Міністерство освіти і науки України
Харківський національний університет радіоелектроніки

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

КВАЛІФІКАЦІЙНА РОБОТА
Пояснювальна записка

рівень вищої освіти другий (магістерський)

ДОСЛІДЖЕННЯ ТА ПОРІВНЯННЯ МЕТОДІВ КЕШУВАННЯ МОНОЛІТНИХ І МІКРОСЕРВІСНИХ ВЕБСИСТЕМ
(тема)

Виконав:
здобувач 2 року навчання,
групи ІНФм-24-2
Воронін Д. О.
(прізвище, ініціали)

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

Тип програми освітньо-професійна

Освітня програма Інформатика
(повна назва освітньої програми)

Науковий керівник доц. Вечірська І. Д.
(посада, прізвище, ініціали)

Допускається до захисту

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

2025 р.

Харківський національний університет радіоелектроніки

Факультет Інформаційно-аналітичних технологій та менеджменту

Кафедра Інформатики

Рівень вищої освіти другий (магістерський)

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

Тип програми освітньо-професійна

Освітня програма Інформатика
(повна назва)

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

ЗАВДАННЯ
НА КВАЛІФІКАЦІЙНУ РОБОТУ

здобувачеві Вороніну Дмитру Олександровичу
(прізвище, ім'я, по батькові)

1. Тема роботи Дослідження та порівняння методів кешування монолітних і мікросервісних вебсистем

затверджена наказом університету від 14 листопада 2025 року № 1045Ст
2. Термін подання здобувачем роботи до екзаменаційної комісії 09 грудня 2025 р.
3. Вихідні дані до роботи наукова-технічна література, матеріали наукових конференцій, середовище розробки Visual Studio Code, обгортка для мови програмування TypeScript

4. Перелік питань, що потрібно опрацювати в роботі
1. Аналіз сучасних методів застосування кешування у вебсистемах.
2. Аналіз літературних джерел щодо апробації методів кешування у монолітних і мікросервісних вебсистемах
3. Формування покрокового алгоритму для кожного із вибраних методів кешування.
4. Проєктування вебзастосунків.
5. Розробка програмних застосунків різної архітектури та різними методами кешування для їх поточного порівняння.

5. Перелік графічного матеріалу із зазначенням креслеників, схем, плакатів, комп'ютерних ілюстрацій (п.5 включається до завдання за рішенням випускової кафедри) актуальність проблеми продуктивності вебсистем, роль кешування, збільшення навантаження в сучасних сервісах, об'єкт, предмет та мета дослідження, постановка задачі дослідження, архітектура монолітної системи, архітектура мікросервісної системи, моделі кешування реалізація експериментальних систем, сценарії навантажувального тестування, результати тестування і порівняння, висновки, перспективи та апробація

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

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

КАЛЕНДАРНИЙ ПЛАН

№ з/п	Назва етапів роботи	Строк / терміни виконання етапів роботи	Примітка
1	Отримання завдання на кваліфікаційну роботу	29.09.2025	
2	Аналіз завдання, підбір літератури	30.09.25-07.10.25	
3	Аналіз літератури з досліджуваної проблеми	08.10.25-14.10.25	
4	Особливості архітектур	15.10.25-20.10.25	
5	Дослідження методів кешування	21.10.25-27.10.25	
6	Програмна реалізація	28.10.25-05.11.25	
7	Обґрунтування отриманих результатів	06.11.25-28.11.25	
8	Оформлення пояснювальної записки	29.11.25-02.12.25	
9	Перевірка на нормоконтроль	03.12.25-04.12.25	
10	Перевірка на плагіат	04.12.25	
11	Рецензування	05.12.25	
12	Підготовка презентації та доповіді	06.12.25-07.12.25	
13	Занесення роботи в електронний архів	18.12.25	
14	Попередній захист кваліфікаційної роботи	18.12.25	

Дата видачі завдання 29 вересня 2025 р.

Здобувач _____
(підпис)

Керівник роботи _____ доц. Вечірська І. Д.
(підпис) (посада, прізвище, ініціали)

РЕФЕРАТ

Пояснювальна записка до кваліфікаційної роботи: 72 с., 7 табл., 7 рис., 31 джерел.

БАЗА ДАНИХ, КЕШУВАННЯ, МАСШТАБОВАНІСТЬ, МОНОЛІТНА АРХІТЕКТУРА, МІКРОСЕРВІСНА АРХІТЕКТУРА, НАВАНТАЖУВАЛЬНЕ ТЕСТУВАННЯ, ПРОДУКТИВНІСТЬ, DOCKER, MEMCACHED, NODE.JS, REDIS, REST API.

Об’єктом дослідження є процеси кешування у вебсистемах різних архітектурних типів − монолітних і мікросервісних.

Метою дослідження є підвищення продуктивності вебсистем шляхом формування рекомендацій щодо вибору оптимальних методів кешування для монолітних і мікросервісних архітектур.

У роботі проведено огляд сучасних методів кешування, класифікацію типів кешу за рівнем реалізації, а також досліджено особливості застосування кешу у вебархітектурах різного типу.

Наукова новизна роботи полягає у комплексному аналізі стратегій кешування в контексті різних архітектур вебсистем із практичним моделюванням реальних умов роботи застосунків.

Взаємозв’язок із іншими дослідженнями полягає у продовженні наукових робіт, присвячених підвищенню продуктивності вебсистем шляхом оптимізації обробки даних та використання сучасних механізмів кешування.

Рекомендації щодо використання результатів роботи можуть застосовуватися під час розробки вебсистем.

У результаті дослідження створено експериментальне середовище для аналізу кешування, проведено вимірювання ефективності різних методів, а також сформульовано практичні висновки та рекомендації щодо вибору технологій кешування для монолітних і мікросервісних вебсистем.

ABSTRACT

Explanatory note to the qualification work: 72 pages, 7 tables, 7 figures, 31 sources.

CACHING, DATABASE, DOCKER, LOAD TESTING, MEMCACHED, MICROSERVICE ARCHITECTURE, MONOLITHIC ARCHITECTURE, NODE.JS, PERFORMANCE, REDIS, REST API, SCALABILITY.

The object of the research is the caching processes in web systems of different architectural types – monolithic and microservice.

The aim of the research is to enhance the performance of web systems by developing recommendations for selecting optimal caching methods for monolithic and microservice architectures.

Scientific novelty lies in the comprehensive analyze of caching strategies within different web system architectures and in practical modeling of real operating conditions of applications.

Interconnection with other works consists in the continuation of research in the field of improving web system performance through data processing optimization, reducing access time, and applying modern caching mechanisms.

Recommendations for using the results can be applied in the development and optimization of web systems to select the optimal caching strategy depending on the architecture type, data volume, and performance requirements.

As a result of the research, an experimental environment for caching analysis was created, efficiency measurements of various methods were conducted, and practical conclusions and recommendations were formulated for selecting caching technologies for monolithic and microservice web systems.

ЗМІСТ

ПЕРЕЛІК УМОВНИХ ПОЗНАЧЕНЬ, СИМВОЛІВ, ОДИНИЦЬ, СКОРОЧЕНЬ І ТЕРМІНІВ

БД – база даних

API – Application Programming Interface

LFU – Least Frequently Used (найменш часто використовуваний)

LRU – Least Recently Used (найменш нещодавно використовуваний)

REST – Representational State Transfer

SQL – Structured Query Language

TTL – Time To Live (час життя)

ВСТУП

Сучасні вебсистеми є невід'ємною складовою цифрової інфраструктури, що забезпечує взаємодію між користувачами, сервісами та великими обсягами даних. Зі зростанням кількості користувачів, складності бізнес-логіки та обсягів інформації постає питання оптимізації продуктивності вебзастосунків. Одним із найефективніших інструментів для цього є кешування − технологія тимчасового збереження даних для прискорення доступу до них і зменшення навантаження на серверні ресурси.

Актуальність роботи полягає у тому, що кешування є ключовим елементом забезпечення продуктивності, масштабованості та надійності сучасних вебсистем. Зі збільшенням кількості мікросервісних архітектурних підходів виникає потреба у глибокому розумінні відмінностей у реалізації кешування для різних архітектурних типів. Якщо в монолітних системах кешування реалізується централізовано, то в мікросервісних воно має розподілений характер, що створює додаткові виклики − такі як забезпечення узгодженості даних, синхронізація кешів і мінімізація затримок при міжсервісній взаємодії.

Сучасний стан проблеми характеризується наявністю великої кількості бібліотек, фреймворків і рішень для кешування − таких як Redis, Memcached, Hazelcast, Nginx Cache, Spring Cache, тощо. Водночас питання вибору оптимальної стратегії кешування залежно від архітектури вебсистеми залишається відкритим. Більшість існуючих досліджень зосереджуються на покращенні швидкодії окремих рішень, але не проводять комплексного порівняльного аналізу кешування в монолітних і мікросервісних середовищах. Саме тому дослідження, спрямоване на систематизацію та порівняння підходів до кешування для різних архітектур, має як наукову, так і практичну значущість.

Таким чином, дана робота спрямована на розв'язання актуальної задачі оптимізації продуктивності вебсистем шляхом дослідження й порівняння методів кешування, що забезпечує її наукову новизну та практичну цінність для галузі сучасної веброзробки.

1 АНАЛІЗ ПРЕДМЕТНОЇ ОБЛАСТІ

1.1 Аналіз архітектурних підходів

1.1.1 Монолітна архітектура

Монолітна архітектура є класичним підходом до розробки вебзастосунків, у якому вся логіка системи реалізована у вигляді єдиного програмного модуля. У межах такого підходу клієнтська частина, бізнес-логіка та робота з базою даних поєднані в одному застосунку, який розгортається як єдине ціле [1].

Основна перевага монолітної архітектури полягає у її простоті. Початковий етап розробки, тестування та розгортання такого застосунку зазвичай є швидшим, адже всі компоненти працюють у єдиному середовищі та не потребують складної взаємодії між собою. Монолітна система має спільний простір пам’яті, тому виклики функцій та доступ до даних здійснюються без додаткових мережевих затримок. Це дозволяє досягти стабільної продуктивності при відносно невеликому навантаженні.

Водночас із зростанням обсягу проєкту монолітна архітектура стикається з низкою обмежень. По-перше, масштабування стає складним: щоб збільшити продуктивність, доводиться розгортати копії всього застосунку, навіть якщо навантаження зростає лише на окремий модуль. По-друге, тісна зв’язаність компонентів ускладнює підтримку та оновлення − зміна однієї частини коду може спричинити збоїв в інших. Крім того, тривалий час розгортання, складність тестування та обмежені можливості для командної роботи роблять цей підхід менш зручним для великих систем [1,2].

У контексті кешування монолітна архітектура зазвичай передбачає централізований кеш, спільний для всіх компонентів. Це спрощує реалізацію та контроль, але водночас створює ризик перевантаження пам’яті або блокування доступу, якщо обсяги даних надто великі.

1.1.2 Мікросервісна архітектура

Мікросервісна архітектура є сучасним підходом, що базується на ідеї розділення великої системи на невеликі незалежні сервіси, кожен з яких відповідає за певну функціональність. Наприклад, в інтернет-магазині можуть існувати окремі сервіси для управління користувачами, каталогом товарів, оплатою та доставкою. Кожен мікросервіс розробляється, розгортається та масштабується окремо [3].

Одним із головних принципів мікросервісної архітектури є незалежність. Кожен сервіс має власну базу даних, бізнес-логіку та інтерфейс доступу, частіше за все − REST API. Комунікація між сервісами відбувається через мережу, зазвичай за допомогою HTTP-запитів або черг повідомлень.

Мікросервісний підхід має кілька ключових переваг. Він забезпечує гнучке масштабування, оскільки дає змогу збільшувати ресурси лише для тих компонентів, які відчувають підвищене навантаження. Розробка та оновлення системи стають простішими, адже команди можуть працювати над окремими сервісами незалежно одна від одної. Такий підхід також підвищує надійність, тобто збій одного сервісу не зупиняє всю систему. Крім того, мікросервіси підтримують технологічну різноманітність, дозволяючи реалізовувати різні частини системи різними мовами програмування або фреймворками.

Однак мікросервісна архітектура має і свої недоліки. Основна складність полягає у забезпеченні стабільної комунікації між сервісами, моніторингу, централізованого логування та підтримання узгодженості даних. Також зростає складність у тестуванні та налагодженні системи через розподілений характер.

Щодо кешування, у мікросервісній архітектурі часто використовується розподілене кешування, коли кожен сервіс має власний локальний кеш або спільний доступ до централізованого сховища, наприклад Redis або Memcached. Це підвищує гнучкість, але водночас ускладнює підтримку консистентності кешованих даних між сервісами.

1.1.3 Порівняльний аналіз архітектур

Монолітна та мікросервісна архітектури є двома основними підходами до побудови сучасних вебсистем. Вони мають суттєві відмінності у структурі, способі взаємодії компонентів, масштабуванні та підходах до кешування. Розуміння цих відмінностей є ключовим для подальшого дослідження ефективності кешування.

На рисунку 1.1 зображено спрощену схему, що ілюструє різницю між монолітною та мікросервісною архітектурами вебсистем. У моноліті всі модулі – бізнес-логіка, обробка запитів, база даних, інтерфейс користувача – інтегровані в один застосунок [1,2]. У мікросервісній архітектурі кожна функціональна частина системи реалізована як окремий сервіс, який взаємодіє з іншими через API або повідомлення.

