Міністерство освіти і науки України
НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ «КИЄВО-МОГИЛЯНСЬКА АКАДЕМІЯ»
Кафедра мультимедійних систем факультету інформатики

Кваліфікаційна робота

освітній ступінь – бакалавр на тему

«Дослідження стратегій кешування у високонавантажених .NET-додатках»

Виконала: студентка 4-го року навчання
Освітньої програми «Комп'ютерні науки»
Дяченко Владислава Юліївна
Керівник: Борозенний С.О., старший викладач
_____ 2026 р.

Київ 2026

Міністерство освіти і науки України
НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ «КИЄВО-МОГИЛЯНСЬКА АКАДЕМІЯ»
Кафедра мультимедійних систем факультету інформатики

ЗАТВЕРДЖУЮ
Зав. кафедри інформатики,
доцент, кандидат наук
_____ Жежерун О.П.
(підпис)
“____” _____ 2026 р.

ІНДИВІДУАЛЬНЕ ЗАВДАННЯ

на кваліфікаційну роботу

студентці 4 року навчання БП «Комп'ютерні науки» Дяченко Владиславі Юліївні на тему: Дослідження стратегій кешування у високонавантажених .NET-додатках

Зміст ТЧ до кваліфікаційної роботи:

Зміст

Вступ

Теоретичні основи організації кешування у високонавантажених системах

Аналіз інструментарію та особливостей кешування в екосистемі .NET

Розробка та експериментальне дослідження адаптивної системи кешування

Висновки до роботі

Список використаних джерел

Дата видачі “8” жовтня 2025 р.

Керівник _____
(підпис)

Завдання отримала _____
(підпис)

Календарний план виконання роботи

Тема: **Дослідження стратегій кешування у високонавантажених .NET-додатках**

№ п/п	Назва етапу дипломної роботи	Термін виконання етапу	Примітка
1	Отримання теми кваліфікаційної роботи	Жовтень 2025	
2	Пошук тематичної літератури	Жовтень 2025 - Листопад 2025	
3	Написання першого розділу теоретичної частини щодо аналізу теми, огляду патернів та проблематики	Грудень 2025 - Січень 2026	
4	Написання другого розділу теоретичної частини щодо аналізу інструментарію	Січень 2026	
5	Проєктування та розробка практичної частини	Лютий 2026	
6	Дослідження та аналіз отриманих результатів та написання третього розділу текстової частини роботи	Лютий 2026 - Березень 2026	
7	Написання вступу та висновків	Березень 2026	
8	Оформлення роботи відповідно до вимог	Березень 2026	
9	Узгодження роботи з науковим керівником та внесення коректив	Березень 2026	
10	Створення презентації та підготовка доповіді	Квітень 2026	
11	Захист кваліфікаційної роботи	Червень 2026	

Дяченко В.Ю. _____

Борозенний С.О. _____

«____» _____ _____ р.

ЗМІСТ

ВСТУП

Сучасний етап розвитку програмної інженерії характеризується стрімким зростанням обсягів даних та інтенсивності користувацьких запитів. Успіх веб-орієнтованих систем сьогодні залежить не лише від зручності інтерфейсу чи чистоти архітектури коду, але й від здатності системи забезпечувати високу швидкодію та відмовостійкість в умовах пікових навантажень.

Для високонавантажених систем ключовими показниками якості стають час відгуку (latency) та пропускна здатність (throughput). Коли кількість користувачів масштабується до сотень тисяч або мільйонів, пряме звернення до джерел даних, таких як бази даних, зовнішні API, при кожному запиті стає слабким місцем. Це призводить до деградації продуктивності та, як наслідок, до втрати аудиторії.

Окрім технічних аспектів, критично важливим є економічний фактор. Масштабування апаратної інфраструктури для підтримки продуктивності часто є економічно невиправданим через швидке зростання витрат на хмарні ресурси та обслуговування. Тому на перший план виходять методи архітектурної оптимізації, основним з яких є кешування даних.

Кешування виступає фундаментом, що дає змогу знайти баланс між вартістю інфраструктури та якістю обслуговування клієнтів. Воно сприяє суттєвому зниженню навантаження на первинні сховища даних та зменшити час відповіді сервера. Проте в екосистемі .NET ефективне використання кешу вимагає вирішення низки складних завдань: керування оперативною пам'яттю (Garbage Collection), забезпечення узгодженості даних та вибір оптимальних стратегій інвалідації (TTL).

Проведений у роботі аналіз існуючих стратегій та вищезазначені виклики зумовлюють необхідність розробки адаптивних алгоритмів, здатних динамічно реагувати на зміни в патернах доступу користувачів.

Метою цієї кваліфікаційної роботи є підвищення ефективності обробки запитів у високонавантажених .NET-додатках шляхом розроблення методу адаптивного кешування з динамічним керуванням часом життя даних (TTL).

Об’єктом дослідження є процеси обробки даних та забезпечення швидкодії у високонавантажених розподілених інформаційних системах.

Предметом дослідження є методи, моделі та алгоритми багаторівневого кешування даних з використанням адаптивних стратегій керування життєвим циклом записів у середовищі .NET.

Наукова новизна отриманих результатів зводиться до наступного:

1. **Удосконалено метод визначення часу життя (TTL) кеш-записів** шляхом впровадження моделі ковзного вікна часу (Sliding Window). На відміну від стандартних підходів із фіксованим часом життя або простими лінійними лічильниками, запропонований метод допомагає враховувати ретроспективну популярність даних та нівелювати вплив випадкових сплесків трафіку, забезпечуючи точне прогнозування корисності об’єктів у пам’яті;
2. **Запропоновано спосіб стабілізації високонавантаженої інфраструктури** через інтеграцію механізму випадкового зсуву (Jitter) у алгоритм розрахунку адаптивного TTL. Це дозволяє запобігти виникненню ефекту натовпу (Cache Stampede) шляхом стохастичного розподілення моментів інвалідації популярних ключів у часі, що мінімізує ризик виникнення ударних навантажень на первинні джерела даних та базу даних;
3. **Набув подальшого розвитку метод багаторівневого кешування в .NET** через впровадження алгоритму динамічної деградації (Decay). Це забезпечує проактивне очищення оперативної пам'яті від застарілих («холодних») даних до моменту повного закінчення

їхнього TTL, що суттєво знижує навантаження на систему автоматичного прибирання сміття (Garbage Collection) та запобігає фрагментації керованої купи (Managed Heap).

Використані терміни

Адаптивне кешування - метод динамічного керування часом життя даних (TTL) на основі частоти запитів у режимі реального часу;
Високонавантажена система (High-load system) - програмний комплекс, здатний забезпечувати високу швидкодію та відмовостійкість в умовах стрімкого зростання обсягів даних та інтенсивності користувацьких запитів;
Гарячі дані (Hot tier) - найбільш часто запитувані дані, які доцільно розміщувати максимально близько до обчислювального вузла;
Голодування пулу потоків (Thread Pool Starvation) - стан системи, при якому всі доступні робочі потоки заблоковані (наприклад, тривалими паузами збирача сміття), що призводить до різкого зростання затримок та неможливості обробки нових HTTP-запитів;
Деградація TTL (Decay) - механізм проактивного та поступового зменшення часу життя об'єкта в кеші при падінні частоти звернень до нього для швидкого звільнення оперативної пам'яті;
SOLID - п'ять принципів об'єктно-орієнтованого проєктування, що забезпечують гнучкість, масштабованість і підтримуваність програмного коду;
Ефект натовпу (Cache Stampede) - критичний стан, коли при спливанні терміну дії популярного ключа сотні запитів одночасно та неконтрольовано навантажують базу даних;
Закон Парето (Розподіл 80/20) - принцип нерівномірного розподілу, за яким 20% найпопулярніших товарів (даних) формують 80% усього мережевого трафіку в системі;
Керована купа (Managed Heap) - зона оперативної пам'яті в середовищі .NET, де розміщуються всі посилальні типи даних, зокрема елементи кешу;
Кешування (Caching) - процес зберігання копій даних у тимчасовому сховищі (кеші), доступ до якого здійснюється значно швидше, ніж до основного джерела;
Локальне кешування (In-Memory) - зберігання даних безпосередньо в оперативній пам'яті процесу додатка (наприклад, через інтерфейс IMemoryCache);
Мережа доставки контенту (CDN - Content Delivery Network) - географічно розподілена мережа серверів, що кешує контент максимально близько до кінцевого користувача для мінімізації мережевих затримок;

Пробуксовка кешу (Cache Thrashing) - негативна ситуація, коли через занадто короткий статичний TTL дані видаляються з кешу швидше, ніж надходять повторні запити на них;
Розподілене кешування (Distributed Caching) - використання зовнішнього централізованого сховища (наприклад, Redis), спільного та доступного для всіх екземплярів розподіленої системи;
Серіалізація та десеріалізація - процес перетворення об'єктів у бінарний або текстовий формат (наприклад, JSON-рядки) для їхньої передачі по мережі або збереження у розподіленому кеші;
Стан гонитви (Race Condition) - помилка проєктування багатопотокових систем, при якій результат операції непередбачувано залежить від порядку виконання потоків. Усувається використанням потокобезпечних структур;
Стрибок навантаження (Spike Load) - сценарій тестування, що імітує різке та непередбачуване хвилеподібне зростання трафіку (наприклад, розпродаж "Чорна п'ятниця") для перевірки відмовостійкості архітектури;
Cache-Aside (Lazy Loading) - архітектурний патерн, при якому додаток самостійно перевіряє кеш і, у разі промаху, завантажує дані з БД та оновлює кеш;
Cache Hit Ratio (Коефіцієнт влучань) - ключова метрика ефективності, що відображає відсоток запитів, успішно оброблених кешем без звернення до бази даних;
Garbage Collector (GC) - автоматична підсистема керування пам'яттю в .NET, що використовує поколінну модель (Gen 0, Gen 1, Gen 2) для очищення відпрацьованих об'єктів;
Jitter (Випадковий зсув) - механізм додавання випадкової похибки до розрахованого часу TTL для розподілення моментів інвалідації ключів у часі та запобігання перевантаженню бази даних;
Large Object Heap (LOH) - спеціальна область пам'яті для об'єктів розміром понад 85 000 байт, яка не підлягає автоматичній дефрагментації (ущільненню) та очищується найрідше;
Latency (Затримка) - показник якості, що визначає час відгуку системи на запит користувача;
Throughput (Пропускна здатність) - кількість запитів, які система здатна обробити за одиницю часу (вимірюється в RPS - Requests Per Second);
TTL (Time-To-Live) - параметр, що визначає примусовий час життя (тривалість зберігання) запису в кеші до моменту його видалення.

