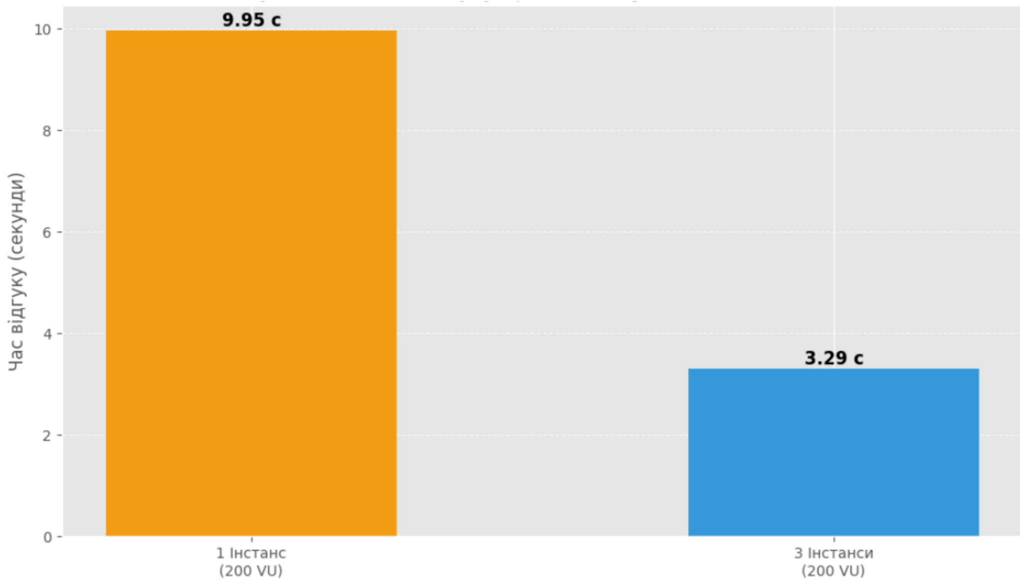

Рисунок 4.2 – Час відгуку при піковому навантажені

Окрім перцентильних значень, важливим індикатором стабільності системи є аналіз граничних значень затримок (Max Latency), отриманих з протоколів тестування (див. Додаток Б та В). У базовому сценарії максимальний час очікування відповіді досягав критичної позначки у 10,1 с, що є неприйнятним для інтерактивних веб-застосунків і призводить до розриву з'єднання по таймауту (HTTP Timeout) [30]. Після горизонтального масштабування максимальна затримка знизилася до 3,39 с, що гарантує успішну обробку навіть найважчих запитів у межах стандартних таймаутів браузера. Також варто відзначити загальну продуктивність (Total Throughput): за фіксований час тестування (50 с) масштабована архітектура успішно обробила 2942 запити проти 1094 у монолітному варіанті, що підтверджує зростання корисної дії системи майже втричі. Зведені показники ефективності наведено в таблиці 4.1.

Таблиця 4.1 – Порівняльний аналіз показників ефективності архітектур

Показник	Монолітний підхід	Мікросервісний кластер	Покращення
Середня пропускна здатність	19,76 req/s	58,78 req/s	2,97x (↑)
Всього оброблено	1094	2942	2,69x (↑)
Latency Avg (Середній час	5,46 с	1,77 с	3,08x (↓)
Latency P95 (95% запитів)	9,95 с	3,29 с	3,02x (↓)
Latency Max (Найгірший випадок)	10,10 с	3,39 с	2,98x (↓)
Data Received (Швидкість мережі)	7,5 kB/s	22 kB/s	2,93x (↑)

Отже, в лінійному масштабуванні експеримент довів, що мікросервісна архітектура дозволяє долати фізичні обмеження одного сервера. Збільшення кількості вузлів у 3 рази призвело до зростання продуктивності у 2,97 рази, що підтверджує ефективність горизонтального масштабування.

Точкове (гранулярне) масштабування: На відміну від моноліту, де потрібно дублювати всю систему, мікросервіси дозволяють виділяти ресурси лише для тих компонентів, які знаходяться під навантаженням (у нашому випадку - products-service), що економить ресурси.