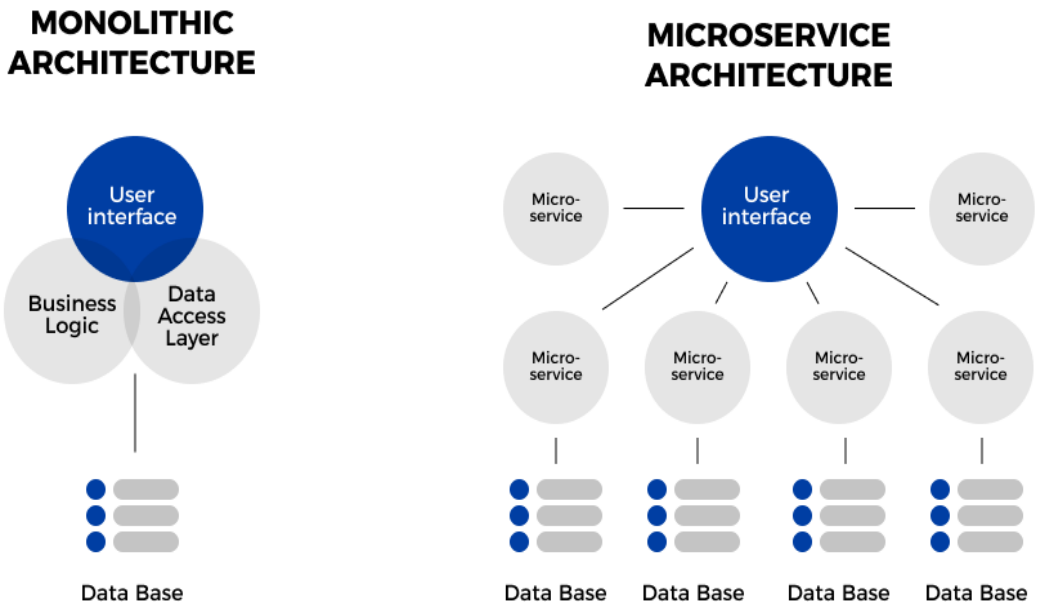

Рисунок 1.1 – Порівняння монолітної та мікросервісної архітектури вебсистем

Як видно з рисунку, у монолітній архітектурі всі частини системи тісно пов’язані, що спрощує розробку на початковому етапі, проте ускладнює масштабування та оновлення. Мікросервісна архітектура, навпаки, забезпечує

гнучкість і незалежність сервісів, але потребує додаткових механізмів координації, у тому числі для кешування даних.

Для більш детального аналізу проведено порівняння ключових характеристик обох підходів, представлено у таблиці 1.1.

Таблиця 1.1 – Порівняння характеристик монолітної та мікросервісної архітектур

Критерій	**Монолітна архітектура**	**Мікросервісна архітектура**
Масштабованість	Горизонтальне масштабування всієї системи	Незалежне масштабування окремих сервісів
Швидкодія	Висока при малій кількості користувачів	Висока при належному розподілі навантаження
Складність розгортання	Проста на початковому етапі	Вимагає складнішої інфраструктури
Підтримка та оновлення	Складна при великому коді	Гнучка, можливі незалежні оновлення
Надійність	Один збій може зупинити систему	Відмова одного сервісу не критична
Кешування	Централізоване, простіше в реалізації	Розподілене, потребує синхронізації

1.2 Кешування у вебсистемах різних типів

1.2.1 Принципи кешування

Кешування є одним із базових механізмів оптимізації роботи вебсистем, який дозволяє значно скоротити час доступу до часто запитуваних даних. Основна ідея полягає в тому, щоб зберігати результати попередніх обчислень або запитів у тимчасовому сховищі − кеші, щоб при повторному зверненні до цих самих даних не виконувати повторно дорогі операції, наприклад запити до бази даних чи зовнішніх API [4].

Принцип дії кешу базується на використанні закономірності: користувачі часто звертаються до тих самих даних. Наприклад, сторінка каталогу товарів або профіль користувача завантажується сотні разів на день, тоді як зміни в цих даних відбуваються рідко. Зберігаючи копію таких даних у кеші, система зменшує кількість запитів до основного сховища й тим самим підвищує швидкість відповіді.

Ефективне кешування ґрунтується на кількох базових принципах. Воно враховує просторово-часову локальність, оскільки дані, до яких нещодавно зверталися або які пов’язані між собою, з високою ймовірністю будуть потрібні знову. Водночас кеш є обмеженим ресурсом, тому необхідні механізми керування на кшталт LRU чи LFU. Важливим є й продумана політика інвалідації, яка визначає, коли кешовані дані слід оновлювати або видаляти через їх застарівання. Усе це має поєднуватися з правильним балансом між точністю та продуктивністю, адже надто часте оновлення кешу знижує його ефективність, а надто рідке призводить до використання застарілої інформації.

У веброзробці кешування використовується на різних рівнях − від браузера клієнта до бази даних на сервері. Відповідно до цього виділяють різні види кешів.

1.2.2 Види кешів

У вебсистемах застосовують кілька типів кешування [5], кожен з яких вирішує власну групу завдань і працює на різних рівнях інфраструктури. Одним із найпоширеніших є кеш браузера, який зберігає статичні ресурси, такі як HTML-сторінки, стилі, скрипти та зображення, безпосередньо на стороні клієнта.

На рівні мережевої інфраструктури важливу роль відіграє кеш проміжного рівня, який використовується у зворотних проксі-серверах або CDN. Його основне завдання скоротити шлях доставки контенту та знизити навантаження на основні сервери, що особливо важливо у глобально розподілених системах.

У межах самого застосунку часто застосовують кеш застосунку, який дозволяє зберігати результати обчислень, часто використовувані дані або готові HTML-фрагменти. У різних фреймворках такі механізми реалізовані по-різному, але працюють за спільним принципом: зменшити кількість повторних операцій і скоротити час генерації відповіді.

Також можна відокремити кеш бази даних, який зберігає результати SQL-запитів або об’єкти ORM, що дозволяє уникати повторної вибірки незмінних даних. У різних СУБД цей механізм реалізований специфічними засобами, наприклад query_cache у MySQL або Second Level Cache у Hibernate.

У великих системах, зокрема у мікросервісних архітектурах, важливим є розподілений кеш. Він забезпечує швидкий доступ до даних, що зберігаються в оперативній пам’яті одного або кількох серверів, і доступний усім компонентам системи. Найчастіше для цього використовують Redis, Memcached або Hazelcast.

На найвищому рівні інфраструктури працює кеш CDN. Він призначений для зберігання статичного контенту на серверах, що істотно скорочує затримку доступу та пришвидшує завантаження контенту.

Різні типи кешів часто поєднуються, створюючи багаторівневу систему кешування − multi-layer caching, де дані можуть зберігатися одночасно на рівні клієнта, проксі та сервера.

1.2.3 Інструменти кешування у веброзробці

Сучасні вебсистеми активно використовують спеціалізовані програмні рішення для організації кешу. Найбільш поширені з яких: Redis, Memcached, Varnish Cache, Nginx Cache, HTTP Cache-Control.

Платформа Redis − швидке in-memory сховище з підтримкою ключ-значення, структур даних (списки, множини, хеші) та TTL (часу життя даних). Використовується для зберігання сесій, результатів запитів, черг повідомлень. Redis підтримує реплікацію, кластери й механізми стійкості до збоїв.

Сховище Memcached − легковаговий розподілений кеш, орієнтований на простоту та швидкість. Підходить для зберігання короткочасних даних, таких як результати SQL-запитів.

Varnish Cache − проксі-сервер, що кешує HTTP-відповіді. Використовується для прискорення роботи сайтів із великим трафіком.

Nginx Cache − реалізація кешування на рівні веб-сервера, який часто застосовується як зворотний проксі для кешування динамічних сторінок.

HTTP Cache-Control − стандартний механізм керування кешуванням через заголовки HTTP-протоколу, що дозволяє задавати політики оновлення контенту на клієнті.

Вибір інструменту залежить від типу архітектури, обсягів даних і вимог до швидкодії. Для монолітів зазвичай достатньо вбудованого або Redis-кешу, а в мікросервісах частіше використовують розподілені системи кешування.

1.2.4 Порівняння кешування у монолітних і мікросервісних системах

У монолітних системах кешування зазвичай централізоване − застосунок має спільний кеш, який використовується всіма його компонентами. Це дозволяє легко контролювати обсяг даних і забезпечувати консистентність, але створює єдину точку відмови. Якщо кеш вийде з ладу, це вплине на всю систему. Також

монолітам властива простіша стратегія інвалідації кешу, оскільки всі компоненти працюють у єдиному процесі.

У мікросервісних системах ситуація складніша. Кожен сервіс може мати власний кеш або спільний доступ до розподіленого кеша. Це дає змогу масштабувати кешування незалежно, але створює проблеми із синхронізацією − наприклад, при зміні даних у сервісі «Користувачі» необхідно оновити кеш у сервісі «Замовлення». Розподілене кешування потребує спеціальних протоколів узгодження або стратегій «cache-aside» та «write-through» (табл. 1.2).

Таблиця 1.2 – Характеристика вебсистем

Характеристика	**Монолітна система**	**Мікросервісна система**
Тип кешу	Централізований	Розподілений або локальний
Інвалідація	Спільна, централізована	Асинхронна, потребує синхронізації
Залежність від архітектури	Тісно інтегрований у додаток	Незалежний компонент
Продуктивність	Висока при невеликому масштабі	Висока при правильному розподілі
Складність реалізації	Нижча	Вища через комунікацію між сервісами

У результаті можна зробити висновок, що оптимальна стратегія кешування має враховувати тип архітектури. Для монолітних систем доцільним є централізований кеш, який мінімізує затримки, тоді як у мікросервісних − гнучкі розподілені рішення, здатні масштабуватися та забезпечувати консистентність даних у розподіленому середовищі.

1.3 Проблематика ефективного кешування

Ефективне кешування відіграє ключову роль у забезпеченні високої продуктивності вебсистем, однак його впровадження пов’язане з низкою технічних і концептуальних проблем. Неправильно обрана стратегія або алгоритм кешування може не лише не покращити роботу системи, а навіть призвести до зворотного ефекту − збільшення затримок, неузгодженості даних або надмірного використання ресурсів пам’яті.

1.3.1 Проблема інвалідації кешу

Інвалідація кешу − це процес видалення або оновлення застарілих даних у кеші. Вважається, що «інвалідація кешу − одна з найскладніших проблем у комп’ютерних науках», адже необхідно знайти баланс між актуальністю даних і швидкістю роботи системи [6].

Якщо дані у кеші зберігаються занадто довго, користувач може отримати застарілу інформацію. Якщо ж кеш очищується занадто часто, це призводить до втрати переваг кешування − збільшується кількість звернень до бази даних і час відповіді системи.

Типові стратегії інвалідації:

– Time-to-Live (TTL): дані зберігаються у кеші протягом заданого часу, після чого видаляються;

– Write-through / Write-behind: дані оновлюються одночасно у кеші та базі даних;

– Cache-aside: кеш оновлюється лише тоді, коли дані запитуються;

– Explicit Invalidation: ручне або подієве очищення кешу при зміні даних у джерелі.

У монолітних системах інвалідацію простіше контролювати − усі компоненти працюють у єдиному середовищі. У мікросервісній архітектурі ситуація складніша: кілька незалежних сервісів можуть використовувати одну й

ту саму інформацію, тому зміни в одному з них потребують синхронного оновлення кешів інших. Без спеціальних механізмів (наприклад, Pub/Sub або Event Bus) це призводить до розсинхронізації.

1.3.2 Проблема узгодженості даних

Ще однією серйозною проблемою є підтримання консистентності між кешем і основним джерелом даних. Ситуація, коли у базі даних уже оновлена інформація, а кеш зберігає стару, може викликати логічні помилки в роботі системи, особливо в умовах розподіленого середовища [6].

Розрізняють кілька типів консистентності:

– сувора (strong consistency): кеш і база даних завжди містять однакові дані;

– кінцева (eventual consistency): оновлення даних у кеші відбувається з певною затримкою;

– кешова (cache consistency): припускається, що невелика розбіжність допустима задля підвищення швидкодії.

У монолітних системах, де всі операції виконуються в одному процесі, підтримувати консистентність відносно просто. У мікросервісах же консистентність потребує спеціальних протоколів комунікації. Наприклад, сервіс, який оновлює дані, має надсилати повідомлення про зміну іншим сервісам, що кешують цю інформацію. Для цього часто використовують механізми message broker або вбудовані механізми Redis − Pub/Sub чи keyspace notifications.

Проблема ускладнюється, якщо система розгорнута у кластері або хмарному середовищі. У таких випадках важливо забезпечити атомарність операцій над кешем і синхронізацію між вузлами, інакше можливі колізії або втрати даних.

1.3.3 Проблема вибору стратегії кешування

Кешування завжди передбачає компроміс між продуктивністю, точністю даних і ресурсами пам’яті. Надмірне кешування призводить до перевитрат оперативної пам’яті, тоді як недостатнє − до частих звернень до джерела даних.

Основні фактори, що впливають на вибір стратегії наступні:

– частота оновлення даних у системі;
– обсяг даних, які зберігаються;
– кількість одночасних користувачів;
– тип архітектури (моноліт чи мікросервіси);
– вимоги до консистентності та доступності.

Найпоширенішими стратегіями кешування є такі:

– кеш заповнюється лише при першому запиті (Cache-aside);
– кеш і база даних оновлюються одночасно (Write-through);
– кеш оновлюється одразу, а база даних з затримкою (Write-behind);
– додаток взаємодіє лише з кешем, який самостійно оновлює дані при необхідності (Read-through).

Для монолітних систем зазвичай використовують cache-aside або read-through, оскільки це просто реалізується в межах одного застосунку. Для мікросервісів більш ефективними вважаються write-behind і write-through, які дозволяють синхронно оновлювати дані у спільному кеші.

Ще один важливий аспект ефективності кешування − знаходження оптимального балансу між швидкодією системи та актуальністю даних. У більшості реальних систем неможливо одночасно забезпечити і миттєвий доступ до даних, і їх абсолютну точність. Наприклад, у соціальних мережах допускається коротка затримка в оновленні кількості лайків чи переглядів, оскільки це не впливає на основну логіку роботи, але дає значне підвищення продуктивності [4].

Рішення про допустимий рівень “застарівання” даних приймається виходячи з критичності бізнес-логіки. Для фінансових транзакцій важлива

сувора консистентність, тоді як для контентних систем пріоритетом є швидкодія. Таким чином, ефективне кешування − це завжди компроміс між точністю та продуктивністю.

1.4 Постановка задачі дослідження

Таким чином, дослідження та порівняння методів кешування монолітних і мікросервісних вебсистем є актуальним завданням сучасної веброзробки. Зі зростанням складності архітектур та обсягів оброблюваних даних постає необхідність у виборі ефективних стратегій кешування, що забезпечують оптимальну швидкодію, узгодженість даних і стабільність роботи системи.