РОЗДІЛ 1

ТЕОРЕТИЧНІ ОСНОВИ ОРГАНІЗАЦІЇ КЕШУВАННЯ У ВИСОКОНАВАНТАЖЕНИХ СИСТЕМАХ

1.1 Роль та місце кешування в архітектурі сучасних програмних систем

У цьому розділі виконується аналіз сучасних архітектурних рішень та стратегій кешування, що є необхідним для виявлення обмежень існуючих підходів та обґрунтування розробки адаптивного алгоритму.

У сучасному проєктуванні високонавантажених систем кешування виступає одним із фундаментальних архітектурних компонентів, що забезпечує продуктивність та масштабованість. Згідно з визначенням, кешування - це процес зберігання копій даних у тимчасовому сховищі (кеші), доступ до якого здійснюється значно швидше, ніж до основного джерела даних, такого як база даних або віддалений веб-сервіс [2; 4].

В архітектурному плані кеш розташовується як проміжний шар між споживачем даних (програмою або користувачем) та джерелом істини (базою даних) [1]. Його основне завдання - наблизити часто запитувані дані (їх ще називають “гарячими даними”) до обчислювального вузла, який їх потребує. Замість того, щоб при кожному запиті звертатися до повільних носіїв інформації, тобто дискової підсистеми або через мережу, система спочатку перевіряє наявність даних у швидкій пам'яті кешу [3].

Впровадження кешування у програмну систему вирішує три ключові проблеми [1; 3; 4]:

1. **Зниження затримок (Latency Reduction).** Оскільки кеш зазвичай розміщується в оперативній пам'яті (RAM), час доступу до даних вимірюється наносекундами або мікросекундами, на відміну від мілісекунд при роботі з традиційними базами даних. Це дає змогу

суттєво скоротити час відгуку системи для кінцевого користувача.

2. **Зменшення навантаження на джерело даних (Reduced Load).** Кешування діє як буфер, що перехоплює повторювані запити. Це дозволяє розвантажити бекенд та базу даних, звільняючи їх ресурси для виконання більш складних транзакційних операцій або запитів, що не підлягають кешуванню.
3. **Підвищення пропускної здатності (Throughput Improvement).** Зменшуючи час обробки кожного окремого запиту та навантаження на інфраструктуру, система отримує здатність обслуговувати більшу кількість одночасних користувачів без необхідності пропорційного збільшення апаратних ресурсів.

Варто зазначити, що роль кешування стає найбільш вагомою у сценаріях, де навантаження на читання даних переважає над записом (read-heavy workloads), а самі дані змінюються відносно рідко [4]. У таких випадках кеш дає нагоду досягти економічної ефективності, зменшуючи витрати на хмарні ресурси та масштабування баз даних, забезпечуючи при цьому стабільно високу швидкість роботи додатку [1].

Таким чином, кешування є невід'ємною частиною архітектури сучасних розподілених систем, що забезпечує баланс між швидкістю доступу до інформації та ефективністю використання ресурсів.

1.2. Класифікація рівнів та стратегій кешування

1.2.1. Клієнтське кешування

Клієнтське кешування виступає першим бар'єром (рівнем) у системі кешування. У цьому сценарії дані зберігаються безпосередньо на пристрої

користувача, що допомагає уникнути мережевих запитів. Виділяють два основні підходи до організації клієнтського кешу [5]:

1. Браузерний кеш (Browser Cache) - це механізм, що дає браузеру зберігати копії завантажених ресурсів веб-сторінки: зображень, відеофайлів, таблиць стилів (CSS) та скриптів (JS). Алгоритм роботи відбувається таким чином: коли браузер ініціює запит до сервера, спочатку відбувається перевірка локального сховища на наявність актуальної версії контенту. Якщо дані присутні та валідні, браузер завантажує їх прямо з пам'яті диска або оперативної пам'яті [5].

   Для того, щоб зрозуміти чи проходить кеш валідацію, було представлено ряд правил, які здійснюються через HTTP-заголовки (HTTP Headers) [5]:

   - Без кешування (no-cache) - вказує браузеру, що перед використанням збереженої копії необхідно зробити запит до сервера для перевірки її актуальності;
   - Без збереження (no-store) - забороняє браузеру зберігати будь-яку частину запиту чи відповіді;
   - Публічний - дозволяє кешувати ресурс як браузеру, так і будь-яким проміжним вузлам;
   - Приватний - вказує, що ресурс призначений для одного користувача і не може бути збережений на публічних проміжних вузлах (тільки в браузері);
   - Максимальний час (max-age) - визначає час життя кешу у секундах з моменту отримання відповіді.
2. Кеш даних застосунку (web-storage & cookies) - оскільки HTTP протокол не може відстежити відповіді та запити від сервера чи клієнта, виникає потреба впроваджувати такі механізми [6]:

   - Локальне сховище (local storage) - допускає веб-сайтам зберігати пари “ключ-значення” у браузері. Дані залишаються

доступними навіть після закриття вкладки, браузера або перезавантаження комп'ютера, доки не будуть явно видалені;

- Сховище сесій (session storage) - забезпечує тимчасову зберігання даних у браузері виключно в рамках однієї сесії. Однак після закриття вкладки або браузера всі збережені дані автоматично видаляються;
- Файли cookie - невеликі фрагменти даних, що зберігаються в браузері користувача та передаються на сервер із кожним HTTP-запитом

1.2.2. Кешування на рівні доставки та інфраструктури

Для зниження мережевих затримок, навантаження на основні сервери та оптимізації витрат на передачу даних впроваджується рівень інфраструктурного кешування. Цей підхід базується на використанні проміжних компонентів мережі, які наближають контент до кінцевого користувача та мінімізують кількість прямих запитів до бекенд-сервісів. Виділяють два ключові компоненти цього рівня [7]:

1. Мережі доставки контенту (CDN - Content Delivery Network) - це система географічно розподілених серверів, що кешує статичний контент (зображення, CSS та JS-файли) на вузлах, розташованих фізично найближче до користувача. Використання CDN дає можливість обслуговувати мільйони запитів під час пікових навантажень, не залучаючи потужності основного сервера;
2. Зворотні проксі-сервери (Reverse Proxies) - це веб-сервери, які розташовуються перед серверами додатку, кешують HTTP-відповіді та повертають їх клієнту самостійно. Окрім кешування, вони часто виконують функцію балансування навантаження (Load Balancing),

розподіляючи трафік між кількома екземплярами додатку для підвищення відмовостійкості.

1.2.3. Серверне кешування

Серверне кешування передбачає тимчасове зберігання копій часто запитуваного контенту, вже оброблених даних або результатів запитів безпосередньо на спеціальному проміжному сервері або в пам'яті для швидкого доступу. Це допускає уникнути повторного виконання ресурсоємних обчислень та звернень до бази даних, зменшуючи час відгуку системи, зменшує навантаження на основний сервер та прискорює завантаження для користувачів, скорочуючи мережевий трафік.
Залежно від архітектури зберігання, цей рівень поділяють на дві відмінні стратегії [9].

1.2.3.1. Локальне кешування (In-Memory Caching)

При цьому підході дані зберігаються безпосередньо в оперативній пам'яті (RAM) процесу, що виконує додаток. Це найпростіший метод реалізації, що не вимагає додаткової інфраструктури.

Переваги: забезпечує найвищу швидкість доступу (в наносекундах), оскільки відсутні витрати на мережеву передачу даних та серіалізацію та десеріалізацію об'єктів.

Недоліки: кеш є ізольованим для кожного екземпляра сервера. У розподілених системах (мікросервісах) це призводить до проблеми неузгодженості даних (cache incoherence), коли різні сервери можуть віддавати різні версії одних і тих самих даних. Крім того, обсяг кешу жорстко обмежений доступною пам'яттю сервера [9].

1.2.3.2. Розподілене кешування (Distributed Caching)

Ця стратегія використовує зовнішнє централізоване сховище, до якого мають доступ усі екземпляри додатку через мережу.

Переваги: гарантує строгу узгодженість даних, тобто всі сервери бачать одне й те саме значення, та дозволяють зберігати значно більші обсяги інформації, які не залежать від пам'яті окремого додатку. Дані зберігаються навіть при перезавантаженні серверів додатку.

Недоліки: доступ до даних повільніший порівняно з локальним кешуванням через мережеві затримки та необхідність серіалізації даних для передачі по мережі [9].

1.2.4. Кешування на рівні даних

Кешування на рівні даних є останнім шаром оптимізації перед безпосереднім зверненням до фізичної дискової підсистеми. Основна мета цього рівня - уникнути повільних операцій вводу-виводу та виконання ресурсномістких обчислень для складних SQL-запитів, які повторюються.