Підвищення якості обслуговування (QoS): Розподіл навантаження дозволив усунути черги обробки, скоротивши час очікування користувача втричі (з ~10 с до ~3,3 с) навіть в умовах пікового навантаження.

Висновки за розділом 4

У четвертому розділі наведено результати обчислювального експерименту, спрямованого на перевірку ефективності горизонтального масштабування веб-застосунків у порівнянні з монолітним підходом.

У підрозділі 4.1 сформульовано методологію дослідження та визначено умови проведення експерименту. Описано підготовку тестового середовища з використанням інструментів k6 та Docker Compose. Розроблено сценарії навантажувального тестування («Читання» та «Запис»), які імітують реальну поведінку користувачів, та визначено ключові метрики оцінки: пропускну здатність (RPS), час відгуку (Latency) та відмовостійкість.

У підрозділі 4.2 здійснено аналіз отриманих емпіричних даних. Стрес-тестування при навантаженні у 200 віртуальних користувачів підтвердило гіпотезу про лінійне масштабування мікросервісної системи. Збільшення кількості екземплярів сервісу з одного до трьох призвело до зростання середньої пропускної здатності з 19,76 до 58,78 запитів за секунду (коефіцієнт масштабування 2,97). Одночасно зафіксовано трикратне зменшення часу відгуку (з 9,95 с до 3,29 с), що суттєво покращило якість обслуговування (QoS).

Таким чином, результати експерименту кількісно підтвердили переваги горизонтального масштабування.

ВИСНОВКИ

Результати дослідження архітектурних підходів до розробки масштабованих веб-застосунків підтверджують ефективність обраної стратегії проєктування. Висновки базуються на комплексному системному аналізі та практичній реалізації прототипу.

Проведене оцінювання архітектурних рішень для бекенду за допомогою методу аналізу ієрархій визначило мікросервісний підхід як оптимальний (інтегральна оцінка 0,47). Визначальним фактором стала перевага у ключовому критерії масштабованості порівняно з монолітною архітектурою. Реалізація системи на базі NestJS та Docker довела здатність до ефективного горизонтального масштабування та ізоляції відмов.

Система продемонструвала високу стабільність під час навантажувального тестування інструментом k6, підтвердивши коректність роботи механізмів розподілу навантаження. Результати можуть бути застосовані при проєктуванні високонавантажених E-commerce платформ та систем обробки замовлень у реальному часі.

Наукова та технічна значущість роботи полягає в обґрунтуванні методології вибору архітектури на основі кількісних показників, а не лише експертних оцінок. Технічний внесок включає реалізацію гнучкої схеми взаємодії сервісів, що дозволяє проводити ітеративний "bottleneck-аналіз" та оптимізувати вузькі місця системи без зупинки її роботи. Це сприяє зниженню операційних витрат та підвищенню відмовостійкості бізнес-процесів.

Перспективним напрямом розвитку є адаптація розробленого рішення для середовищ оркестрації контейнерів, таких як Kubernetes, та впровадження автоматизованих конвеєрів CI/CD для прискорення доставки оновлень. Також запропонована архітектура може бути розширена модулями моніторингу та логування для підвищення спостережуваності системи в продуктовому середовищі.

ПЕРЕЛІК ДЖЕРЕЛ ПОСИЛАННЯ

1. Халтурін А. Аналіз архітектурних патернів для високонавантажених веб-систем. *Науково-технічна конференція «Комп'ютерно-інтегровані технології автоматизації технологічних процесів на транспорті та у виробництві»* : зб. матеріалів конференції (м. Харків, 25 листопада 2025 р.). Секція 4. Комп'ютерно-інтегровані технології автоматизації технологічних процесів на транспорті та у виробництві, 2024. С. 392–394.

2. Kleppmann M. Designing Data-Intensive Applications: The Big Ideas Behind Reliable, Scalable, and Maintainable Systems. Sebastopol : O'Reilly Media, 2017. 614 p.