Прийнято рішення щодо розроблення моделі та експериментального порівняння ефективності різних методів кешування для двох типів архітектур: монолітної та мікросервісної. У межах дослідження передбачається застосування кількох підходів до кешування, зокрема локального кешу, розподіленого кешу та кешування на рівні запитів і даних. Крім того, важливо враховувати, що для різних архітектур характерні відмінні вимоги до механізмів зберігання та оновлення даних. Монолітні системи можуть покладатися на локальні кешовані структури, що працюють у межах одного процесу і забезпечують мінімальний час доступу. Натомість мікросервісні системи потребують розподіленого кешу, доступного одночасно з кількох вузлів, що значно ускладнює забезпечення узгодженості та вимагає використання протоколів синхронізації та інвалідації даних. Саме ці відмінності створюють підґрунтя для глибокого порівняння поведінки кешу в обох архітектурах.

Об’єктом дослідження є процес кешування даних у вебсистемах із різними архітектурними підходами.

Предметом дослідження є методи, алгоритми та стратегії організації кешу в монолітних і мікросервісних вебсистемах, а також їхній вплив на продуктивність, час відгуку та узгодженість даних.

Метою дослідження є порівняння ефективності різних методів кешування шляхом моделювання та аналізу їх впливу на продуктивність монолітних і мікросервісних вебсистем.

Для досягнення поставленої мети необхідно вирішити такі завдання:

– провести аналіз наукових та технічних джерел щодо застосування кешування в сучасних вебсистемах;

– дослідити архітектурні особливості монолітних і мікросервісних систем, що впливають на підходи до кешування;

– розглянути сучасні методи кешування (in-memory cache, distributed cache) та визначити їх придатність для різних архітектур;

– побудувати математичні моделі процесу кешування для обох типів систем;

– розробити модельну реалізацію монолітної та мікросервісної вебсистем із використанням обраних методів кешування;

– провести експериментальне порівняння ефективності методів кешування за такими критеріями, як час відгуку, навантаження на базу даних, обсяг використаної пам’яті та частота оновлення даних;

– проаналізувати результати моделювання та зробити висновки щодо доцільності застосування певних методів кешування для кожного типу архітектури.

Очікуваним результатом дослідження є формування рекомендацій щодо вибору оптимальних методів кешування для монолітних і мікросервісних вебсистем, що дозволить підвищити їхню продуктивність і надійність у реальних умовах експлуатації.

2 МАТЕМАТИЧНЕ МОДЕЛЮВАННЯ МЕТОДІВ КЕШУВАННЯ

2.1 Теоретичні основи математичного моделювання кешування.

Сучасні вебсистеми характеризуються високою інтенсивністю обміну даними та великою кількістю одночасних запитів від користувачів [7]. У таких умовах ефективне керування кешем стає одним із найважливіших завдань для забезпечення стабільної продуктивності системи. Щоб кількісно оцінити вплив різних методів кешування, необхідно формалізувати цей процес за допомогою математичних моделе.

Математичне моделювання дозволяє описати поведінку системи у формі аналітичних залежностей [8, 9], що враховують імовірність потрапляння даних у кеш (cache hit), частоту запитів, обсяг доступної пам’яті та інші параметри. Завдяки цьому можна провести порівняння різних стратегій кешування без необхідності безпосереднього впровадження в реальну систему.

2.1.1 Загальні принципи побудови математичних моделей у вебсистемах.

Будь-яка вебсистема може бути представлена як множина запитів $R=\{r_1,r_2,...,r_n\}$, що надходять від користувачів до сервера протягом певного часу TTT. Кожен запит обробляється або безпосередньо з бази даних, або через кеш.

Тоді середній час відповіді системи можна записати як:

$$T_{avg} = P_{hit} \cdot t_{cache} + P_{miss} \cdot t_{db} \,, \tag{2.1}$$

де P_{hit} − ймовірність потрапляння запиту у кеш;

$P_{miss} = 1 - P_{hit}$ − ймовірність промаху кешу (cache miss);

t_{cache} − середній час доступу до даних із кешу;

t_{db} − середній час отримання даних із бази даних без кешу.

Цей вираз показує, що продуктивність системи прямо залежить від імовірності попадання у кеш і часу доступу до нього. У випадку ідеального кешу, коли $P_{hit} \rightarrow 1$, середній час відповіді прагне до t_{cache}, що значно менше t_{db}.

2.1.2 Математичні залежності продуктивності та часу відповіді

Для більш детального аналізу можна врахувати, що ймовірність потрапляння запиту у кеш є функцією часу та обсягу кешу.

У багатьох моделях (наприклад, при застосуванні принципу Парето) приймають, що частота звернення до елементів підкоряється степеневому закону:

$$f_i = i^{\alpha_1},\ \alpha > 0. \tag{2.2}$$

де f_i – частота звернення до і-го елемента;

α – параметр степеневого закону, що характеризує нерівномірність звернень.

Тоді ймовірність потрапляння у кеш при розмірі кешу C можна наближено визначити як:

$$P_{hit}(C) = \frac{\sum_{i=1}^{C} f_i}{\sum_{i=1}^{N} f_i}. \tag{2.3}$$

де $P_{hit}(C)$ – ймовірність попадання запиту у кеш розміром C;

C – місткість кешу;

N – загальна кількість можливих елементів, що можуть бути запитані;

f_i – частота звернення до і-го елемента.

Ця формула показує, що збільшення розміру кешу приводить до зростання P_{hit}, проте зі спадною ефективністю (закон спадної віддачі).

На практиці це означає, що подвоєння обсягу кешу не призводить до подвоєння ефективності.

Крім того, час обробки запиту можна представити як функцію навантаження на систему:

$$T_{sys} = T_{avg} + \beta \cdot L \ , \tag{2.4}$$

де L – коефіцієнт навантаження (кількість одночасних запитів);

β – параметр, що враховує деградацію продуктивності при високому навантаженні.

Таким чином, математична модель дозволяє оцінити, як зміна розміру кешу або кількості користувачів впливає на середній час відповіді системи.

2.1.3 Формалізація процесу кешування як стохастичної системи

Процес кешування можна розглядати як стохастичний процес із дискретним часом, у якому стан системи визначається вмістом кешу в момент часу *t*.
Тоді очікувана ймовірність потрапляння у кеш у момент часу t визначається як математичне сподівання:

$$P_{hit}(t) = E[H(t)]. \tag{2.5}$$

де *S(t)* – множина елементів, що зберігаються в кеші в момент часу t;

X(t) – запитаний елемент у момент часу t;

H(t) – випадкова величина, що дорівнює 1, якщо $X(t) \in S(t)$, і 0, якщо ні;

$E[\cdot]$ – математичне сподівання.

Для динамічного аналізу кешу можна використати ланцюги Маркова, де кожен стан відповідає конкретному набору елементів у кеші. Перехід між

станами відбувається при надходженні нових запитів і видаленні застарілих даних згідно з обраною політикою (LRU, LFU, FIFO тощо).

Для моделі LRU (Least Recently Used) імовірність переходу зі стану S_i у S залежить від того, чи був запитаний елемент у кеші:

$$P(S_i \to S_j)\begin{cases} p_k, & \text{якщо } X_k \in S_i \\ 1 - p_k, & \text{якщо } X_k \notin S_i \end{cases}, \tag{2.6}$$

де p_k – імовірність звернення до елемента X_k.

Таким чином, система переходить між станами з певними ймовірностями, а стаціонарний розподіл дає змогу обчислити середнє значення P_{hit} для даної політики кешування.

2.1.4 Вплив архітектури системи на модель кешування

Особливість математичного моделювання полягає також у врахуванні архітектурних відмінностей між монолітними та мікросервісними системами. У монолітних системах кеш зазвичай є локальним (in-memory cache), тобто усі запити отримують доступ до спільної пам’яті. Для таких систем модель кешування можна вважати централізованою:

$$P_{hit}^{mono}(C) = \frac{\sum_{i=1}^{C} f_i}{\sum_{i=1}^{N} f_i}. \tag{2.7}$$

У мікросервісних системах кожен сервіс може мати власний кеш або використовувати розподілений кеш, що вносить додаткову складність. У цьому випадку ймовірність хіта визначається не лише розміром кешу, а й коефіцієнтом синхронізації між сервісами $\gamma \in [0,1]$:

$$P_{hit}^{micro}(C,\gamma) = \gamma \cdot \frac{\sum_{i=1}^{C} f_i}{\sum_{i=1}^{N} f_i}, \quad (2.8)$$

де γ =1 – відповідає ідеально синхронізованому розподіленому кешу,

γ → 0 – незалежним кешам із низькою ефективністю спільного доступу.

Таким чином, математична модель для мікросервісної системи має додаткові параметри, що враховують комунікаційні затримки, синхронізацію та відмовостійкість сервісів

2.2 Математичне обґрунтування вибраних методів кешування.

Вибір конкретного методу кешування визначає продуктивність вебсистеми, ефективність використання пам’яті та частоту звернень до бази даних. У цьому підпункті розглядаються три найбільш поширені методи: LRU, LFU та TTL. Для кожного з методів розроблена математична модель, яка дозволяє оцінити ймовірність попадання у кеш, середній час відповіді та ефективність у монолітних і мікросервісних системах.

2.2.1 Аналітичні моделі кешу типу LRU, LFU та TTL

LRU – це метод, при якому з кешу видаляється елемент, який найдовше не використовувався. Така стратегія ефективна у випадках, коли запити до даних демонструють тимчасову локальність, тобто недавно використані дані з великою ймовірністю будуть запитані знову.

Математична модель LRU

Позначимо:

C – місткість кешу (кількість елементів);

N – загальна кількість елементів;

f_i – ймовірність звернення до елемента i (частота запитів).

Припустимо, що частоти запитів підпорядковуються степеневому закону:

$$f_i = \left(\sum_{j=1}^{N} \frac{1}{j^{\alpha}}\right) \cdot \frac{1}{i^{\alpha}},\ \alpha > 0\ , \tag{2.10}$$

де f_i – ймовірність звернення до і-го елемента;

N – загальна кількість можливих елементів;

α – параметр степеневого закону, що визначає ступінь нерівномірності розподілу звернень.

Тоді ймовірність попадання запиту у кеш для LRU приблизно визначається сумою частот елементів, що поміщаються у кеш:

$$P_{hit}^{LRU}(C) = \sum_{i=1}^{C} f_i\ . \tag{2.11}$$

Середній час відповіді системи з кешем LRU:

$$T_{avg}^{LRU} = P_{hit}^{LRU} \cdot t_{cache} + \left(1 - P_{hit}^{LRU}\right) \cdot t_{db}\ , \tag{2.12}$$

де t_{cache} – середній час доступу до кешу,

t_{db} – час запиту до бази даних.

Особливості застосування LRU полягають у його природній ефективності для систем, де дані мають тимчасову локальність і часто повторюються у запитах. У мікросервісних архітектурах цей метод також може використовуватися, проте потребує координації між сервісами у випадку розподіленого кешу, щоб уникнути розбіжностей у стані даних. Основним недоліком LRU залишається ускладнена реалізація у великих кешах, оскільки він вимагає додаткових структур даних для коректного відстеження порядку використання елементів.

Інший метод – це LFU, при якому видаляються найменш часто використовувані елементи. Цей підхід підходить для систем, де є стабільно

популярні дані, які часто запитуються незалежно від часу. На відміну від LRU, стратегія LFU орієнтується не на недавність використання, а на частоту звернень, тому забезпечує перевагу для елементів із довготривалою популярністю. Тоді ймовірність попадання у кеш LFU:

$$P_{hit}^{LFU}(C) = \sum_{i \in \text{topC}(f)} f_i \,, \tag{2.13}$$

де $top_C(f)$ − множина C елементів із найвищими частотами запитів.

Середній час відповіді для LFU:

$$T_{avg}^{LFU} = P_{hit}^{LFU} \cdot t_{cache} + \left(1 - P_{hit}^{LFU}\right) \cdot t_{db} \,. \tag{2.14}$$

Особливостями застосування моделі LFU можна виділити такі:

– підходить для систем із стабільним профілем запитів;

– в мікросервісах LFU потребує централізованого або розподіленого обліку частот запитів, що ускладнює реалізацію;

– головний недолік: чутливість до “сезонних” запитів: якщо раптово зростає популярність нового елемента, LFU може не відреагувати миттєво.

TTL − метод, при якому дані зберігаються у кеші певний час, після чого автоматично видаляються або стають недійсними. Ця стратегія дозволяє контролювати свіжість даних, що важливо у динамічних системах.

Ймовірність попадання запиту у кеш з TTL:

$$P_{hit}^{TTL}(i) = 1 - e^{-\lambda_i \tau} \,, \tag{2.15}$$

де τ − час життя елемента у кеші;

λ − інтенсивність запитів до елемента i

Для всієї системи:

$$P_{hit}^{TTL} = \frac{\sum_{i=1}^{N} P_{hit}^{TTL}(i)}{N} \,. \tag{2.16}$$

Середній час відповіді системи з TTL-кешем:

$$T_{avg}^{TTL} = P_{hit}^{TTL} \cdot t_{cache} + \left(1 - P_{hit}^{TTL}\right) \cdot t_{db}. \quad (2.17)$$

Особливості застосування TTL полягають у тому, що цей метод добре підходить для роботи з динамічними даними, коли першочергове значення має їхня актуальність. TTL є значно простішим у реалізації порівняно з LRU чи LFU, оскільки не потребує підтримки додаткових структур для відстеження частоти чи давності звернень. У мікросервісних системах цей підхід особливо зручний, бо кожен сервіс може самостійно визначати час життя закешованих даних.

2.2.2 Порівняння ефективності моделей

Враховуючи моделі LRU, LFU та TTL, можна оцінити їх продуктивність у різних архітектурах (табл 2.1).

Таблиця 2.1 – Порівняння моделей

Методи кешування	**Монолітна система**	**Мікросервісна система**
LRU	Висока ефективність при тимчасовій локальності; проста реалізація	Потребує синхронізації у розподіленому кеші; ефективність зменшується при великій кількості сервісів
LFU	Оптимальний для стабільно популярних даних	Складно реалізувати централізований облік частот; потребує додаткової комунікації між сервісами
TTL	Простий, дозволяє контролювати свіжість даних	Підходить для розподіленого кешу; ефективність залежить від правильно обраного часу життя τ

Жоден метод не є універсально найкращим. Вибір кешу залежить від типу даних, профілю запитів та архітектури системи. У монолітних системах LRU та LFU зазвичай забезпечують вищу продуктивність при стабільних даних, а TTL зручний для контролю свіжості. У мікросервісах TTL та LRU з адаптованою синхронізацією краще підходять для розподіленого кешу.