Виділяють такі стратегії організації кешування на цьому рівні [8]:

1. Внутрішнє кешування СКБД (Internal Database Caching) - сучасні реляційні бази даних автоматично використовують виділену частину оперативної пам'яті (Buffer Pool) для зберігання сторінок даних та індексів, до яких відбувалося звернення. Це дає змогу виконувати наступні операції читання безпосередньо з пам'яті (RAM), минаючи повільний жорсткий диск.
2. Кешування результатів запитів (Query Caching) - механізм, при якому база даних або ORM (Object-Relational Mapping) зберігає готовий

результат виконання конкретного SQL-запиту. Якщо система отримує ідентичний запит, вона миттєво повертає збережені дані [8].

1.3. Огляд основних патернів роботи з кешем

Ефективність системи кешування визначається не лише обраним сховищем, а й стратегією взаємодії між додатком, кешем та базою даних. Вибір конкретного патерну залежить від співвідношення операцій читання й запису та вимог до актуальності даних [2; 9].

У сучасних високонавантажених системах застосовують такі фундаментальні патерни:

1. Відкладене “ліниве” завантаження (Cache-Aside або Lazy Loading) - це найпоширеніший підхід, де логіка керування кешем реалізована безпосередньо в коді додатку. Алгоритм роботи має такий вигляд: додаток робить запит до кешу і, якщо виникає промах (Cache Miss), то додаток самостійно звертається до бази даних. Отриманий результат записується в кеш для подальшого використання і повертається клієнту [9].

Цей патерн ідеально підходить для систем з інтенсивним читанням (Read-heavy), де дані не оновлюються занадто часто.

2. Читання через кеш (Read-Through) - цей підхід схожий на ліниве завантаження, але з великою та важливою архітектурною відмінністю: додаток звертається не до бази даних, а бібліотеки кешування. Якщо даних у кеші немає, бібліотека сама йде в базу, завантажує та повертає дані [9].

Цей патерн робить код додатку чистішим, приховуючи логіку роботи з базою даних в середині провайдера кешу.

3. Наскрізний запис (Write-Through) - при цьому підході запис даних відбувається синхронно: спочатку в кеш, а потім одразу в базу даних. Операція вважається успішною тільки тоді, коли дані оновлено в обох місцях [9].

Цей патерн з одного боку гарантує, що дані в кеші ніколи не будуть застарілими відносно бази даних, проте з іншого боку збільшує час виконання операцій запису, оскільки потрібно чекати відповіді від двох систем.

4. Відкладений запис (Write-Back або Write-Behind) - найбільш продуктивний, але ризикований патерн. Додаток записує дані в кеш і одразу отримує підтвердження. Оновлення основної бази даних відбувається асинхронно у фоновому режимі, але варто зазначити, що із затримкою [2].

Цей патерн є критично важливим для систем з величезною кількістю записів, де база даних може не встигати обробляти кожен запит миттєво.

1.4. Проблематика керування життєвим циклом даних у кеші

Керування даними в кеші не обмежується лише їх записом та читанням. У високонавантажених системах критично важливими є процеси своєчасного видалення застарілих даних та звільнення місця для нової інформації. Неефективне керування цими процесами може призвести до переповнення пам'яті або роботи з неактуальною інформацією.

1.4.1. Аналіз стратегій витіснення даних (Eviction Policies)

Оскільки обсяг оперативної пам'яті завжди обмежений, система повинна мати алгоритм вибору записів для видалення, коли кеш заповнений. Виділяють такі базові стратегії [2]:

- LRU - Least Recently Used (найменш нещодавно використані) - алгоритм видаляє елементи, до яких не зверталися найдовше. Ця стратегія базується на припущенні, що якщо дані запитували нещодавно, вони знадобляться знову в найближчому майбутньому. Це найпоширеніша стратегія для загальних завдань;
- LFU - Least Frequently Used (найменш часто використовувані) - система підраховує кількість звернень до кожного запису. При переповненні видаляється елемент з найменшим лічильником популярності. Це ефективно для стабільних патернів доступу, але алгоритм повільніше адаптується до нових трендів;
- FIFO - First-In, First-Out (першим зайшов - першим вийшов) - це найпростіший метод, що працює за принципом черги: першим видаляється найстаріший запис, незалежно від того, як часто до нього звертаються. Має низькі накладні витрати, але низьку ефективність влучань;
- TTL - Time-To-Live (час життя) - це примусове видалення даних після спливання заданого часу життя, незалежно від частоти використання. Це гарантує, що дані не будуть "вічно" займати пам'ять [9].

1.4.2. Проблеми узгодженості даних у розподілених системах

Головним викликом при використанні кешування є підтримка узгодженості (consistency) - відповідності даних у кеші реальному стану даних у базі. Згідно з [9], у розподілених системах, особливо при використанні локального кешу на кількох серверах, виникає проблема розсинхронізації: один сервер може оновити дані, тоді як інші продовжують віддавати стару версію зі свого локального кешу. Для вирішення цього використовують дві моделі [2]:

- Сувора узгодженість (strong consistency) - гарантує, що після оновлення жоден клієнт не отримає застарілі дані. Це часто вимагає блокування операцій або використання розподілених транзакцій, що знижує швидкість;
- Узгодженість у кінцевому рахунку (eventual consistency) - допускає тимчасову розбіжність даних заради високої продуктивності. Це стандарт для високонавантажених систем, де мілісекундна затримка оновлення є прийнятною.

1.4.3. Методи запобігання ефектам натовпу та "холодного старту"

Окрім стандартної роботи, існують граничні стани системи, які можуть спричинити аварійну відмову [2]:

- Ефект натовпу (cache stampede) - Виникає, коли термін дії популярного ключа спливає, і одночасно сотні клієнтів роблять запит до цього ключа. Оскільки в кеші його вже немає, всі запити одночасно йдуть до бази даних, що може "покласти" її;
- "Холодний старт" (cold start) - це ситуація після перезапуску системи, коли кеш абсолютно порожній. Перші запити йдуть напряму в БД, створюючи пікове навантаження.

Використання стратегії попереднього завантаження (pre-fetching), коли система заздалегідь заповнює кеш найбільш популярними даними ще до початку обслуговування користувачів, є досить надійним методом вирішення цієї проблеми [2].

1.5. Вплив параметра TTL на ефективність системи та використання ресурсів

Налаштування часу життя даних (TTL - Time-To-Live) є критичним фактором, що визначає баланс між продуктивністю системи та актуальністю інформації. Згідно з аналізом компромісів кешування (cache trade-offs), виділяють три основні аспекти впливу цього параметра [2]:

- Свіжість даних проти тривалості зберігання (Data Freshness vs. Duration) - встановлення довгого TTL покращує продуктивність, оскільки зменшує необхідність повторних звернень до джерела даних. Однак, чим довше дані знаходяться в кеші, тим вищий ризик того, що вони стають застарілими (stale data). Якщо інформація змінюється часто, тривалий TTL може призвести до того, що користувачі отримуватимуть неактуальний контент.
- Використання пам'яті проти розміру кешу (Memory Usage vs. Cache Size) - збільшення часу життя записів неминуче призводить до зростання обсягу зайнятої оперативної пам'яті, оскільки старі об'єкти не видаляються. Це створює ризик переповнення пам'яті (Out-Of-Memory errors) у середовищах з обмеженими ресурсами. Необхідно знаходити баланс, щоб кеш містив достатньо даних для ефективної роботи, але не споживав надмірну кількість ресурсів сервера [2].

- Коефіцієнт влучань проти політики витіснення (Hit Ratio vs. Eviction) - короткий TTL може призвести до передчасного видалення корисних даних, що знижує коефіцієнт влучань (Cache Hit Ratio) і змушує систему частіше звертатися до повільного бекенду. Високий коефіцієнт влучань знижує затримки і навантаження, але вимагає, щоб найбільш затребувані дані залишалися в кеші якомога довше.

Таким чином, вибір статичного значення TTL завжди є компромісом. Для досягнення максимальної ефективності параметр TTL має корелювати з частотою зміни конкретного типу даних, а не встановлюватися глобально для всієї системи.

Висновки до розділу 1

У першому розділі проведено комплексний аналіз теоретичних засад організації кешування у високонавантажених системах. Встановлено, що кешування є критично важливим архітектурним компонентом, який забезпечує не лише зниження навантаження на серверну інфраструктуру, але й мінімізацію затримок для кінцевого користувача.

У ході дослідження було класифіковано рівні кешування та визначено, що для бекенд-розробки ключовим є серверний рівень. Порівняльний аналіз локального та розподіленого кешу показав, що жоден із цих підходів не є універсальним: перший виграє у швидкості, але програє в узгодженості даних, тоді як другий забезпечує надійність ціною додаткових мережевих витрат.

Окрему увагу приділено проблемам керування життєвим циклом даних. Виявлено в результаті аналізу, що використання статичних параметрів часу життя (TTL) та стандартних алгоритмів витіснення (LRU, FIFO) не завжди є ефективним в умовах змінного навантаження. Це створює ризики або отримання застарілих даних, або неефективного використання оперативної пам'яті.

На основі проведеного аналізу обґрунтовано доцільність розробки адаптивного механізму кешування, який зможе динамічно коригувати параметри TTL залежно від частоти запитів, що стане предметом подальшої розробки у цій роботі.