3. Grigorik I. High Performance Browser Networking. 2nd Edition. O'Reilly Media, 2021. 400 p.

4. Ibryam B., Huß R. Kubernetes Patterns: Reusable Elements for Designing Cloud-Native Applications. 2nd Edition. O'Reilly Media, 2023. 280 p.

5. Richards M., Ford N. Fundamentals of Software Architecture: An Engineering Approach. O'Reilly Media, 2020. 432 p.

6. Feature-Sliced Design: Documentation. URL: https://feature-sliced.design/ (дата звернення: 15.11.2025).

7. Xu A. System Design Interview – An Insider's Guide. Vol. 2. Independently published, 2022. 500 p.

8. Abbott M. L., Fisher M. C. The Art of Scalability: Scalable Web Architecture, Processes, and Organizations for the Modern Enterprise. Boston : Addison-Wesley, 2015. 624 p.

9. Davis C. Cloud Native Patterns: Designing Change-tolerant Software. Manning Publications, 2020. 384 p.

10. Valadez J. Modern Systems Analysis and Design. 9th Edition. Pearson, 2020. 600 p.

11. Site Reliability Engineering: How Google Runs Production Systems / B. Beyer, C. Jones, J. Petoff, N. R. Murphy. Sebastopol : O'Reilly Media, 2016. 552 p.

12. Newman S. Building Microservices: Designing Fine-Grained Systems. 2nd Edition. Sebastopol : O'Reilly Media, 2020. 612 p.

13. Ford N., Parsons R., Kua P. Building Evolutionary Architectures. 2nd Edition. O'Reilly Media, 2023. 250 p.

14. Fowler M., Lewis J. Microservices: A Definition of This New Architectural Term. URL: https://martinfowler.com/articles/microservices.html (дата звернення: 19.11.2025).

15. Khononov V. Learning Domain-Driven Design: Aligning Software Architecture and Business Strategy. O'Reilly Media, 2021. 300 p.

16. Indrasiri K., Siriwardena P. Microservices for the Enterprise: Designing, Developing, and Deploying. Apress, 2021. 450 p.

17. Conway M. E. How Do Committees Invent? Datamation Magazine. 1968. Vol. 14, № 4. P. 28–31.

18. Kim G., Humble J. The DevOps Handbook: How to Create World-Class Agility, Reliability, and Security in Technology Organizations. 2nd Edition. IT Revolution Press, 2021. 480 p.

19. Fowler M. Strangler Fig Application. URL: https://martinfowler.com/bliki/StranglerFigApplication.html (дата звернення: 21.11.2025).

20. Serverless Architectures with AWS Lambda : AWS Whitepaper / Amazon. URL: https://docs.aws.amazon.com/pdfs/whitepapers/latest/serverless-architectures-lambda/serverless-architectures-lambda.pdf (дата звернення: 25.11.2025).

21. Bellemare A. Building Event-Driven Microservices. O'Reilly Media, 2020. 270 p.

22. Fowler M. CQRS. URL: https://martinfowler.com/bliki/CQRS.html (дата звернення: 22.11.2025).

23. Poulton N. Docker Deep Dive. Independently published, 2023. 430 p.

24. NestJS Fundamentals / Official Documentation. URL: https://docs.nestjs.com (дата звернення: 10.12.2025).

25. Vanderkam D. Effective TypeScript: 62 Specific Ways to Improve Your TypeScript. O'Reilly Media, 2020. 268 p.

26. Casciaro M., Mammino L. Node.js Design Patterns. 3rd Edition. Packt Publishing, 2020. 664 p.

27. Hunter T. Distributed Systems with Node.js: Building Enterprise-Ready Backend Services. O'Reilly Media, 2020. 350 p.

28. Redis Documentation. URL: https://redis.io/docs/ (дата звернення: 12.12.2025).

29. k6 Documentation: Open Source Load Testing Tool. URL: https://k6.io/docs/ (дата звернення: 14.12.2025).

30. Petrov A. Database Internals: A Deep Dive into How Distributed Data Systems Work. O'Reilly Media, 2020. 390 p.