2.3 Математичне обґрунтування вибраних методів кешування

Моделювання дозволяє оцінити ефективність обраних методів кешування в умовах різних архітектур без необхідності безпосередньої реалізації системи в продуктивному середовищі. У цьому підпункті розглядається побудова моделей для монолітної та мікросервісної архітектур із застосуванням стратегій LRU, LFU та TTL, а також проводиться експериментальне порівняння їх ефективності [8,9].

Монолітна система розглядається як єдиний блок, що обробляє всі запити користувачів. Вона характеризується централізованим кешем пам’яті, доступним для всіх модулів системи.

Середній час відповіді системи для кожного методу кешування обчислюється за формулою:

$$T_{avg} = P_{hit} \cdot t_{cache} + (1 - P_{hit}) \cdot t_{db}\ , \qquad (2.18)$$

де P_{hit} визначається згідно з моделями LRU, LFU та TTL;

N – кількість унікальних елементів даних;

f_i – частота звернень до iii-го елемента;

t_{cache} – середній час отримання даних із кешу;

t_{db} – середній час отримання даних із бази даних.

Процес моделювання можна представити у вигляді наступного алгоритму:

Крок 1. Ініціалізація кешу заданого розміру C.

Крок 2. Генерація послідовності запитів на основі частот f_i.

Крок 3. Обробка запитів відповідно до обраної політики кешування.

Крок 4. Фіксація кількості попадань та промахів кешу.

Крок 5. Обчислення середнього часу відповіді T_{avg} та ймовірності попадання P_{hit}.

У мікросервісній архітектурі кожен сервіс має власний кеш або використовує спільний розподілений кеш (Redis, Memcached). Модель має додатковий параметр синхронізації кешу γ, що відображає рівень узгодженості між сервісами.

Ймовірність попадання у кеш для мікросервісів:

$$P_{hit}^{micro} = \gamma \cdot \sum_{i \in cache} f_i \,, \tag{2.19}$$

де γ – коефіцієнт взаємодії між мікросервісами, що враховує вплив розподілу запитів і спільного використання кешу.

Середній час відповіді:

$$T_{avg}^{micro} = P_{hit}^{micro} \cdot t_{cache} + \left(1 - P_{hit}^{micro}\right) \cdot t_{db} \,. \tag{2.20}$$

Процес моделювання можна представити у вигляді наступної послідовності дій:

Крок 1. Ініціалізація кешу для кожного сервісу;

Крок 2. Генерація запитів з урахуванням розподілу між сервісами;

Крок 3. Обробка запитів відповідно до обраної політики кешування;

Крок 4. Врахування синхронізації через коефіцієнт γ;

Крок 5. Обчислення середнього часу відповіді та ймовірності попадання у кеш для всієї системи.

2.4 Проєктування експериментальних застосунків

2.4.1 Вибір архітектури, технологій та інструментів

Вибір архітектури вебзастосунку є критично важливим етапом проєктування, оскільки від нього безпосередньо залежить ефективність реалізації методів кешування та масштабованість системи [10]. Для проведення експериментального дослідження було прийнято рішення реалізувати дві архітектури: монолітну та мікросервісну. Це дозволяє порівняти вплив централізованого та розподіленого кешування на продуктивність вебсистеми та оцінити ефективність різних стратегій LRU, LFU та TTL у різних умовах експлуатації.

Монолітна архітектура передбачає об’єднання всієї логіки додатку в єдиному програмному продукті. Такий підхід забезпечує простоту реалізації та тестування, оскільки всі компоненти взаємодіють безпосередньо через внутрішні функції та методи. Крім того, централізований кеш у монолітній системі дозволяє швидко отримувати дані, зменшуючи час відповіді системи та підвищуючи ймовірність попадання запитів у кеш. Основним недоліком моноліту є складність масштабування окремих модулів та обмежена відмовостійкість, оскільки збій у одному компоненті може вплинути на всю систему.

Мікросервісна архітектура передбачає розподіл системи на незалежні сервіси, кожен з яких реалізує окрему бізнес-логіку та має власний кеш. Такий підхід забезпечує високу масштабованість та стійкість до відмов, оскільки кожен сервіс можна окремо масштабувати або перезапускати без впливу на інші компоненти. Крім того, мікросервісна архітектура дозволяє застосовувати різні стратегії кешування для різних сервісів, що є важливим для дослідження ефективності LRU, LFU та TTL у розподілених системах. Основною складністю при використанні мікросервісів є необхідність синхронізації кешів між сервісами, що потребує додаткових ресурсів та може знижувати продуктивність при високому навантаженні.

Для реалізації серверної частини було обрано платформу Node.js, яка забезпечує асинхронну обробку запитів та високу продуктивність при одночасній обробці великої кількості клієнтських запитів. Node.js також має широкі можливості для інтеграції з системами кешування, такими як Redis або in-memory кеш, що робить його оптимальним вибором для дослідження методів кешування.

У монолітній системі кеш централізований і розташований безпосередньо на сервері, тоді як у мікросервісній архітектурі кожен сервіс має власний кеш, що синхронізується через розподілений механізм.

Таким чином, реалізація двох архітектур дозволить провести експериментальне порівняння ефективності методів кешування та оцінити їх вплив на продуктивність вебсистеми залежно від обраної архітектури. Обидва підходи забезпечують основу для побудови моделі додатку, яка стане основою для подальшого проєктування бази даних, API та логіки обробки запитів.

При проєктуванні додатку для дослідження ефективності методів кешування важливе значення має правильний вибір технологічного стеку. Вибір технологій визначає не лише можливості реалізації різних стратегій кешування, але й забезпечує продуктивність, масштабованість та гнучкість системи. У ході проєктування було прийнято рішення використати платформу Node.js як серверну частину, базу даних MongoDB або PostgreSQL [11,12], а також систему кешування Redis для розподіленого кешу та in-memory кеш для монолітної архітектури.

Використання Node.js обумовлено його здатністю обробляти велику кількість одночасних запитів завдяки асинхронній моделі виконання та подієвому циклу. Node.js підтримує роботу з популярними бібліотеками для створення REST API, що дозволяє організувати чітку структуру проєкту та забезпечити ефективну маршрутизацію запитів. Крім того, Node.js забезпечує високу продуктивність при роботі з кешем, що є критично важливим для оцінки ефективності стратегій LRU, LFU та TTL.

Для зберігання даних обрано базу даних MongoDB або PostgreSQL, оскільки обидва підходи дозволяють гнучко працювати з великими обсягами інформації та ефективно організовувати доступ до об’єктів, що кешуються. MongoDB забезпечує швидку роботу з документами та підтримує горизонтальне масштабування, що зручно для мікросервісної архітектури. PostgreSQL, у свою чергу, забезпечує надійність та підтримку складних запитів SQL, що може бути корисним у монолітній архітектурі для побудови централізованих звітів та аналітики.

Систему кешування було спроєктовано на основі Redis для мікросервісної архітектури та in-memory кешу для монолітної системи. Redis обрано через його здатність зберігати дані в оперативній пам’яті з високою швидкістю доступу та підтримку політик кешування LRU, LFU і TTL. Використання Redis дозволяє організувати розподілений кеш, синхронізований між сервісами, що особливо важливо для мікросервісної архітектури. In-memory кеш у моноліті дозволяє швидко обробляти запити без необхідності додаткової мережевої комунікації та є оптимальним для централізованого кешування.

Для реалізації клієнтської частини додатку можна використати простий інтерфейс на HTML/JavaScript або тестові засоби, такі як Postman, для відправки запитів до серверного API. Це дозволяє швидко перевіряти роботу стратегій кешування та оцінювати час відповіді системи без потреби розробляти повноцінний фронтенд.

Таким чином, обраний технологічний стек забезпечує можливість реалізації обох архітектур, дозволяє проводити експерименти з різними методами кешування та оцінювати їх ефективність в умовах як монолітної, так і мікросервісної системи. Застосування Node.js, Redis та обраної бази даних дозволяє створити гнучку та масштабовану платформу для моделювання та дослідження кешування в реальних умовах.

2.4.2 Проєктування бази данних та логіки обробки запитів

Проєктування бази даних є ключовим етапом розробки вебдодатку, оскільки від її структури залежить швидкість доступу до даних та ефективність роботи кешу [13]. У рамках даного дослідження база даних спроєктована так, щоб забезпечити зручну інтеграцію з різними стратегіями кешування та можливість порівняння їх ефективності в монолітній та мікросервісній архітектурах.

Вибір типу бази даних визначався характером збережених даних та потребами проєкту. Для монолітної системи доцільно використовувати реляційну базу даних PostgreSQL, що забезпечує структуроване зберігання даних та підтримку складних SQL-запитів. Для мікросервісної архітектури та роботи з документами у форматі JSON використовується MongoDB, яка дозволяє швидко масштабуватися та ефективно працювати з неструктурованими даними. Обидві бази даних підтримують високу продуктивність при роботі із кешованими даними та легко інтегруються з Redis [14].

Структура бази даних передбачає збереження основних об’єктів, що будуть кешуватися. Для кожного об’єкта зберігаються ключові поля: унікальний ідентифікатор, дата створення або оновлення, значення даних та додаткові параметри, необхідні для бізнес-логіки. Важливим аспектом є правильне індексування полів, які часто запитуються, що дозволяє підвищити швидкість отримання даних та збільшити ймовірність попадання у кеш.

Інтеграція кешу з базою даних реалізується через проміжний модуль, який перевіряє наявність даних у кеші перед зверненням до бази. У разі промаху (cache miss) дані отримуються з бази, зберігаються у кеші та повертаються користувачу. Такий підхід дозволяє зменшити навантаження на базу даних і підвищити швидкість обробки запитів.

Для мікросервісної архітектури кожен сервіс може мати власну схему зберігання даних або працювати з окремою базою. Це забезпечує незалежність сервісів та можливість застосовувати різні стратегії кешування залежно від

частоти звернень та обсягу даних. Для синхронізації кешів між сервісами використовується Redis, що дозволяє централізовано контролювати час життя даних (TTL) та політику видалення за LRU або LFU.

У монолітній системі всі запити проходять через один кеш, а у мікросервісній архітектурі кожен сервіс має локальний кеш з можливістю синхронізації через Redis [13].

Таким чином, розроблена структура бази даних забезпечує ефективну роботу як централізованого, так і розподіленого кешу, дозволяє проводити експериментальне порівняння методів кешування та оцінювати їх вплив на продуктивність системи в умовах реальних навантажень.

Проєктування API та логіки обробки запитів є наступним ключовим етапом створення додатку, оскільки від цього залежить швидкість взаємодії клієнта з сервером та ефективність кешування даних. У рамках дослідження було прийнято рішення реалізувати REST API, що дозволяє стандартизовано обробляти запити на отримання, створення, оновлення та видалення даних. Такий підхід забезпечує універсальність додатку, легкість інтеграції з клієнтськими інтерфейсами та можливість тестування за допомогою стандартних засобів, таких як Postman.

Кожен запит спочатку проходить через проміжний шар middleware, який відповідає за роботу з кешем. На цьому етапі перевіряється наявність необхідних даних у кеші. У разі попадання (cache hit) сервер повертає результат без звернення до бази даних, що значно зменшує час відповіді системи та знижує навантаження на базу. Якщо дані відсутні в кеші (cache miss), запит обробляється базою даних, після чого отриманий результат зберігається у кеші відповідно до обраної стратегії (LRU, LFU або TTL).

Алгоритм обробки запиту можна представити наступним чином:

Крок 1. Клієнт надсилає HTTP-запит до API.

Крок 2. Сервер перевіряє наявність даних у кеші.

Крок 3. Якщо дані є (cache hit) → повертається результат клієнту.

Крок 4. Якщо даних немає (cache miss) → виконується запит до бази даних.

Крок 5. Отримані дані зберігаються у кеші згідно з обраною стратегією та повертаються клієнту.

Лістинг 2.1 Middleware кешування:

```
async function cacheMiddleware(req, res, next) {
  const key = req.url;
  const cachedData = await cache.get(key);
  if (cachedData) {
    return res.send(cachedData);
  } else {
    res.sendResponse = res.send;
    res.send = async (body) => {
      await cache.set(key, body, ttl);
      res.sendResponse(body);
    };
    next();
  }
}
```

Цей механізм дозволяє забезпечити прозору роботу кешу для клієнта та серверних модулів, не змінюючи бізнес-логіку обробки даних. Для мікросервісної архітектури та централізованого кешу (Redis) аналогічний middleware може взаємодіяти з Redis, використовуючи ключі, що унікально ідентифікують об’єкти у кожному сервісі, та враховувати час життя даних (TTL).

У монолітній архітектурі кеш розташовується безпосередньо в серверному модулі і використовується одна централізована пам’ять для всіх запитів, що забезпечує високу ефективність при стабільних профілях навантаження. У мікросервісній архітектурі кожен сервіс має власний кеш, а Redis дозволяє підтримувати синхронізацію між ними.

Таким чином, проєктування API та механізму обробки запитів дозволяє організувати ефективну роботу кешу у обох архітектурах, забезпечує масштабованість та гнучкість системи та створює основу для подальшого тестування ефективності стратегій кешування під різним навантаженням.

2.4.3 Проєктування компонентів кешування

Проєктування компонентів кешування є ключовим етапом розробки додатку, оскільки саме від ефективності кешу залежить продуктивність системи та швидкість обробки запитів. У рамках дослідження було прийнято рішення реалізувати три основні стратегії кешування: LRU, LFU та TTL, які дозволяють оцінити ефективність різних підходів у монолітній та мікросервісній архітектурах.

У монолітній системі кеш реалізується як централізований in-memory модуль, що дозволяє швидко отримувати дані без звернення до бази даних. Кеш зберігає останні запити та результати їх обробки, використовуючи обрану стратегію видалення об’єктів. LRU забезпечує видалення найстаріших за часом використання даних, LFU − найменш часто використовуваних, а TTL − даних, які перевищили заданий час життя. Такий підхід дозволяє оптимізувати використання пам’яті та підвищити ймовірність попадання запитів у кеш.