РОЗДІЛ 2
АНАЛІЗ ІНСТРУМЕНТАРІЮ ТА ОСОБЛИВОСТЕЙ КЕШУВАННЯ В ЕКОСИСТЕМІ .NET

2.1. Специфіка керування пам'яттю в .NET середовищі

Ефективність кешування в .NET неможливо розглядати окремо від внутрішніх механізмів керування пам'яттю CLR (Common Language Runtime). Оскільки кеш - це набір об'єктів, що зберігаються в оперативній пам'яті, розуміння того, як саме ці об'єкти розміщуються та звільняються, є ключем до побудови високопродуктивної системи.

2.1.1. Організація Managed Heap та робота Garbage Collector

У середовищі .NET пам'ять логічно розділена на дві основні зони: стек (Stack) та керована купа (Managed Heap). Стек використовується для виконання коду та зберігання типів значень (Value Types), які мають короткий життєвий цикл і видаляються автоматично після завершення методу. Натомість, усі посилальні типи (Reference Types), до яких належать і елементи кешу, розміщуються в Managed Heap [10].

Для керування об'єктами в купі Garbage Collector (GC) використовує поколінну модель (Generational Model), яка поділяє об'єкти на три категорії [10]:

- Generation 0 (Gen 0) - сюди потрапляють усі нові об'єкти. Очищення цього покоління відбувається найчастіше і є найшвидшим;
- Generation 1 (Gen 1) - слугує буфером для об'єктів, які пережили очищення Gen 0;
- Generation 2 (Gen 2) - тут знаходяться довгоживучі об'єкти. Саме

сюди з часом переміщуються дані кешу, оскільки вони мають тривалий час життя (TTL). Очищення цього покоління є найбільш ресурсоємним і викликає блокування потоків програми.

Окремо виділяють Large Object Heap (LOH) - спеціальну зону для об'єктів, розмір яких перевищує 85,000 байт (наприклад, великі масиви даних або серіалізовані JSON-рядки в кеші). Особливістю LOH є те, що пам'ять тут не дефрагментується автоматично, тобто без спеціальних налаштувань, що при інтенсивному кешуванні великих об'єктів може призвести до фрагментації пам'яті та помилок OutOfMemory [10].

Також важливо враховувати, що .NET підтримує різні режими роботи збирача сміття, вибір яких залежить від типу навантаження на систему [11]:

- Workstation GC (GC робочої станції) - призначений для клієнтських застосунків (консольні, десктопні), де пріоритетом є відгук інтерфейсу користувача. У цьому режимі збірка сміття намагається мінімізувати паузи, але не оптимізована для високої пропускної здатності.
- Server GC (Серверний GC) - є стандартом для високонавантажених серверних застосунків (зокрема ASP.NET Core). У цьому режимі для кожного логічного процесора створюється окрема купа (heap) та виділений потік для збірки сміття. Це забезпечує очищення пам'яті паралельно, значно підвищуючи пропускну здатність та масштабованість системи при інтенсивній роботі з кешем.

2.1.2. Вплив кешування об'єктів на навантаження GC

Інтенсивне використання локального кешування створює специфічне навантаження на Garbage Collector, яке може нівелювати переваги від швидкого доступу до даних. Основна проблема полягає у взаємодії кешу з кучею великих об'єктів (LOH).

Згідно з документацією Microsoft, будь-який об'єкт розміром 85,000 байт і більше автоматично розміщується в LOH [12]. У контексті веб-застосунків це типовий сценарій: кешування серіалізованих списків товарів, HTML-фрагментів або результатів API-запитів часто перевищує цей поріг.

Це призводить до негативних ефектів наведених нижче [12]:

1. Зв'язок з поколінням 2 (Gen 2) - LOH очищається лише під час збірки сміття другого покоління. Це найдорожча операція, яка змушує CLR призупиняти виконання всіх потоків програми. Якщо кеш постійно оновлює великі об'єкти, це провокує часті зупинки сервера.
2. Фрагментація пам'яті - на відміну від SOH (Small Object Heap), пам'ять у LOH за замовчуванням не ущільнюється після видалення об'єктів. Це означає, що між зайнятими блоками пам'яті утворюються "дірки". З часом це призводить до ситуації, коли вільної пам'яті сумарно багато, але систему неможливо виділити суцільний блок для нового великого об'єкта кешу, що викликає помилку OutOfMemoryException.
3. Вартість виділення - хоча запис у LOH відбувається швидше через відсутність копіювання при ущільненні, саме очищення є значно повільнішим. Для високонавантажених систем це означає, що неправильно налаштований TTL для великих об'єктів може призвести до деградації продуктивності [12].

Таким чином, при проєктуванні підсистеми кешування в .NET необхідно враховувати не лише обсяг даних, а й розмір окремих об'єктів, щоб мінімізувати тиск на LOH.

Варто зазначити, що починаючи з версії .NET 4.5.1, розробникам доступна властивість GCSettings.LargeObjectHeapCompactionMode, яка сприяє примусовому виконуванню ущільнення LOH під час наступної збірки сміття. Однак, згідно з документацією [12], ця операція є вкрай ресурсоємною. Примусове переміщення великих блоків пам'яті викликає тривалі затримки в роботі застосунку, тому покладатися виключно на механізми середовища (GC) для вирішення проблем неефективного кешування є хибною стратегією. Це підтверджує необхідність розумного керування часом життя об'єктів (TTL) на рівні логіки програми.

2.2. Огляд стандартних механізмів платформи .NET

Екосистема .NET надає набір уніфікованих абстракцій для реалізації кешування, які дозволяють розробникам змінювати сховища даних без суттєвої зміни бізнес-логіки. Архітектура платформи розділяє ці механізми на три рівні [13; 14]:

1. Локальне кешування (IMemoryCache) -це найшвидший механізм, який зберігає дані безпосередньо в оперативній пам'яті процесу веб-сервера. Реалізація базується на потокобезпечному словнику (ConcurrentDictionary), що забезпечує високу швидкість доступу.
    - Керування пам'яттю: на відміну від старих версій (ASP.NET WebForms), сучасна реалізація IMemoryCache в .NET Core не має автоматичного механізму очищення при нестачі оперативної пам'яті (Memory Pressure). Розробник зобов'язаний явно вказувати SizeLimit та налаштовувати

CompactionPercentage (відсоток записів, що видаляються при переповненні) [13].

- Політики витіснення*:* підтримує AbsoluteExpiration (жорсткий час видалення) та SlidingExpiration (продовження часу життя при зверненні).

2. Розподілене кешування (IDistributedCache) - це інтерфейс-абстракція, що дозволяє підключати зовнішні сховища (Redis, SQL Server, NCache) без зміни коду додатку.
    - Особливості роботи*:* на відміну від IMemoryCache, який зберігає живі об'єкти, IDistributedCache оперує виключно масивами байтів (byte[]). Це означає, що перед записом будь-який об'єкт має бути серіалізований, а при читанні - десеріалізований, що створює додаткове навантаження на CPU [14].
3. Кешування відповідей (Output Caching) - починаючи з .NET 7, платформа отримала вбудований middleware для кешування повних HTTP-відповідей. Таким чином можна зберігати згенеровану HTML-сторінку або JSON-відповідь API цілком, уникаючи виконання всього ланцюжка обробки запиту (Controllers, Filters, Database calls) [15].

Така архітектура надає гнучкість, однак стандартні реалізації мають суттєвий недолік: вони покладаються на статично задані параметри часу життя (TTL), які не враховують поточний стан навантаження на систему.

2.3. Аналіз існуючих рішень та бібліотек

Екосистема кешування в .NET не обмежується лише стандартними

інтерфейсами. Для побудови високонавантажених систем розробники використовують спеціалізовані сховища даних та високорівневі бібліотеки, що розширюють базовий функціонал. Проведемо аналіз найбільш актуальних рішень.

2.3.1. Сховища даних

1. Redis (Remote Dictionary Server) - на сьогодні є стандартом де-факто для розподіленого кешування. Це сховище типу "ключ-значення", яке підтримує складні структури даних (списки, хеш-таблиці, множини).

Особливості: підтримує персистентність (збереження даних на диск), реплікацію та кластеризацію;

Недоліки: Redis є однопоточним у своїй основі, що може стати вузьким місцем при обробці великої кількості дрібних запитів на потужних багатоядерних серверах [16].

2. Memcached - це історичний попередник Redis. Це просте, високопродуктивне сховище для об'єктів довільного типу.

Особливості: на відміну від Redis, є багатопоточним, що дає можливість ефективно використовувати всі ядра CPU;

Недоліки: обмежена функціональність (тільки прості рядки/об'єкти), відсутність вбудованої реплікації та персистентності. Дані втрачаються при перезавантаженні вузла, що робить його менш надійним для критичних даних [16].

3. Microsoft Garnet - новітнє сховище від Microsoft Research, розроблене на мові C# (.NET 8). Воно позиціонується як швидша альтернатива Redis.

Особливості*:* повністю сумісне з протоколом Redis (RESP), що сприяє використанню існуючих клієнтів. Використовує сховище Tsavorite для ефективного керування пам'яттю, уникаючи проблем з Garbage Collector;

Переваги: показує значно вищу пропускну здатність та меншу затримку на 99-му перцентилі порівняно з Redis завдяки ефективній багатопотоковості [17].

2.3.2. Бібліотеки керування кешуванням

Для спрощення роботи зі сховищами в .NET використовують бібліотеки, які вирішують типові проблеми, як-от: синхронізація, серіалізація тощо.

1. FusionCache - на даний момент є однією з найбільш досконалих бібліотек для .NET. Вона реалізує багаторівневе кешування (L1 Memory + L2 Distributed) з автоматичною синхронізацією.