Для мікросервісної архітектури кеш реалізується розподілено за допомогою Redis, що дозволяє кожному сервісу мати власний локальний кеш, а також синхронізувати дані між сервісами. Redis підтримує всі три обрані стратегії кешування та забезпечує високу швидкість доступу до даних. Використання TTL у Redis дозволяє автоматично видаляти застарілі дані, а LRU та LFU керують обсягом кешу при обмеженій пам’яті.

Лістинг 2.2 Приклад реалізації LRU кешування:

```
class LRUCache {
  constructor(limit) {
    this.limit = limit;
    this.cache = new Map();
```

```
    }

    get(key) {
        if (!this.cache.has(key)) return null;
        const value = this.cache.get(key);
        this.cache.delete(key);
        this.cache.set(key, value);
        return value;
    }

    set(key, value) {
        if (this.cache.size >= this.limit) {
            const oldestKey = this.cache.keys().next().value;
            this.cache.delete(oldestKey);
        }
        this.cache.set(key, value);
    }
}
```

Для LFU реалізація передбачає ведення лічильника звернень для кожного ключа та видалення найменш популярного елемента при переповненні кешу. TTL реалізується через додавання поля з часом життя об’єкта і автоматичне видалення даних після закінчення цього часу.

Для наочності та узагальнення основних характеристик кожної стратегії кешування в різних архітектурах, у таблиці 2.2 наведено порівняння особливостей реалізації LRU, LFU та TTL у монолітній та мікросервісній системах.

Таблиця 2.2 – Порівняння стратегій кешування

Стратегія	**Моноліт**	**Мікросервіси (Redis)**
LRU	Централізований, простий	Розподілений, синхронізація між сервісами
LFU	Централізований, облік частоти звернень	Розподілений, складніший облік частоти
TTL	Автоматичне видалення за часом	Автоматичне видалення з Redis, можливість централізованого контролю

Таким чином, проєктування компонентів кешування забезпечує ефективну роботу системи при високому навантаженні, дозволяє оцінити переваги та недоліки різних стратегій, а також формує основу для подальшого моделювання ефективності кешування та експериментальних досліджень у обох архітектурах.

2.4.4 Проєктування логіки застосунку

Проєктування логіки застосунку є важливим етапом розробки, оскільки визначає спосіб організації обробки даних, взаємодії між компонентами та ефективності реалізації стратегій кешування. Для забезпечення масштабованості та простоти подальшого тестування було прийнято рішення побудувати модульну структуру, де кожен модуль відповідає за окремий аспект роботи системи.

Основними модулями застосунку є: API-модуль, модуль кешування, модуль доступу до бази даних та модуль бізнес-логіки.

API-модуль відповідає за прийом та маршрутизацію запитів від клієнта, виклик відповідних функцій бізнес-логіки та повернення результатів користувачу.

Модуль кешування обробляє запити до кешу, реалізує стратегії LRU, LFU та TTL, а також забезпечує взаємодію з Redis у мікросервісній архітектурі або in-memory кешем у моноліті.

Модуль доступу до бази даних реалізує CRUD-операції з базою даних, підтримує індексування полів та забезпечує оптимізацію запитів для швидкого отримання даних.

Модуль бізнес-логіки реалізує алгоритми обробки даних та управління кешем на основі частоти звернень, часу життя даних та політики видалення.

Алгоритм обробки запиту в застосунку можна представити таким чином:

Крок 1. Клієнт надсилає запит до API.

Крок 2. API-модуль передає запит до бізнес-логіки.

Крок 3. Бізнес-логіка перевіряє наявність даних у кеші через модуль кешування.

Крок 4. Якщо дані присутні у кеші, то вони повертаються користувачу.

Крок 5. Якщо даних немає, то виконується запит до бази даних через модуль доступу до даних.

Крок 6. Отримані дані зберігаються у кеші відповідно до обраної стратегії та повертаються клієнту.

Модульна структура дозволяє легко змінювати окремі компоненти системи без впливу на інші частини. Наприклад, можна змінити стратегію кешування або тип бази даних без модифікації бізнес-логіки чи API. Також така структура полегшує тестування продуктивності та ефективності різних стратегій кешування у монолітній та мікросервісній архітектурах.

Таким чином, проєктування логіки застосунку з модульною структурою забезпечує ефективну роботу всіх компонентів, дозволяє гнучко реалізовувати стратегії кешування та створює основу для подальшого тестування продуктивності та порівняння ефективності кешу у різних архітектурах.

2.4.5 Параметри тестування та метрики оцінювання

Тестування вебзастосунку – це не просто формальна частина роботи, а ключовий етап, який дозволяє оцінити, наскільки ефективно працюють розроблені стратегії кешування і чи дійсно обрана архітектура витримує реальні навантаження. Для нашого дослідження важливо не тільки перевірити роботу системи, але й порівняти монолітну та мікросервісну архітектури, щоб зрозуміти, які методи кешування найбільш ефективні у кожному випадку.

Щоб отримати максимально достовірні результати, були визначені три сценарії тестування. Перший сценарій – низьке навантаження, коли система отримує невелику кількість одночасних запитів. Це дозволяє оцінити базову швидкість роботи кешу та час відповіді системи. Другий сценарій – середнє

навантаження, що імітує реальні умови роботи вебзастосунку з регулярною кількістю користувачів. І нарешті, третій сценарій – високе навантаження, яке перевіряє стійкість системи до пікових ситуацій і дозволяє оцінити, наскільки обрана стратегія кешування здатна зменшувати навантаження на базу даних під час пікових звернень.

Для проведення тестів використовуються сучасні інструменти. Postman дозволяє швидко перевірити правильність обробки запитів і функціонування API, а Apache JMeter та Artillery, k6 дають змогу створювати реалістичне навантаження, відстежувати час відповіді та поведінку системи під високим трафіком [17,18]. Комбінація цих інструментів забезпечує комплексний підхід до оцінки продуктивності.

Особливу увагу приділено ключовим метрикам, які демонструють ефективність кешування. Серед них – середній час відповіді сервера, який показує, наскільки швидко користувач отримує дані; ймовірність попадання у кеш, що дозволяє оцінити, наскільки часто дані доставляються безпосередньо з кешу, зменшуючи навантаження на базу; та навантаження на базу даних, що відображає кількість запитів, які обробляються безпосередньо базою, а не через кеш.

Процес тестування організовано у вигляді покрокового алгоритму. Спершу ініціалізується сервер та налаштовується кеш з обраною стратегією і розміром. Далі створюється навантаження відповідно до сценарію. Після цього збираються дані про час відповіді, ймовірність попадання у кеш та навантаження на базу. Після завершення одного сценарію тест повторюється для іншого навантаження та змін параметрів кешу. Результати систематизуються та візуалізуються у вигляді графіків і таблиць, що дозволяє наочно порівняти ефективність різних стратегій.

Таким чином, підготовка до тестування не обмежується простою перевіркою роботи застосунку. Це системний підхід, що дозволяє оцінити реальну ефективність кешування, порівняти архітектури та отримати дані для наукового аналізу та практичних рекомендацій щодо оптимізації вебсистем.

3 РЕАЛІЗАЦІЯ ТА ЕКСПЕРИМЕНТАЛЬНЕ ДОСЛІДЖЕННЯ СИСТЕМИ КЕШУВАННЯ У МОНОЛІТНІЙ ТА МІКРОСЕРВІСНІЙ АРХІТЕКТУРАХ

3.1 Вибір інструментальних засобів для реалізації систем

Створення двох експериментальних застосунків – монолітного та мікросервісного – вимагало не лише інженерного підходу, а й уважного добору інструментів. Адже кожен з них формує стиль розробки, визначає можливості системи й навіть впливає на точність результатів дослідження. У сучасному середовищі веброзробки вибір технологій уже давно перестав бути другорядним питанням: це фактично частина самої архітектури.

Саме тому ключові рішення щодо стеку ґрунтувалися на поєднанні трьох факторів:

– практична придатність,

– можливість масштабування та модульності,

– придатність для експериментів із кешуванням і вимірюванням швидкодії.

У результаті було сформовано набір інструментів – NestJS, PostgreSQL, Redis, Docker і k6. Кожен із них виконує свою роль у структурі дослідження, і разом вони забезпечують цілісність та відтворюваність експерименту.

Для побудови обох застосунків було важливо використати один і той самий технологічний підхід. NestJS виявився оптимальним варіантом завдяки своїй модульній структурі. Він дозволяє однаково органічно реалізувати: монолітну аріхітектуру, або мікросервіси.

Архітектурні принципи NestJS – контролери, модулі, сервіси – роблять код передбачуваним і легко масштабованим. Крім того, вбудована підтримка TypeScript значно знижує кількість помилок на етапі розроблення, що особливо важливо під час побудови великої експериментальної бази.

PostgreSQL обрано як основну систему зберігання даних. Причини доволі прості: вона поєднує надійність реляційної моделі та достатню гнучкість для

роботи зі структурованими й напівструктурованими даними. Підтримка jsonb, можливість створення GIN-індексів, коректна робота з транзакціями та масштабними наборами даних роблять цю СУБД цілком придатною для реального вебзастосунку. Для дослідження кешування PostgreSQL підходить ще й тому, що вона не є надто швидкою, щоб “прикрити” користь від кешу, але й не настільки повільна, щоб штучно погіршувати результати. Завдяки цьому порівняння ефекту кешу в монолітній і мікросервісній архітектурах стало більш показовим.

Redis – це одна з найпотужніших систем кешування, яка працює в оперативній пам’яті, забезпечуючи час відгуку на рівні мікросекунд. В застосунках Redis використовується для двох різних сценаріїв: кеш “всередині” моноліту, кешування, до якого можуть звертатися кілька мікросервісів. Завдяки підтримці TTL, структур даних, високій швидкості та можливості працювати в кластерному режимі Redis дозволяє отримати максимально показові результати при високому навантаженні.

Для коректності дослідження було критично важливо забезпечити однакові умови запуску системи. Docker фактично усунув різницю між середовищами, адже всі компоненти – застосунки, Redis, PostgreSQL – працюють у контейнерах з чітко визначеними конфігураціями.

Це дозволило зменшити вплив зовнішніх факторів, зокрема налаштувань операційної системи, а також значно спростило запуск, перенесення та повторення експериментів. Крім того, використання контейнеризації забезпечило стабільність середовища, що дало змогу отримувати зіставні та відтворювані результати навантажувального тестування.

Щоб оцінити ефективність кешування, необхідно виміряти продуктивність системи під навантаженням. Саме тому для тестування застосовано k6 – легкий, але потужний інструмент для performance-тестів. Його головні переваги:

– створювати реалістичні сценарії навантаження,
– генерувати десятки тисяч запитів на секунду,
– збирати метрики латентності, пропускної здатності, RPS,

– точно відстежувати ефект кешу.

Його сценарії – це просто JavaScript-файли, що дає змогу швидко змінювати конфігурації та повторювати вимірювання стільки разів, скільки потрібно.

3.2. Опис концепції застосунків

Побудова двох експериментальних систем – монолітної та мікросервісної вимагала значно більше, ніж просто перенесення однакової бізнес-логіки в різні архітектури [19, 20]. Основним викликом було створити такі умови, за яких ці архітектури можна буде порівнювати об’єктивно, без викривлення результатів сторонніми чинниками. Тому концепція застосунку розроблялася як своєрідний “полігон”, де можна ізольовано спостерігати поведінку кешування, навантаження, затримок та пропускної здатності системи.

Саме тому концепція застосунку була розширена до трьох повноцінних модулів:

- Product Module, що відповідає за роботу з товарами, пошуком, кешуванням;
- User Module, що представляє собою облік користувачів, базові операції над їхніми профілями;
- Cart Module для створення та управління корзинами, додавання/видалення товарів, валідація доступності, інтеграція з Product Service.

Саме така структура дозволяє відтворити сценарії, характерні для реальних вебзастосунків – від простих операцій вибірки конкретної сутності до повнотекстового пошуку й комбінованих фільтрів високої складності. Подібні операції є типовими для інтернет-магазинів, каталогів, інформаційних порталів, і якраз вони формують значну частину навантаження у практичних системах.

Ці модулі формують типовий мінімальний функціонал e-commerce-системи, що ідеально підходить для тестування кешу, навантаження та взаємодії сервісів.

Для дослідження було створено дві окремі, але функціонально тотожні версії застосунку. Перший застосунок є монолітним, у якому усі модулі, такі як робота з товарами, користувачами та корзиною, інтегровані в межах одного NestJS-додатку. Уся бізнес-логіка виконується в одному процесі, на одному сервері та через один порт, що забезпечує максимально тісну зв’язність компонентів і передбачувану взаємодію між ними (рис. 3.1).

Рисунок 3.1 – UML-діаграма монолітного застосунку

Мікросервісний застосунок, де логіка поділена на окремі сервіси: product service, user service, cart service, які взаємодіють між собою через HTTP та Redis (рис 3.2).

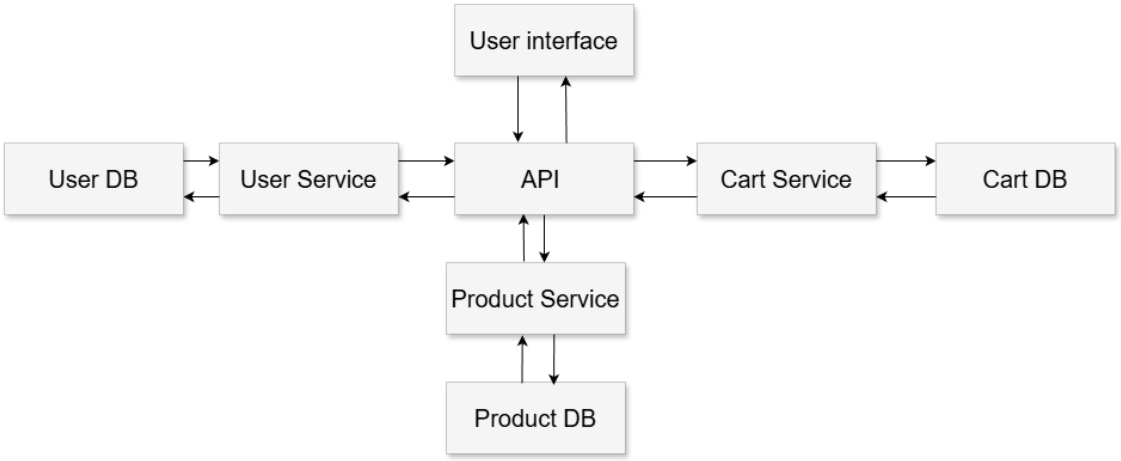

Рисунок 3.2 - UML-діаграма мікросервісного застосунку

Таке розділення відтворює типовий сценарій роботи розподіленої системи, у якому Cart Service звертається до Product Service для отримання характеристик товару, паралельно перевіряє користувача через User Service, зберігає власний стан у окремій базі даних і використовує Redis для кешування найбільш частих операцій. Таким чином формується ланцюжок взаємодій, характерний для мікросервісної архітектури, де кожен компонент виконує строго окреслену роль, але водночас залежить від коректної роботи інших сервісів.