Ключові механізми:

- Cache Stampede Protection: гарантує, що при одночасному запиті одного ключа лише один потік піде в базу даних, а інші чекатимуть результату [18].
- Fail-Safe: якщо база даних або розподілений кеш недоступні, бібліотека може тимчасово повернути застарілі дані, замість того щоб викидати помилку [18].
- Soft Timeouts: якщо отримання свіжих даних займає забагато часу, повертається остання відома версія з кешу.

2. EasyCaching - бібліотека, що надає уніфікований API для роботи з різними провайдерами (Redis, Memcached, SQLite, Disk).

Особливості: підтримує перехоплення запитів для реалізації аспектно-орієнтованого програмування. Це забезпечує додавання кешування через атрибути над методами, не змінюючи код методу [19].

3. LazyCache - легковагова обгортка над стандартним IMemoryCache (локальним кешем).

Особливості: фокусується на потокобезпечності. Використовує механізм Lazy<T>, щоб гарантувати, що "важка" операція створення об'єкта виконається лише один раз, навіть якщо її запитують кілька потоків одночасно [20].

Для наочності зведемо розглянуті рішення у таблицю 2.1.

Рішення	Тип	Ключові переваги	Основні обмеження
Redis	Сховище даних	Стандарт індустрії, підтримка складних структур даних, персистентність (збереження на диск)	Однопотокова архітектура - використовує лише одне ядро CPU
Memcached	Сховище даних	Багатопотоковість, висока швидкість для простих даних, простота налаштування	Відсутність реплікації, втрата даних при перезавантаженні, робота тільки з простими ключами

Garnet	Сховище даних	Екстремальна продуктивність, архітектура на C#, повна сумісність з протоколом Redis	Відносно нове рішення, менша спільнота підтримки порівняно з Redis
FusionCache	Бібліотека	Захист від пікових навантажень, відмовостійкість, автоматичний дворівневий кеш	Складніша конфігурація порівняно з іншими конфігураціями
EasyCaching	Бібліотека	Модульність, підтримка кешування через атрибути над методами	Менший фокус на механізмах відмовостійкості, ніж у FusionCache
LazyCache	Бібліотека	Простота використання, потокобезпечність	Працює лише локально, слабка підтримка розподілених сценаріїв

Табл. 2.1. Порівняння інструментів кешування для .NET

Проведений аналіз показує, що екосистема .NET має потужні інструменти для зберігання даних (Redis, Garnet) та безпечного доступу до них (FusionCache). Існуючі бібліотеки ефективно вирішують технічні проблеми, такі як:

- Узгодженість між рівнями L1/L2
- Захист від пікових навантажень

- Обробка збоїв мережі

Однак, жодне з розглянутих рішень не пропонує механізму автоматичного визначення часу життя даних (TTL).

У всіх документаціях [18; 19; 20] час життя кешу задається розробником статично (наприклад, TimeSpan.FromMinutes(10)). Це змушує інженерів обирати TTL інтуїтивно, що призводить до двох проблем:

1. Занадто довгий TTL: користувачі отримують застарілу інформацію;
2. Занадто короткий TTL: система надмірно навантажує базу даних, хоча дані не змінилися.

Відсутність готового інструменту для адаптивного керування TTL на основі частоти запитів обґрунтовує необхідність розробки власного алгоритму в рамках цієї роботи.

2.4. Обґрунтування необхідності розробки адаптивного алгоритму керування TTL

Як показав аналіз існуючих рішень, сучасні інструменти кешування надають потужні механізми зберігання даних, але перекладають відповідальність за визначення часу життя (Time-To-Live, TTL) на розробника.

Згідно з дослідженнями [21], налаштування TTL є критичним фактором, що визначає компроміс між двома протилежними метриками:

1. Актуальність даних (Data Freshness): короткий TTL гарантує, що користувач бачить свіжу інформацію, але збільшує навантаження на мережу та базу даних;

2. Продуктивність (Performance): довгий TTL максимізує коефіцієнт влучень у кеш (Cache Hit Ratio) і економить ресурси, але підвищує ризик відображення застарілого контенту.

Використання статичного TTL (фіксованого значення для всіх даних або групи даних) має суттєві недоліки в умовах реального навантаження:

- Проблема "гарячих" даних**:** для контенту, який активно обговорюється або змінюється (наприклад, курс валют під час торгів), стандартний TTL у 5-10 хвилин є неприпустимо великим, оскільки інформація втрачає актуальність за секунди;
- Проблема "холодних" даних: для стабільного контенту (наприклад, опис товару), який запитують рідко, той самий TTL у 5 хвилин є занадто малим. Це призводить до зайвих запитів у базу даних там, де можна було б зберігати кеш годинами.

Таким чином, виникає потреба в розробці адаптивного алгоритму, який би автоматично коригував значення TTL для кожного ключа окремо, базуючись на статистиці звернень до нього. Такий підхід дозволить:

1. Зменшити TTL для об'єктів з високою частотою оновлень (підвищення актуальності);
2. Збільшити TTL для стабільних об'єктів з високою частотою читання (економія ресурсів сервера);
3. Виключити людський фактор при налаштуванні параметрів кешування.

Розробка та програмна реалізація такого алгоритму в середовищі .NET є основною метою наступного розділу роботи.

Висновки до розділу 2

У другому розділі було проаналізовано особливості реалізації кешування в екосистемі .NET. Встановлено, що специфіка роботи збирача сміття (Garbage Collector, GC) та наявність купи великих об'єктів (Large Object Heap, LOH) накладають суттєві обмеження на зберігання даних у пам'яті: неправильне керування ними призводить до фрагментації пам'яті та затримок у роботі сервера.

Огляд стандартних інструментів та сторонніх бібліотек показав, що індустрія має надійні засоби для зберігання та синхронізації даних. Проте виявлено критичний недолік існуючих підходів: відсутність автоматизованого керування часом життя даних (TTL). Усі розглянуті системи покладаються на статичні налаштування, які не здатні ефективно адаптуватися до динамічних змін навантаження.

Це підтверджує доцільність розробки власного програмного компонента для адаптивного керування TTL, який враховуватиме частоту запитів та специфіку керування пам'яттю .NET.

РОЗДІЛ 3

РОЗРОБКА ТА ЕКСПЕРИМЕНТАЛЬНЕ ДОСЛІДЖЕННЯ АДАПТИВНОЇ СИСТЕМИ КЕШУВАННЯ

3.1. Архітектура запропонованої системи багаторівневого кешування

Для реалізації адаптивного керування часом життя даних (TTL) було розроблено архітектуру, що базується на гібридному підході (Multi-Level Caching). Відповідно до рекомендацій Microsoft щодо побудови високонавантажених систем [26], така система поєднує високу швидкість локальної пам'яті (In-Memory) та надійність розподіленого сховища (Distributed Cache), доповнюючи їх модулем інтелектуального аналізу трафіку.

Загальна схема взаємодії компонентів системи наведена на рис. 3.1.

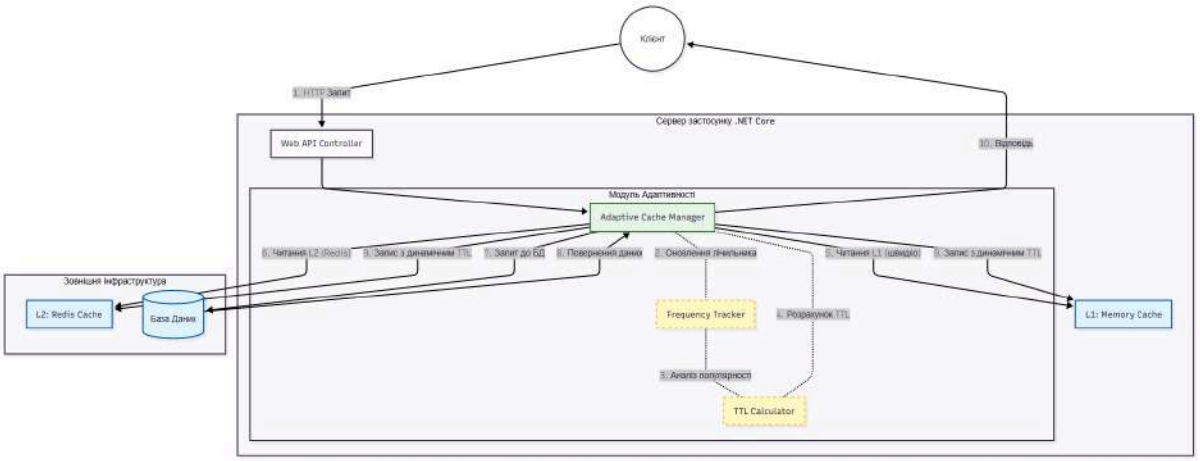

Рис. 3.1. Структурна схема адаптивної системи кешування

Архітектура складається з таких ключових компонентів:

1. **Менеджер адаптивного кешування** (Adaptive Cache Manager) - центральний компонент системи. Його реалізовано за структурним патерном проєктування **"Декоратор" (Decorator)**, описаним у

класичній праці "Банди чотирьох" (GoF) [27]. Він виступає єдиною точкою входу для бізнес-логіки застосунку, тобто контролерів API, динамічно додаючи нову поведінку (адаптивність) до стандартних операцій читання даних. Його основною функцією є інкапсулювання складності роботи з різними рівнями кешу та приймання рішення щодо маршрутизації запитів. Менеджер забезпечує прозорість для клієнта: контролер запитує дані, не знаючи, звідки вони будуть отримані (L1, L2 чи БД).

2. Рівень L1: **Локальний кеш** (In-Memory Cache) - реалізований на базі стандартного інтерфейсу IMemoryCache платформи .NET.

Призначення: зберігання найбільш "гарячих" (часто запитуваних) даних безпосередньо в оперативній пам'яті процесу.

Особливість: забезпечує нульову мережеву затримку, але має обмежений обсяг пам'яті та очищається кожен раз при перезавантаженні сервісу.

3. Рівень L2: **Розподілений кеш** (Distributed Cache) - реалізований з використанням сховища Redis.

Призначення: Забезпечення узгодженості даних між декількома екземплярами сервісу та довготривале зберігання інформації, яка не помістилася в L1. Згідно з офіційною документацією Redis, це сховище забезпечує субмілісекундний час відгуку, що є критичним для другого рівня кешування [28].

Особливість: Виступає гарантом збереження даних у випадку збою окремих вузлів застосунку.

4. **Модуль аналітики** (Adaptive Logic Core) - це унікальна надбудова, що відрізняє розроблену систему від стандартних рішень. Вона складається з двох підсистем:

- Монітор частоти (Frequency Tracker) - високопродуктивний компонент, що в режимі реального часу збирає статистику звернень до кожного ключа кешу. Для мінімізації накладних витрат використовується асинхронний запис без блокування основного потоку;
- Калькулятор TTL (TTL Calculator) - реалізовує математичну модель (описану в п. 3.2), яка на основі даних від монітора динамічно розраховує оптимальний час життя для об'єкта перед його записом у L1/L2.

Алгоритм обробки запиту

Система працює за модифікованою стратегією Cache-Aside (Lazy Loading). Цей патерн є стандартом для хмарних застосунків, де критично важливою є актуальність даних та економія ресурсів [29]:

1. При надходженні запиту Монітор частоти інкрементує лічильник популярності ключа.
2. Виконується пошук у L1. При успіху (Cache Hit) дані повертаються миттєво.
3. При промаху (L1 Miss) виконується пошук у L2.
4. При промаху на обох рівнях (L2 Miss) Менеджер звертається до Бази Даних.
5. Отримані дані передаються в Калькулятор TTL, який визначає час їх зберігання на основі поточної популярності.
6. Дані записуються в L1 та L2 з новим, адаптивним TTL.

Такий підхід допомагає системі автоматично "підлаштовуватися" під зміну навантаження, збільшуючи час життя популярних об'єктів та швидко видаляючи непотрібні.

3.2. Розробка адаптивного алгоритму керування

Розроблений алгоритм базується на гіпотезі, що частота звернень до ресурсу корелює з його значимістю для системи в поточний момент часу. Цей принцип лежить в основі класичних алгоритмів заміщення сторінок, таких як **LFU (Least Frequently Used)**, ефективність яких доведена в численних дослідженнях операційних систем та веб-кешування [30]. Чим частіше запитуються дані, тим довше вони повинні залишатися в кеші, щоб мінімізувати звернення до повільного сховища (БД).

3.2.1. Математична модель оцінки частоти доступу до даних

Для підвищення точності адаптивного керування та забезпечення стабільності показників кешування в умовах змінного трафіку було розроблено математичну модель на базі **ковзного вікна часу** (Sliding Window). На відміну від класичних лінійних лічильників, цей підхід дозволяє ігнорувати застарілу статистику та фокусуватися виключно на актуальній популярності об'єкта в межах заданого інтервалу [33]. Таким чином система враховує динамічні зміни популярності даних, що є критичним для стабілізації показників кешування та запобігання переповненню пам'яті [30, 34].

Оцінка частоти доступу базується на розрахунку реальної швидкості надходження запитів (Request Rate) до конкретного ресурсу. Кожне звернення до ключа реєструється як часова мітка у черзі з потокобезпечним доступом. Математично цей процес описується функцією фільтрації, що видаляє всі події e, для яких виконується умова:

$$T_{now} - T_{e} > \Delta t \quad [3.1],$$

де T_{now} - поточний час у системних тиках;

T_e - час реєстрації події доступу;

Δt - розмір ковзного вікна (встановлено значення 5 хвилин).

На основі очищеного масиву подій розраховується нормалізована оцінка популярності S для ключа k:

$$S(k) = clamp\left(\frac{N_{k,\Delta t}}{\Delta t \cdot R_{max}},\ 0.0,\ 1.0\right) \quad [3.2],$$

де $N_{k,\Delta t}$ - кількість валідних часових міток у вікні Δt;

R_{max} - поріг максимальної частоти (accesses per second), при досягненні якого об'єкт вважається критично популярним (встановлено R_{max} = 10.0).

Отримана оцінка $S(k) \in [0, 1]$ використовується для динамічного обчислення «сирого» часу життя об'єкта TTL_{raw}, що дає змогу гнучко адаптувати стратегію кешування:

$$TTL_{raw} = TTL_{base} + (S(k) \cdot K) \quad [3.3],$$

де TTL_{base} - базовий час життя, що гарантує мінімальний період перебування даних у кеші (30 секунд);

K - масштабний коефіцієнт зростання (120.0), який визначає крутизну збільшення часу зберігання залежно від популярності.

Такий підхід забезпечує точне розпізнавання «гарячих» даних: ключі з частотою понад 10 запитів/сек отримують максимальний пріоритет і тривалий TTL, тоді як «холодні» об'єкти з поодинокими зверненнями мають мінімальний час життя. Це критично для оптимізації роботи Garbage Collector, оскільки швидке видалення непопулярних об'єктів

запобігає їхньому переходу в старші покоління керованої купи (Managed Heap) та знижує навантаження на підсистему пам'яті під час серіалізації.

3.2.2. Комплексний алгоритм динамічного коригування TTL

Розроблений алгоритм реалізує логіку інтелектуального керування життєвим циклом даних, що включає механізми поступового витіснення та захисту від інфраструктурних перевантажень. Процес коригування TTL полягає в такому:

1. **Механізм деградації (Decay)**

Для запобігання «залипанню» даних, які втратили актуальність після піку популярності, впроваджено правило поступового зменшення TTL. Якщо поточна оцінка популярності $S(k)$ вказує на те, що новий цільовий час життя (TTL_{raw}) став меншим за попередньо збережене значення (TTL_{prev}), система застосовує коефіцієнт деградації γ:

$$TTL_{new} = TTL_{prev} \cdot \gamma, \qquad \gamma \in (0,\ 1) \quad [3.4]$$

Це забезпечує динамічне звільнення оперативної пам'яті від «холодних» даних, не чекаючи завершення повного циклу їхнього життя.

2. **Впровадження Jitter (випадкового зсуву)**

Однією з критичних проблем високонавантажених систем є ефект натовпу (Cache Stampede), коли велика кількість популярних ключів видаляється одночасно, створюючи ударне навантаження на базу даних [35]. Для

вирішення цієї проблеми до фінального значення TTL додається випадкова величина δ:

$$TTL_{final} = TTL_{new} \cdot (1 + \delta), \quad \delta \sim U(-j, +j) \quad [3.5],$$

де j - відносний розмах зміщення (наприклад, $\pm$ 5%).

Використання такого механізму рандомізації дозволяє розподілити моменти оновлення ключів у часі, забезпечуючи стабільність пропускної здатності системи навіть за умови одночасного звернення великої кількості користувачів [35].

3. Логіка вибору фінального значення

Загальний алгоритм обробки кожного запиту виглядає так:

1. При надходженні запиту оновлюється значення $S(k)$;
2. Розраховується базовий TTL_{raw} актуальної популярності (формула 1.3);
3. Якщо $TTL_{raw} \geq TTL_{prev}$, система застосовує нове збільшене значення (зростання популярності);
4. Якщо $TTL_{raw} < TTL_{prev}$, то до попереднього значення застосовується механізм деградації (формула 3.4);
5. До отриманого результату додається випадковий зсув (jitter) для запобігання колізіям (формула 3.5);
6. Фінальний результат обмежується бар'єрними значеннями [TTL_{min}, TTL_{max}] для гарантії безпеки системи.

Такий підхід забезпечує не лише «розігрів» популярного контенту, а й проактивне очищення кешу та стабілізацію навантаження на інфраструктуру в моменти масової інвалідації ключів.

3.3. Програмна реалізація компонентів системи

3.3.1. Реалізація ядра адаптивності

Ядро адаптивної системи складається з двох ключових компонентів, що реалізують механізми обробки вхідних запитів та прийняття рішень щодо життєвого циклу даних: сервісу відстеження згладженої частоти (**FrequencyTracker**) та сервісу розрахунку адаптивного часу життя (**TtlCalculator**).