Зовні ці системи працюють однаково: вони мають ідентичні маршрутні точки, однакові тести навантаження та однакові інтерфейси взаємодії. Проте внутрішня організація в них принципово відрізняється. Моноліт – це єдиний цілісний механізм, де всі компоненти живуть у межах одного процесу, що гарантує мінімальні внутрішні затримки та просту модель виконання. Натомість мікросервіси вибудовують децентралізовану систему, де кожен компонент потенційно може працювати на іншому сервері чи навіть у іншому дата-центрі.

Це дозволяє спостерігати, як архітектура впливає на час відповіді, поведінку під навантаженням, ефективність кешування, стабільність системи та мережеві накладні витрати.

Для чесного порівняння функціонал монолітної та мікросервісної систем було зроблено структурно однаковим. Кожен із трьох модулів реалізує однакові API-можливості. Зокрема, Product Module підтримує отримання товару за

ідентифікатором, виконання повнотекстового пошуку, фільтрацію за категоріями, тегами та характеристиками, а також кешування результатів як пошуку, так і вибірки товарів за ID.

User Module дозволяє створювати нового користувача, отримувати його профіль, оновлювати інформацію та виконувати базову авторизацію на рівні сервісу. Також додатково впроваджено кешування профілів, що часто запитуються.

Cart Module забезпечує створення корзини, додавання й видалення товарів, автоматичну синхронізацію стану з Product Service, а також прив'язку корзини до конкретного користувача через User Service. Окрім цього, кешується вміст корзини для пришвидшення повторних операцій.

Особливо цікавим є той факт, що Cart Service ніколи не зберігає інформацію про товари локально – він завжди звертається до Product Service, що створює мережеву залежність. Це дозволяє ізолювати архітектурні ефекти від логіки самої корзини.

У моноліті всі три модулі працюють у межах одного процесу без мережевої взаємодії, без проксі рівня API Gateway та з моментальним доступом до Redis і PostgreSQL. Взаємодія між модулями відбувається через внутрішні сервіси NestJS, що забезпечує мінімальні затримки. Так моноліт виступає максимально швидкою і стабільною системою, у якій ефект кешування проявляється найповніше. У такій структурі кешування є максимально «близьким» до застосунку, тому час доступу мінімальний. Моноліт працює передбачувано, стабільно і без зайвих мережевих накладних витрат.

Мікросервісна версія створює складнішу, але реалістичнішу архітектуру:

- приймає запити, виконує авторизацію, маршрутизує на сервіси (API Gateway).
- працює з товарами, виконує кешування пошуку (Product Service).
- відповідає за профілі користувачів (User Service).
- активно звертається до двох інших сервісів (Cart Service).
- спільний кеш і канал повідомлень (Redis).

– кожен сервіс має власну БД або власну схему (PostgreSQL).

Таким чином, один запит у Cart Service може породжувати 2–3 внутрішні звернення, Product Service може виконувати крос-кешування, User Service забезпечує дані, без яких Cart Service не працює.

Головна ідея мікросервісної версії – створити реальні умови, де кожен запит проходить через мережу, охоплює декілька незалежних компонентів і може бути масштабований горизонтально.

Щоб забезпечити об’єктивність дослідження, обидві системи були побудовані таким чином, аби відрізнялися лише архітектурним підходом, а не логікою роботи чи використовуваними інструментами. Монолітна та мікросервісна версії застосунку використовують однакові API-маршрути, працюють на тому самому технологічному стеку та оперують ідентичним набором даних. Для них визначено однакові сценарії навантаження, встановлено однакові параметри TTL для кешування, а весь процес тестування проводиться у середовищі Docker, що забезпечує повну відтворюваність умов. Таким чином, у дослідженні змінною є лише архітектура системи, тоді як бізнес-логіка та технологічна база залишаються незмінними.

3.3 Програмна реалізація мікросервісного застосунку

Програмна реалізація експериментальної системи охоплює три окремі мікросервіси, API Gateway та допоміжні інструменти, які забезпечують кероване середовище роботи. Хоча монолітна реалізація повторює ті самі функції, саме мікросервісний варіант демонструє повну структуру взаємодії, властиву розподіленим вебсистемам.

Мікросервісна архітектура складається з таких компонентів:

– центральна точка входу, що приймає HTTP-запити та перенаправляє їх до відповідних сервісів (API Gateway).

– робота з товарами, пошук, кешування, генерація тестових записів (Product Service).

– управління профілями користувачів (User Service).

– створення й ведення корзин, інтеграція із сервісами користувачів і товарів (Cart Service).

– спільний кеш і система швидкого доступу до ключових даних (Redis).

– окремі бази даних для кожного сервісу (PostgresSQL).

Усі компоненти розгортаються в Docker, що гарантує повну повторюваність середовища.

Найбільш навантажений сервіс – це Product Service, оскільки саме він оперує великим масивом даних (50 000 товарів). Його функції наступні:

– повнотекстовий пошук;

– фільтрація за характеристиками;

– кешування результатів;

– повернення товару за ID;

– генерація тестових записів у базу даних.

– PostgreSQL для зберігання даних про товари;

– TypeORM та QueryBuilder для виконання складних запитів;

– Redis для кешування запитів за ключем «product:{id}» та пошукових комбінацій;

Особливістю реалізації є велика кількість текстових полів: поле longDescription довжиною до 2000 символів і метадані формату JSONB. Це імітує реальну модель e-commerce-каталогу. Пошук виконується з використанням SQL-конструкцій: ILIKE для нечітких текстових збігів, «metadata->>'color'» для фільтрації JSONB, ANY для відповідності тегів.

Саме цей сервіс найбільше виграє від кешування, що і стало ключовою частиною експерименту.

User Service був доданий для створення більш реалістичної системи, оскільки більшість вебзастосунків вимагають роботи з користувачами. У межах експерименту він реалізує: створення користувача, отримання профілю,

оновлення даних, кешування часто запитуваних профілів, елементарну авторизацію.

Хоча User Service не створює великого навантаження на базу даних, він є критичним для Cart Service, який перевіряє за наступним алгоритмом:

Крок 1. Чи існує користувач.

Крок 2. Чи має він право на виконання операції.

Крок 3. Кому належить корзина.

Таким чином User Service вводить у систему залежності між мікросервісами, що дозволяє оцінити вплив мережевої взаємодії на продуктивність.

Cart Service – це найбільш інтеграційно складний сервіс. На відміну від інших, він зберігає мінімум власних даних: id корзини, id користувача список товарів з кількістю.

Для кожного товару він запитує актуальну інформацію у Product Service. Відповідно, будь-який запит до корзини може породжувати одразу декілька мережевих звернень.

Його основні функції такі:

– створення корзини для користувача;
– додавання/видалення товарів;
– перевірка наявності товарів (через Product Service);
– перевірка користувача (через User Service);
– кешування вмісту корзини для зменшення навантаження.

Саме Cart Service демонструє ефект «ланцюгової затримки», характерної для мікросервісів: один клієнтський запит може спричинити 2–5 внутрішніх. Сервіси взаємодіють через HTTP-запити у межах Docker-мережі:

– «Cart Service → User Service: GET /users/{id}»
– «Cart Service → Product Service: GET /products/{id}»
– «API Gateway → Product Service: GET /products/search»

Так реалізовано синхронну взаємодію. У реальних умовах це імітує класичний REST-підхід до мікросервісів.

3.4 Програмна реалізація монолітного застосунку

Монолітна версія застосунку була створена як базова точка порівняння з мікросервісною архітектурою. Її структура значно простіша, але саме це дає змогу оцінити вплив розподіленості та міжсервісної взаємодії на продуктивність. Моноліт реалізовано на тих самих технологіях, що і мікросервіси – NestJS, PostgreSQL, Redis – однак усі компоненти об’єднані в одну програму, яка працює у межах єдиного процесу.

Мета такої реалізації – створити максимально оптимізовану систему без мережевих затримок, яка використовує кеш «локально» та виконує запити напряму до бази даних. Саме моноліт виступає умовно «ідеальним» варіантом з точки зору продуктивності, і тому є зручним еталоном для подальших вимірювань.

Монолітна система складається з таких чотирьох модулів:

- робота з товарами (Product Module);
- управління користувачами (User Module);
- логіка корзини (Cart Module);
- єдине підключення до Redis для всього застосунку (Cache Module).

Лістинг 3.2 Реалізація підключення in-memory або розподільного Redis кешу:

```
CacheModule.registerAsync({
    isGlobal: true,
    inject: [ConfigService],
    useFactory: async (config: ConfigService) => {

      const mode = config.get<string>('CACHE_MODE', 'none');
      const ttl = config.get<number>('CACHE_TTL', 10);

      if (mode === 'redis') {
```

```
        const host = config.get<string>('REDIS_HOST', 'localhost');
        const port = config.get<number>('REDIS_PORT', 6379);

        return {
          ttl,
          store: await redisStore({
            socket: {
              host,
              port,
            },
          }),
        };
      }
      if (mode === 'memory') {
        return {
          ttl,
          max: 1000,
        };
      }
      return {
        ttl: 0,
        max: 0,
      };
    },
  }),
```

Усі модулі працюють усередині одного середовища, використовують спільні інстанси сервісів, мають доступ до загального контейнера залежностей NestJS та не взаємодіють один з одним через мережу.

Моноліт реалізовано за стандартною схемою NestJS: «Controllers → Services → Repositories». Таке структурування дозволяє зберігати логіку окремо, однак відсутні будь-які фізичні межі між підсистемами – усі звернення відбуваються у пам’яті.

Функціональність модуля товарів повністю повторює версію з мікросервісів, а саме:

- пошук товарів;
- фільтрація;
- отримання товару за ID;
- кешування результатів;
- генерація тестових даних.

Проте є кілька важливих відмінностей, кеш in-memory працює без мережевих витрат. Застосунок і Redis взаємодіють у межах одного процесу, тому час доступу мінімальний. Відсутні додаткові перевірки, логіка маршрутизації та перетворення форматів, які неминучі у мікросервісній системі. Виклик сервісних методів відбувається безпосередньо, без HTTP-обгортки та HttpService.

Усі запити до БД виконуються через TypeORM, а Redis використовується як глобальний кеш для всього застосунку.

User Module у моноліті відповідає за створення та зміну профілів користувачів, а також виконує базові функції автентифікації й авторизації. Крім того, він забезпечує кешування часто запитуваних профілів, що дозволяє зменшити кількість звернень до бази даних і прискорити роботу системи в умовах підвищеного навантаження.

Оскільки моноліт є єдиним додатком, взаємодія між модулями відбувається напряму.

Лістинг 3.3 Приклад використання UserService в CartService:

```
export class CartService {
  constructor(
    @InjectRepository(CartItem)
```

```
    private readonly repo: Repository<CartItem>,
    private readonly usersService: UsersService,
    private readonly productsService: ProductsService,
  ) {}

  async addToCart(userId: number, productId: number, quantity = 1) {
    await this.usersService.findOne(userId);
```

Без мережевих запитів, без черг повідомлень, без логіки міжсервісних контрактів. Це дозволяє досягти максимальної швидкодії, але водночас робить архітектуру менш гнучкою з точки зору масштабування.

Cart Module у моноліті повторює логіку мікросервіса, а саме :

- створення корзин,
- додавання товарів,
- взаємодія з товарами та користувачами.

Проте оскільки в моноліті усі дані доступні безпосередньо через пам’ять, доступ до сервісу товарів виглядає так:

Лістинг 3.4 Використання ProductService у CartService:

```
const product = this.productService.findOne(id)
```

У мікросервісах цей самий виклик перетворюється на HTTP-запит, що може мати затримку в десятки мілісекунд. Таким чином моноліт демонструє «ідеальні умови», у яких логіка корзини працює максимально швидко.

Завдяки DI-контейнеру NestJS усі сервіси взаємодіють напряму, що суттєво зменшує затримку виконання операцій: «Cart → Product», «Cart → User», «Product → Cache», «User → Cache». Це дозволяє не створювати жодних повторних серіалізацій та десеріалізацій JSON.

Це принципова перевага моноліту, яку видно на всіх етапах тестування.

Завдяки єдиному контейнеру залежностей уся система працює з одним екземляром кешу.

Затримки доступу мінімальні, а продуктивність – максимально передбачувана. Логіка TTL, ключів та форматів повністю повторює мікросервісну реалізацію, що дозволяє напряму порівнювати результати. У монолітній реалізації використовується один екземпляр PostgreSQL і одна схема. Операції виконуються без додаткових накладних витрат, характерних для розподілених систем.

Лістинг 3.5 Підключення TypeORM в моноліті:

```
TypeOrmModule.forRootAsync({
    inject: [ConfigService],
    useFactory: (config: ConfigService) => ({
      type: 'postgres',
      host: config.get<string>('DB_HOST'),
      port: config.get<number>('DB_PORT'),
      username: config.get<string>('DB_USER'),
      password: config.get<string>('DB_PASS'),
      database: config.get<string>('DB_NAME'),
      entities: [Product],
      synchronize: true,
    }),
  }),
```

Таке рішення особливо важливе для коректного порівняння монолітної та мікросервісної реалізацій. Одна схема, одна база та одна конфігурація підключення дозволяють виключити вплив додаткових змінних і сконцентруватися саме на різниці архітектурних підходів. Порівняння розроблених застосунків наведено у таблиці 3.1, де відображено ключові

технічні характеристики обох реалізацій і їхній вплив на продуктивність та складність системи

Таблиця 3.1 – порівняння розроблених додатків

Аспект	**Моноліт**	**Мікросервіси**
Взаємодія	внутрішні виклики у межах процесу	HTTP-запити між сервісами
Кеш	спільний Redis, in-memory	спільний Redis, але мережевий доступ
Логіка	єдиний застосунок	розподілена між сервісами
Дані користувачів, товарів і корзин	спільна база	окремі бази
Продуктивність	максимальна	нижча через мережеві затримки
Масштабування	вертикальне	горизонтальне
Модульність	логічна	фізична

3.5 Тестування розроблених застосунків та аналіз результатів

Після завершення програмної реалізації монолітного та мікросервісного застосунків основним завданням стала перевірка їхньої ефективності в умовах різного навантаження. Для цього необхідно не просто виміряти швидкість обробки окремих запитів, а й оцінити поведінку системи при одночасній роботі сотень і тисяч клієнтів, взаємодію сервісів між собою, а також вплив кешу на загальну продуктивність.

Тестування проводилося у контрольованому середовищі, побудованому за допомогою Docker, що гарантувало відтворюваність результатів і виключало

вплив сторонніх факторів, пов’язаних зі специфікою операційної системи або конфігурацією обладнання.

Основним інструментом навантажувального тестування виступив k6, який дозволив змоделювати реальні сценарії поведінки користувачів: від вибірки окремих товарів до складних пошукових запитів із фільтрами.

Аби порівняння було коректним, обидві версії застосунку – монолітна та мікросервісна – запускалися в ідентичних умовах:

– однакова база даних з 50 000 товарів;
– однакові маршрути API;
– однакове кешування в Redis;
– однакові параметри Docker-контейнерів;
– однакова кількість одночасних віртуальних користувачів (VUs).

Перший сценарій отримання одного продукту за id. Для тестування був написан скрипт за допомоги бібліотеки k6.

Лістинг 3.5 k6-скрипт для отримання одного продукту за id:

```
import http from "k6/http";
import { check, sleep } from "k6";
export const options = {
  vus: 1000,
  duration: "30s",
};
const MAX_ID = 50000;
function randomId() {
  return Math.floor(Math.random() * MAX_ID);
}
export default function () {
  const id = randomId();
  const url = `http://localhost:3000/products/${id}`;
```

```
  const res = http.get(url);

  check(res, {
    "status is 200": (r) => r.status === 200,
  });
  sleep(1);
}
```

Конфігурацією цього скрипта є запуск 1000 віртуальних користувачів, що еквівалентно приблизно 1000 запитам на секунду. Тривалість навантаження становить 30 секунд, що дозволяє оцінити середню пропускну здатність системи та відстежити поведінку застосунку під постійним тиском, коли кеш ще порожній.

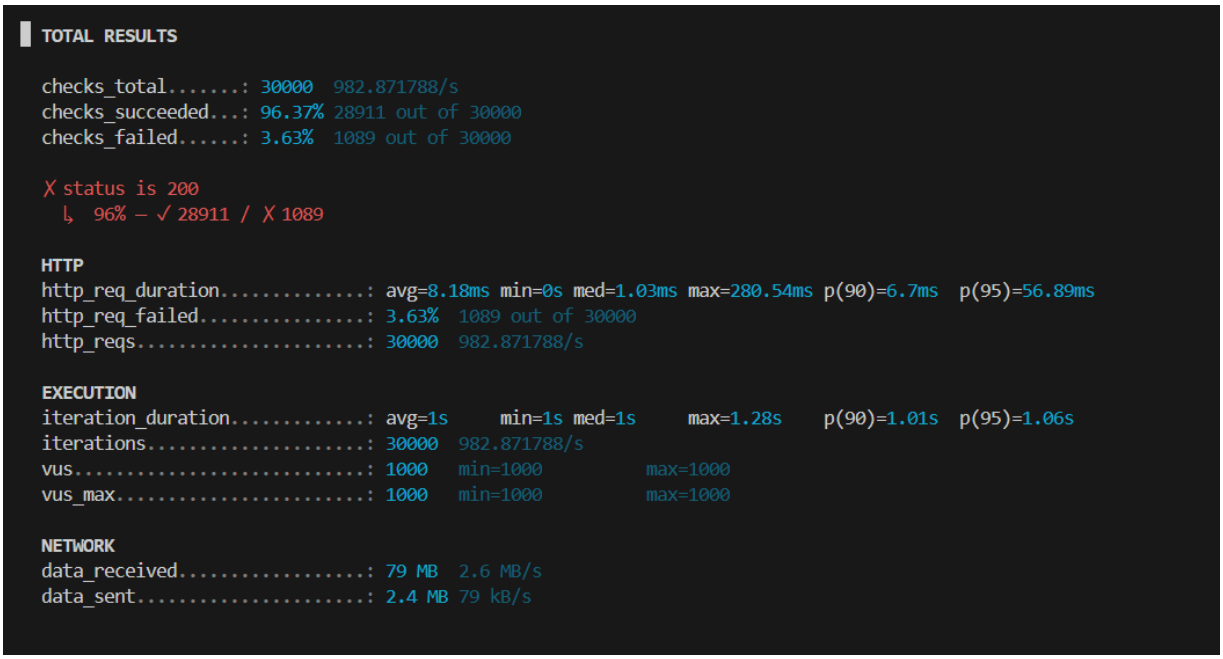

Рисунок 3.3 – Результат виконання тестування без кешу

Тепер підключимо in-memory кеш до нашого застосунку і виконаємо повторне тестування, щоб оцінити, наскільки сильно зміниться продуктивність.

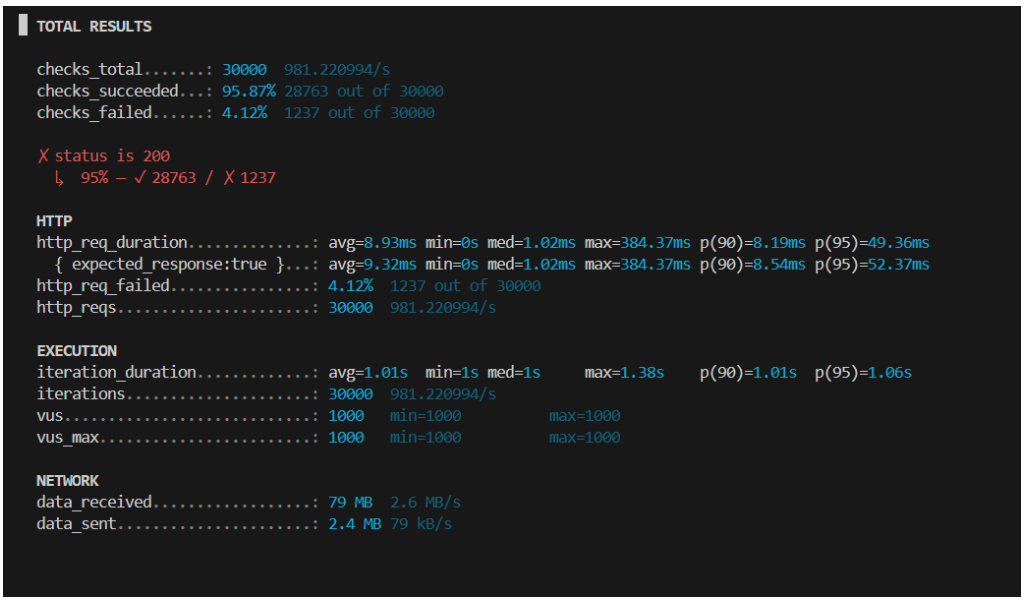

Рисунок 3.4 – Результат виконання тестування in-memory кешу перший раз

Результати виявились очікуваними, бо при першому запуску кеш у застосунку пустий. Виконаємо, ще одне тестування, коли в кеші вже з’явилися записи.

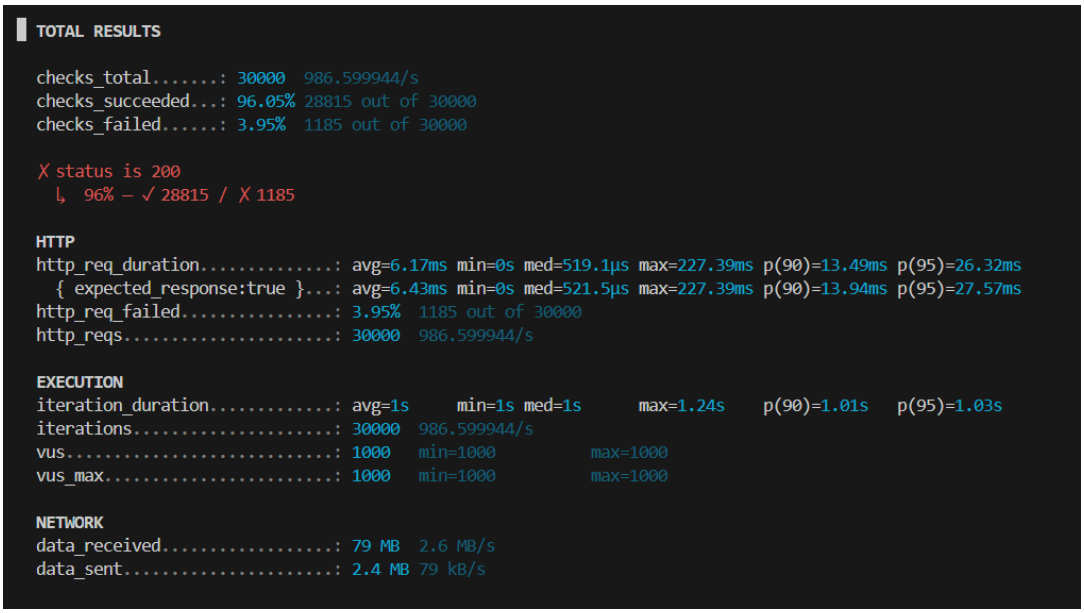

Рисунок 3.5 – Результат виконання тестування in-memory кешу

Як можна побачити з рисунку 3.3 та рисунку 3.5 середній час відповіді зменшився з ~8 мілесекудн до ~6мс. Різниця сягнула ~30%. А медіанний час відповіді знизився з 1 мілісекунди до 500 мікросекунд, різниця в два рази. Що є гарним результатом, але не показовим, оскільки сама операція є дуже простою: знайти об’єкт

в БД та повернути його. Таж сама ситуація з використанням розподіленного кешу Redis (рис.3.6).

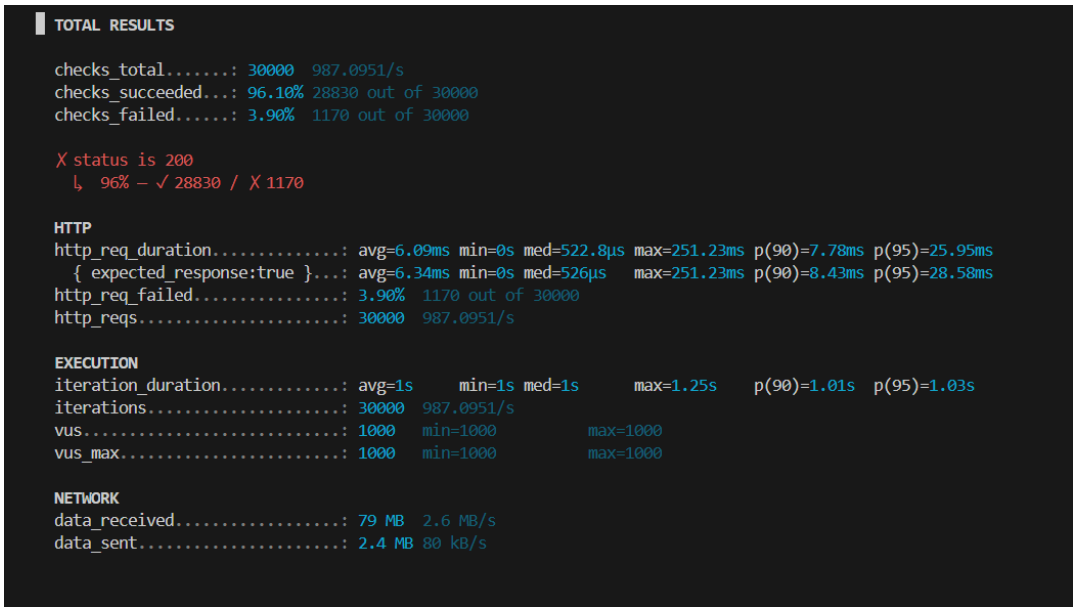

Рисунок 3.6 – Результат виконання тестування in-memory кешу

Другий сценарій отримання одного продукту за id. Для тестування був написан скрипт за допомоги бібліотеки k6.

Лістинг 3.6 k6-скрипт для навантаження пошуку продуктів з пошуковими параметрами:

```
import http from "k6/http";
import { check, sleep } from "k6";

export const options = {
  vus: 1000,
  duration: "30s",
};

const categories = ["electronics", "books", "clothes"];
const colors = ["red", "green", "blue"];
```

```
const sizes = ["S", "M", "L", "XL"];
const tags = ["tag1", "tag2", "tag3", "tag4", "tag5"];

function randomItem(arr) {
  return arr[Math.floor(Math.random() * arr.length)];
}
export default function () {
  const category = randomItem(categories);
  const color = randomItem(colors);
  const size = randomItem(sizes);
  const tag = randomItem(tags);

  const url =
`http://localhost:3000/products/search?category=${category}&color=${color}&size
=${size}&tag=${tag}`;

  const res = http.get(url);

  check(res, { "status is 200": (r) => r.status === 200 });

  sleep(1);
}
```

Цей скрипт отримує випадкові пошукові параметри та передає їх до URL. Це тестування буде найбільш показовим для двох систем. Це складні запити з багатьма параметрами: ключове слово, тег, категорія, колір, розмір тощо. Вони значно навантажують базу даних, оскільки містять повнотекстові операції та фільтрацію по jsonb.