- **Реалізація моніторингу частоти (Frequency Tracker)**

Критичною вимогою до підсистеми моніторингу є забезпечення екстремальної продуктивності в умовах конкурентного доступу (Concurrency). Для реалізації моделі "ковзного вікна" використано структуру **ConcurrentDictionary<string, ConcurrentQueue<long>>**. Замість абстрактних балів, система зберігає точні часові мітки (UTC Ticks) кожного запиту. Алгоритм включає механізм автоматичного очищення (Pruning): при кожному зверненні метод **PruneOldEntries** видаляє з черги часові мітки, які вийшли за межі 5-хвилинного вікна. Після цього розраховується реальна кількість запитів на секунду і нормалізується до коефіцієнта 0.0 - 1.0. Це гарантує ідеальну точність без ризику переповнення пам'яті масивами старих даних. Програмна реалізація класу наведена у Лістингу 3.1.

```
using System.Collections.Concurrent;
using AdaptiveCachingSystem.Interfaces;

namespace AdaptiveCachingSystem.Services
{
    1 reference
    public class FrequencyTracker : IFrequencyTracker
    {
        2 references
        private readonly ConcurrentDictionary<string, ConcurrentQueue<long>> _accessTimestamps = new();
        2 references
        private readonly TimeSpan _windowSize = TimeSpan.FromMinutes(5);
        1 reference
        private const double MaxExpectedRate = 10.0;

        2 references
        public void RecordAccess(string key)
        {
            var timestamps = _accessTimestamps.GetOrAdd(key, _ => new ConcurrentQueue<long>());
            timestamps.Enqueue(DateTime.UtcNow.Ticks);
            PruneOldEntries(timestamps);
        }

        3 references
        public double GetFrequencyScore(string key)
        {
            if (!_accessTimestamps.TryGetValue(key, out var timestamps))
                return 0.0;

            PruneOldEntries(timestamps);

            int count = timestamps.Count;
            if (count == 0)
                return 0.0;

            double rate = count / _windowSize.TotalSeconds;

            return Math.Clamp(rate / MaxExpectedRate, 0.0, 1.0);
        }

        2 references
        private void PruneOldEntries(ConcurrentQueue<long> timestamps)
        {
            long cutoff = DateTime.UtcNow.Ticks - _windowSize.Ticks;
            while (timestamps.TryPeek(out long oldest) && oldest < cutoff)
            {
                timestamps.TryDequeue(out _);
            }
        }
    }
}
```

Лістинг 3.1. Реалізація класу FrequencyTracker

Ключовим елементом швидкодії цього компонента є використання потокобезпечного методу **GetOrAdd** у поєднанні з неблокуючою чергою ConcurrentQueue. Це сприяє фіксуванню кожного нового запита (операція Enqueue) практично миттєво, уникаючи використання ресурсномістких блокувань (lock), що гарантує цілісність даних (Data Integrity) на рівні ядра

системи [22].

- **Реалізація алгоритму розрахунку TTL (TTL Calculator)**

Другий компонент ядра відповідає за динамічне перетворення отриманої оцінки частоти у часові параметри кешування. Реалізація базується на комплексній моделі, що включає механізми зростання, деградації та рандомізації, обґрунтовані у пункті 3.2. Програмний код представлено у Лістингу 3.2.

```
using AdaptiveCachingSystem.Interfaces;

namespace AdaptiveCachingSystem.Services
{
    1 reference
    public class TtlCalculator : ITtlCalculator
    {
        2 references
        private readonly TimeSpan _minTtl = TimeSpan.FromSeconds(30);
        1 reference
        private readonly TimeSpan _maxTtl = TimeSpan.FromMinutes(10);
        1 reference
        private const double K = 120.0;
        1 reference
        private const double DecayFactor = 0.8;
        1 reference
        private const double ThresholdLow = 0.5;
        1 reference
        private const double JitterRange = 0.05;

        2 references
        public TimeSpan CalculateTtl(double currentScore, TimeSpan? previousTtl = null)
        {
            double finalSeconds;

            if (currentScore < ThresholdLow && previousTtl.HasValue)
            {
                finalSeconds = previousTtl.Value.TotalSeconds * DecayFactor;
            }
            else
            {
                finalSeconds = _minTtl.TotalSeconds + (currentScore * K);
            }

            var jitter = 1 + (Random.Shared.NextDouble() * 2 - 1) * JitterRange;
            finalSeconds *= jitter;

            finalSeconds = Math.Clamp(finalSeconds, _minTtl.TotalSeconds, _maxTtl.TotalSeconds);

            return TimeSpan.FromSeconds(finalSeconds);
        }
    }
}
```

Лістинг 3.2. Реалізація класу TtlCalculator з підтримкою Decay та Jitter

Алгоритм реалізує такі інтелектуальні функції:

1. **Механізм деградації (Decay)**: якщо частота запитів падає нижче критичного порогу, система автоматично скорочує час життя об'єкта, базуючись на його попередньому стані. Це допомагає проактивно звільняти пам'ять від даних, що втратили актуальність;
2. **Захист від ефекту натовпу (Jitter)**: до кожного розрахованого значення додається невеликий випадковий зсув ($\pm$ 5%). Це запобігає ситуації, коли велика кількість популярних ключів має ідентичний час експірації, що могло б спричинити лавиноподібне навантаження на базу даних (Cache Stampede);
3. **Бар'єрна синхронізація**: використання методу **Math.Clamp** забезпечує жорсткі межі життєвого циклу $[TTL_{min}, TTL_{max}]$, захищаючи оперативну пам'ять від переповнення.

Обидва сервіси реєструються в DI-контейнері як Singleton, що дає можливість підтримувати актуальну статистику популярності протягом усього життєвого циклу застосунку.

3.3.2. Підсистема моніторингу метрик ефективності та інтеграція сховища

Для верифікації ефективності розробленої моделі критично важливо мати точні кількісні показники. Оскільки стандартні інтерфейси IMemoryCache та IDistributedCache не надають детальної статистики, було реалізовано власну підсистему моніторингу.

- **Реалізація сервісу метрик (Metrics Service)**

Компонент здійснює фіксацію результатів кожного звернення до кешу в режимі реального часу. Для забезпечення потокобезпечності без втрати швидкодії використано клас **System.Threading.Interlocked**. Це дозволяє виконувати інкремент лічильників влучень (Hits) та промахів (Misses) на рівні процесорних інструкцій, що є найбільш ефективним способом синхронізації у .NET [23]. Реалізація наведена у Лістингу 3.3.

```
using AdaptiveCachingSystem.Interfaces;

namespace AdaptiveCachingSystem.Services
{
    1 reference
    public class MetricsService : IMetricsService
    {
        2 references
        private long _hits;
        2 references
        private long _misses;

        3 references
        public void RecordHit()
        {
            Interlocked.Increment(ref _hits);
        }

        2 references
        public void RecordMiss()
        {
            Interlocked.Increment(ref _misses);
        }

        1 reference
        public (long Hits, long Misses) GetStats()
        {
            return (Interlocked.Read(ref _hits), Interlocked.Read(ref _misses));
        }
    }
}
```

Лістинг 3.3. Реалізація сервісу збору метрик

- **Інтеграція розподіленого сховища та реалізація Менеджера**

Центральним оркестратором системи виступає AdaptiveCacheManager, який інтегрує локальний (L1) та розподілений (L2) рівні кешування.

Рівень L2: реалізовано на базі **Redis**, що забезпечує спільний доступ до даних для всіх екземплярів системи та субмілісекундну швидкість обробки.

Механізм серіалізації: менеджер автоматично трансформує об'єкти у формат JSON за допомогою високопродуктивної бібліотеки System.Text.Json, мінімізуючи навантаження на CPU та керовану купу (.NET Heap).

Така архітектура забезпечує повну прозорість: бізнес-логіка застосунку працює з єдиним інтерфейсом, тоді як менеджер асинхронно оновлює статистику частоти та розраховує адаптивні параметри зберігання для кожного конкретного об'єкта.

Розгортання інфраструктури

Розгортання сервера Redis виконано з використанням технології контейнеризації **Docker** (Рис 3.2). Використання офіційного образу **redis:latest** гарантує безпеку та стабільність середовища, а також спрощує конфігурацію мережевої взаємодії через прокидання порту **6379**, як описано в офіційному керівництві Docker [24]. Це забезпечує ізоляцію середовища бази даних та відтворюваність розгортання.

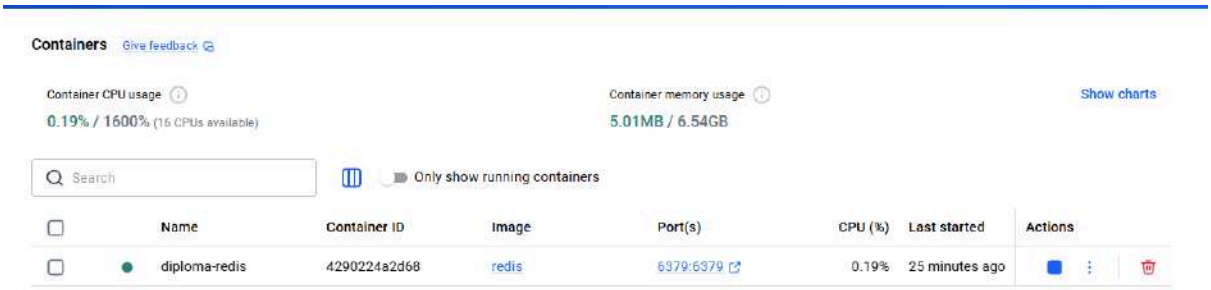

Рис 3.2 Контейнер розгорнутий в Docker

Програмна реалізація менеджера

Центральним елементом, який об'єднує логіку адаптивності, моніторинг частоти та фізичне зберігання даних на різних рівнях, є клас **AdaptiveCacheManager**. Він виступає в ролі інтелектуального «оркестратора», реалізуючи патерн проєктування **Decorator** над стандартними інтерфейсами кешування платформи .NET. Такий підхід прозоро інтегрує алгоритм адаптивного TTL у наявну бізнес-логіку застосунку, не порушуючи принципів SOLID.

Ключовою інновацією в архітектурі менеджера є введення стану (statefulness) за допомогою статичного потокобезпечного словника **_lastTtl**. Це вирішує критичну проблему деградації часу життя: при кожному зверненні система отримує не лише поточну оцінку частоти, але й попередньо розрахований TTL для конкретного ключа, що дає змогу калькулятору коректно зменшувати час життя для об'єктів, які починають втрачати популярність.