Таблиця 3.2 – Результати тестування моноліту

Тип кешування	Середній час відповіді	Мінімальний Час відповіді	Медіанний час відповіді	Максимальний час відповіді
Без кешування	~3.5с	~165мс	~3.4с	~4.5с
Внутрішній	~111мс	0мс	64мс	~4.5с
Redis	~200мс	0мс	110мс	~4.5с

Як видно з наведених результатів, різні типи кешування суттєво впливають на середній та медіанний час відповіді системи, однак максимальний час практично не зазнає змін. Це пояснюється тим, що впродовж навантажувального тестування частина запитів залишалася унікальною, тобто такою, яка не повторювалася достатню кількість разів для потрапляння до кешу. Для таких запитів система, незалежно від режиму кешування, була змушена виконувати повний шлях обробки.

Для мікросервісів було проведено аналогічні тести зі схожим навантаженням і тими ж сценаріями. Водночас сама архітектура наклала додаткові затримки тому що, запит проходить через API-Gateway, сервіс товарів виконує обробку окремо, Redis працює як зовнішній кеш, до якого також звертаються по мережі. У результаті отримані такі показники (табл 3.3).

Таблиця 3.3 – Результати тестування мікросервісів

Тип кешування	Середній час відповіді	Мінімальний Час відповіді	Медіанний час відповіді	Максимальний час відповіді
Без кешування	~5.8с	~350мс	~5.4с	~7с
Redis	~380-420мс	~2-5мс	~210мс	~5.5с

In-memory кеш у мікросервісній архітектурі не застосовувався, оскільки кожен сервіс є окремим процесом. Локальний кеш не має сенсу – він не узгоджується між екземплярами сервісів.

3.6 Порівняльний аналіз

На основі отриманих результатів тестування можна виокремити кілька закономірностей, які чітко демонструють відмінності між двома підходами. Передусім варто зазначити, що мікросервісна архітектура виявилася суттєво повільнішою у режимі роботи без кешу. Якщо монолітна система виконувала складний пошуковий запит у середньому за 3,5 секунди, то мікросервісна – за понад 5,5–6 секунд. Такий розрив пояснюється насамперед структурною природою мікросервісів: кожен запит проходить через низку мережевих переходів – від API-gateway до сервісу товарів, далі до Redis, і знову назад – що збільшує сукупний час обробки. Додатковим фактором виступає повторна серіалізація та десеріалізація великих JSON-структур, необхідна на кожному кроці маршруту.

Використання Redis суттєво покращує продуктивність обох архітектур, однак моноліт демонструє значно кращу динаміку. У монолітній системі середній час відповіді становить близько 200 мілісекунд, тоді як у мікросервісній від 380 до 420 мілісекунд, тобто приблизно удвічі більше. Це природний наслідок того, що моноліт звертається до Redis лише один раз у межах локального запиту, а мікросервіси виконують кілька послідовних зовнішніх викликів.

Порівняння значень медіани підтверджує стабільнішу роботу моноліту в умовах «теплого» кешу. Для монолітної архітектури медіанні значення становлять 64 мілісекунди при використанні in-memory кешу та близько 110 мілісекунд при роботі через Redis. У мікросервісів ці показники значно вищі – приблизно 210 мілісекунд навіть за наявності заповненого кешу. Отже, сама

архітектура мікросервісів природно привносить додаткову латентність, яку неможливо усунути простими оптимізаціями кешування.

Особливо помітна перевага моноліту полягає у можливості використання in-memory кешу. Для нього середній час відповіді становив приблизно 111 мілісекунд, що фактично гарантує майже миттєву реакцію системи. У мікросервісній архітектурі таке рішення непридатне, оскільки кожен сервіс є окремим процесом, і локальний кеш не може бути узгодженим між різними екземлярами сервісів. Це створює природну нерівність умов, однак водночас наочно демонструє, наскільки ефективним може бути кешування у межах єдиного процесу.

У сукупності отримані результати свідчать, що моноліт не лише демонструє вищу швидкодію, але й значно стабільніше працює при інтенсивному використанні кешу. Мікросервіси, хоча й суттєво програють у продуктивності, все ж залишаються більш гнучкими з точки зору масштабування та незалежності компонентів, що зумовлює вибір на користь тієї чи іншої архітектури залежно від потреб проєкту.

ВИСНОВКИ

Таким чином, у кваліфікаційній роботі виконано дослідження методів кешування в монолітних і мікросервісних вебсистемах та досягнуто поставленої мети – розроблено та експериментально перевірено моделі кешування для різних архітектур із подальшим порівнянням їхньої ефективності. Сучасний розвиток науки та технологій характеризується високою динамічністю й міждисциплінарністю, що зумовлює постійне оновлення підходів, методів та інструментів у різних галузях [21-29].

У процесі виконання роботи вирішено такі завдання:

– проведено аналіз наукових і технічних джерел щодо застосування кешування у вебсистемах, що дозволило визначити сучасні підходи, інструменти та тенденції у цій сфері;

– досліджено архітектурні особливості монолітних і мікросервісних систем, що дало можливість правильно проаналізувати їх вплив на стратегії кешування та ефективність обробки запитів;

– розглянуто сучасні методи кешування, зокрема in-memory cache, distributed cache, Redis та Memcached, що дозволило правильно визначити їхню придатність для різних типів архітектур;

– сформульовано математичні моделі процесів кешування для монолітної та мікросервісної архітектур, що дало змогу формалізувати залежності між продуктивністю, часом відгуку та частотою оновлення даних;

– розроблено дві експериментальні модельні системи: монолітну вебсистему з локальним кешем і розподіленим, та мікросервісну систему з розподіленим кешем на основі Redis;

– проведено серію експериментів із використанням Docker та k6 для вимірювання часу відгуку, навантаження на базу даних, обсягу використаної пам’яті та стабільності роботи під час зростання кількості запитів, що дало можливість виконати аналіз отриманих результатів, здійснити порівняння

продуктивності та ефективності методів кешування у двох архітектурних підходах та задовольнити мету кваліфікаційної роботи.

Наукова новизна роботи полягає у комплексному підході до оцінювання ефективності кешування в контексті архітектурних особливостей вебсистем, що поєднує математичне моделювання та експериментальну перевірку.

Загалом результати виконаної роботи підтверджують важливість раціонального підходу до вибору методів кешування залежно від архітектури вебсистеми. Проведений аналіз демонструє, що навіть незначні відмінності у способах організації кешу можуть суттєво впливати на кінцеву продуктивність та стабільність застосунку, особливо у високонавантажених середовищах. Виявлені закономірності дозволяють розглядати кешування не лише як інструмент оптимізації, а як невід’ємний структурний елемент архітектури, що визначає здатність системи масштабуватися та ефективно обробляти великі обсяги даних. Отримані результати можуть бути використані для подальшого вдосконалення вебсистем, розроблення рекомендацій з оптимізації роботи серверних застосунків та формування більш гнучких моделей розподіленої обробки даних.

Результати роботи апробовано у вигляді 2 тез доповідей під час IX Міжнародної студентської конференції «Модернізація та сучасні українські і світові наукові дослідження» [30], IX Міжнародної науково-практичної конференції «Scientific researches and methods of their carrying out: world experience and domestic realities» [31].

ПЕРЕЛІК ДЖЕРЕЛ ПОСИЛАННЯ

1. Gos, K., & Zabierowski, W. (2020). The comparison of microservice and monolithic architecture. In 2020 IEEE XVIth International Conference on the Perspective Technologies and Methods in MEMS Design, pp. 150-153.

2. Gao, Y., Casale, G., & Singhal, R. (2024). Performance modeling of distributed data processing in microservice applications. ACM Transactions on Modeling and Performance Evaluation of Computing Systems, 9(4), pp. 1-30.

3. Podlipnig, S., & Böszörmenyi, L. (2003). A survey of web cache replacement strategies. ACM Computing Surveys (CSUR), 35(4), pp. 374-398.

4. Mirheidari, S. A., Golinelli, M., Onarlioglu, K., Kirda, E., & Crispo, B. (2022). Web cache deception escalates!. In 31st USENIX Security Symposium (USENIX Security 22), pp. 179-196.

5. Falkevych, V., & Lisniak, A. (2025). Cache invalidation based on a declarative approach for separating business logic of microservices from cache update rules. Eastern-European Journal of Enterprise Technologies, 2, pp. 68-76.

6. Zheng, Q., Yang, T., Kan, Y., Tan, X., Yang, J., & Jiang, X. (2021). On the analysis of cache invalidation with LRU replacement. IEEE Transactions on Parallel and Distributed Systems, 33(3), pp. 654-666.

7. Al-Debagy, O., & Martinek, P. (2018). A comparative review of microservices and monolithic architectures. In 2018 IEEE 18th International Symposium on Computational Intelligence and Informatics (CINTI), pp. 149-154.

8. Вечірська І.Д., Вечірська А.Д. (2021) Проблематика застосування засобів алгебри скінченних предикатів на теренах системного аналізу. Proceedings of XI International scientific and practical conference «Fundamental and applied research in the modern world». – Boston, USA. 9-11, pp. 275-277.

9. Кучеренко, Є. І., Творошенко, І. С., Анопрієнко, Т. В. (2016) Моделювання та оцінювання станів складних об’єктів із застосуванням формальної логіки. Системи обробки інформації, (2), с. 76-82.

10. Shelest V., Yakovleva O. (2024) Research on selecting web application architecture based on the analysis of applied requirements. Proceedings of the X International Scientific and Practical Conference «Computerintegrated technologies of automation of technological processes». Hamburg, Germany, pp. 46-54.

11. MongoDB Manual. URL: https://www.mongodb.com/docs/manual/ (дата звернення 12.10.2025).

12. The PostgreSQL Global Development Group. URL: https://www.postgresql.org/docs/ (дата звернення 16.10.2025).

13. Datsok, Y. O., & Vechirska, I. D. (2025). Про застосування методу аналізу ієрархій для вибору онлайн-курсу з вивчення мови SQL [On the application of the analytic hierarchy process for selecting an online SQL course]. In Proceedings of the XV International Scientific and Technical Conference “Modern Directions of Development of Information and Communication Technologies and Control Systems” (Vol. 1, p. 8). Baku–Kharkiv–Žilina.

14. Redis Labs Documentation. URL: https://redis.io/docs/latest/ (дата звернення 05.10.2025)

15. Vechirska, I., Sokolova, A., & Valenda, N. (2025). construction of a formal model of the excursion selection process for students: a case study of the “PUSH” school in Kharkiv in the form of a logical AFP-network. Computer Systems and Information Technologies, (3), pp. 17–27.

16. Chetverikov, G.G. , Vechirska, I.D., Leshchinsky, V.A. (2024) Mathematical modelling and design of multiple-valued logic elements of digital telecommunications networks. 24th International Crimean Conference Microwave and Telecommunication Technology, Conference Proceedings, pp. 354-355.

17. The Apache Software Foundation. URL: https://jmeter.apache.org/usermanual/ (дата звернення 20.10.2025).

18. Artillery Official Documentation. URL: https://www.artillery.io/docs (дата звернення 20.10.2025).

19. Vechirska A.D., Shyrokorad K.A., Supervisor: Vechirska I.D. (2022) Visual modeling of purchase process management in the electronics online store.

Діджиталізація науки як виклик сьогодення: матеріали III Міжнародної студентської наукової конференції, м. Львів. ГО «Молодіжна наукова ліга». Вінниця: ГО «Європейська наукова платформа», с.189- 192.

20. Vechirska I.D., Vechirska A.D., Shyrokorad K.A. (2023) Solving the problem of choosing the bestassembly plan for software deployment in the cloud environment using the analytic hierarchy process. Розвиток суспільства та науки в умовах цифрової трансформації: матеріали IV Міжнародної студентської наукової конференції, м. Дніпро, ГО «Молодіжна наукова ліга». Вінниця: ГО «Європейська наукова платформа», с. 126-129.

21. Daradkeh Y.I., Gorokhovatskyi V., Tvoroshenko I., and Zeghid M. (2022) Tools for fast metric data search in structural methods for image classification, IEEE Access, 10, pp. 124738-124746.

22. Gorokhovatskyi V., Tvoroshenko I., Kobylin O., and Vlasenko N. (2023) Search for visual objects by request in the form of a cluster representation for the structural image description, Advances in Electrical and Electronic Engineering, 21(1), pp. 19-27.

23. Гороховатський В.О., Творошенко І.С., Чмутов Ю.В. (2022) Застосування систем ортогональних функцій для формування простору ознак у методах класифікації зображень, Сучасні інформаційні системи, 6(3), С. 5-12.

24. Гороховатський В., Передрій О., Творошенко І., Марков Т. (2023) Матриця відстаней для множини компонентів структурного опису як інструмент для створення класифікатора зображень, Сучасні інформаційні системи, 7(1), С. 5-13.

25. Pomazan V., Tvoroshenko I., and Gorokhovatskyi V. (2023) Development of an application for recognizing emotions using convolutional neural networks, International Journal of Academic Information Systems Research, 7(7), pp. 25-36.

26. Pomazan V., Tvoroshenko I., and Gorokhovatskyi V. (2023) Handwritten character recognition models based on convolutional neural networks, International Journal of Academic Engineering Research, 7(9), pp. 64-72.

27. Tvoroshenko I., Gorokhovatskyi V., Kobylin O., and Tvoroshenko A. (2023) Application of deep learning methods for recognizing and classifying culinary dishes in images, International Journal of Academic and Applied Research, 7(9), pp. 57-70.

28. Daradkeh Y.I., Gorokhovatskyi V., Tvoroshenko I., Gadetska S., and Al-Dhaifallah M. (2023) Statistical data analysis models for determining the relevance of structural image descriptions, IEEE Access, vol. 11, pp. 126938-126949.

29. Gorokhovatskyi V., Tvoroshenko I., and Yakovleva O. (2024) Transforming image descriptions as a set of descriptors to construct classification features, Indonesian Journal of Electrical Engineering and Computer Science, vol. 33, no. 1, pp. 113-125..

30. Воронін Д.О. (2025) Аналіз методів кешування у монолітних і мікросервісних вебсистемах. IX Міжнародна студентська наукова конференція «Модернізація та сучасні українські і світові наукові дослідження». Житомир, Україна, с. 307-308.

31. Воронін Д.О. (2025) Експериментальне дослідження ефективності алгоритмів заміщення кешу LRU та MRU у вебсистемах. IX Міжнародна науково-практична конференція «Scientific researches and methods of their carrying out: world experience and domestic realities». Вінниця, Україна, с. 808-810.