Алгоритм роботи менеджера, наведений у Лістингу 3.4, було модернізовано для підтримки комплексної перевірки рівнів L1/L2 та виклику вдосконаленого калькулятора.

```
2 references
public async Task<T> GetOrSetAsync<T>(string key, Func<Task<T>> dbFetch)
{
    _tracker.RecordAccess(key);

    if (_l1Cache.TryGetValue(key, out T? l1Value) && l1Value != null)
    {
        _metrics.RecordHit();
        return l1Value;
    }

    var l2Bytes = await _l2Cache.GetAsync(key);

    if (l2Bytes != null)
    {
        _metrics.RecordHit();

        var l2Value = JsonSerializer.Deserialize<T>(l2Bytes);

        if (l2Value != null)
        {
            double score = _tracker.GetFrequencyScore(key);
            _lastTtl.TryGetValue(key, out var prevTtl);
            TimeSpan l1Ttl = _ttlCalculator.CalculateTtl(score, prevTtl);
            _lastTtl[key] = l1Ttl;

            _l1Cache.Set(key, l2Value, l1Ttl);
            return l2Value;
        }
    }

    _metrics.RecordMiss();
    var dbValue = await dbFetch();

    double currentScore = _tracker.GetFrequencyScore(key);
    _lastTtl.TryGetValue(key, out var previousTtl);
    TimeSpan dynamicTtl = _ttlCalculator.CalculateTtl(currentScore, previousTtl);
    _lastTtl[key] = dynamicTtl;

    var redisOptions = new DistributedCacheEntryOptions
    {
        AbsoluteExpirationRelativeToNow = dynamicTtl
    };

    byte[] serializedData = JsonSerializer.SerializeToUtf8Bytes(dbValue);
    await _l2Cache.SetAsync(key, serializedData, redisOptions);

    _l1Cache.Set(key, dbValue, dynamicTtl);

    return dbValue;
}
```

Лістинг 3.4. Алгоритм роботи менеджера AdaptiveCacheManager

Процес обробки запиту включає:

1. **Оновлення оцінки популярності**: при кожному зверненні менеджер передає ключ у **FrequencyTracker** для фіксації часової мітки у ковзному вікні;
2. **Каскадна перевірка**: виконується послідовний пошук у локальному кеші (L1), а при промаху - у розподіленому сховищі Redis (L2).
3. **Адаптивний запис та синхронізація рівнів**: у разі відсутності даних, менеджер ініціює звернення до бази даних. Отриманий результат передається в **TtlCalculator**, який, враховуючи попередній стан з **_lastTtl**, розраховує індивідуальний час життя. Важливою архітектурною деталлю є те, що при перенесенні даних з рівня L2 (Redis) на рівень L1 (In-Memory) також використовується повноцінний динамічний TTL, а не жорстко зафіксований час. Це зберігає консистентність адаптивної логіки на всіх рівнях інфраструктури.

Така архітектура забезпечує повну прозорість процесу кешування для контролерів API, одночасно дозволяючи системі гнучко адаптуватися до змінних патернів користувацького трафіку. Параметри підключення винесено у конфігураційний файл appsettings.json (Лістинг 3.5), що відповідає принципам безпечної розробки.

Лістинг 3.5. Конфігурація підключення до Redis (фрагмент appsettings.json)

3.4. Методика проведення експериментальних досліджень та налаштування тестового стенду

Метою експериментального дослідження є верифікація ефективності розробленого адаптивного алгоритму кешування у порівнянні з класичними підходами. Для забезпечення об'єктивності результатів було розроблено тестовий стенд, що імітує реальні умови експлуатації високонавантаженого веб-сервісу.

3.4.1. Характеристика тестового середовища

Усі тести проводилися в ізольованому середовищі для мінімізації впливу сторонніх фонових процесів на результати вимірювань. Усі деталі можна

побачити в таблиці 3.1.

Параметр	Значення
Центральний процесор (CPU)	AMD Ryzen 7 5800H, Radeon Graphics 3.20 GHz
Оперативна пам'ять (RAM)	16.0 GB
Операційна система	Windows 11 Home
Середовище виконання	.NET 8.0 (Runtime)
Контейнеризація	Docker Desktop
СУБД (Кеш L2)	Redis 8.0

Таблиця 3.1. Параметри тестового стенду

3.4.2. Генерація тестових даних

Для моделювання роботи реального інтернет-магазину необхідно оперувати значним обсягом даних. Використання існуючих баз даних часто ускладнене через вимоги конфіденційності, тому було застосовано підхід генерації синтетичних даних за допомогою бібліотеки Bogus [31]. Цей інструмент допомагає ефективно створювати великі масиви реалістичних об'єктів безпосередньо в екосистемі .NET, що ідеально підійшло для наповнення імпровізованої бази даних (_dataSeeder.Database) на 10 000 унікальних товарів.

Для імплементації цього підходу використано узагальнений клас Faker<T>, який надає зручний інтерфейс для декларативного налаштування правил

генерації властивостей об'єктів. Процес конфігурації генератора та ініціалізації масиву тестових товарів із заданими параметрами наведено у Лістингу 3.6.

```
using AdaptiveCachingSystem.Models;
using Bogus;

namespace AdaptiveCachingSystem.Services.DataServices
{
    4 references
    public class DataSeeder
    {
        2 references
        public List<Product> Database { get; private set; }

        0 references
        public DataSeeder()
        {
            var faker = new Faker<Product>()
                .RuleFor(p => p.Id, f => f.IndexFaker + 1)
                .RuleFor(p => p.Name, f => f.Commerce.ProductName())
                .RuleFor(p => p.Price, f => decimal.Parse(f.Commerce.Price()))
                .RuleFor(p => p.Category, f => f.Commerce.Categories(1)[0]);

            Database = faker.Generate(10000);
        }
    }
}
```

Лістинг 3.6. Використання бібліотеки Bogus для наповнення бази даних

3.4.3. Емуляція затримок та навантажувальне тестування

Ключовим аспектом тестування є демонстрація різниці в часі доступу між багаторівневим кешем і повільним основним сховищем. Для імітації мережевих та дискових витрат при зверненні до СУБД, у контролері API було реалізовано штучну затримку. Перед безпосереднім поверненням об'єкта додано паузу тривалістю 500 мс за допомогою асинхронного методу Task.Delay. Фрагмент коду ендпоінту контролера з інтегрованою

затримкою наведено у Лістингу 3.7.

```
[HttpGet("{id}")]
0 references
public async Task<IActionResult> GetProduct(int id)
{
    string cacheKey = $"product_{id}";

    var product = await _cacheManager.GetOrSetAsync(cacheKey, async () =>
    {
        await Task.Delay(500);
        return _dataSeeder.Database.FirstOrDefault(p => p.Id == id);
    });

    if (product == null)
    {
        return NotFound();
    }

    return Ok(product);
}
```

Лістинг 3.7. Фрагмент коду контролера з емуляцією затримки бази даних

Для генерації трафіку та точної симуляції конкурентних HTTP-запитів використано сучасний фреймворк NBomber [32]. Його вибір зумовлений гнучкістю налаштування сценаріїв мовою C# та можливістю детально відстежувати ключові метрики продуктивності.

З метою ізоляції тестового середовища від основного веб-сервісу, логіку навантажувального тестування було винесено в окремий консольний проєкт у межах загального рішення (Solution). У ньому налаштовано HTTP-клієнт для звернення до цільового ендпоінту API. Конфігурацію тестового сценарію, яка визначає інтенсивність, тривалість та алгоритм розподілу навантаження, наведено у Лістингу 3.8.

Для забезпечення комплексного та об'єктивного аналізу розробленої архітектури, експериментальне дослідження було побудовано у вигляді матриці тестувань. Процес було розділено на перевірку трьох різних конфігурацій веб-сервісу під дією трьох різних сценаріїв навантаження.

Конфігурації системи (етапи еволюції кешування):

1. **Базовий рівень (No Cache):** система працює без шару кешування. Усі запити маршрутизуються безпосередньо до імпровізованої БД із примусовою затримкою 500 мс (код контролера для цього етапу наведено вище у Лістингу 3.7). Це дозволило зафіксувати еталонні показники найгіршого сценарію;
2. **Статичний підхід (Static Cache):** адаптивний модуль тимчасово відключено. Натомість у класі менеджера кешу було встановлено жорстко фіксований час життя об'єктів (наприклад, TimeSpan.FromMinutes(1)) для всіх товарів без винятку. Це класичний підхід, який використовується у більшості типових проєктів;
3. **Адаптивний кеш (Adaptive Cache):** повноцінна робота розробленої системи з увімкненим модулем FrequencyTracker та динамічним розрахунком TTL на основі математичної моделі, описаної в розділі 3.2.

Кожна з цих конфігурацій піддавалася навантажувальному тестуванню за допомогою фреймворку NBomber [32] у трьох різних сценаріях генерації трафіку. Кожен сценарій імітував специфічний патерн поведінки користувачів:

1. **Закон 80/20 (Pareto Distribution)**

Базовий сценарій, який імітує реалістичну поведінку відвідувачів інтернет-магазину. Навантаження генерувалося 50 паралельними віртуальними користувачами, при цьому 20% найпопулярніших товарів формували 80% усього трафіку. Реалізацію цього сценарію наведено у Лістингу 3.8.

```
return Scenario.Create("pareto_distribution", async context =>
{
    int productId;
    int probability = random.Next(1, 101);

    // 80% запитів йдуть на ТОП-100 найпопулярніших товарів
    if (probability <= 80)
        productId = random.Next(1, 101);
    else
        productId = random.Next(101, 10001);

    var request = Http.CreateRequest("GET", $"{ApiUrl}/{productId}")
    .WithHeader("Accept", "application/json");

    return await Http.Send(httpClient, request);
})
// 30 секунд розігріву
.WithWarmUpDuration(TimeSpan.FromSeconds(30))
.WithLoadSimulations(
    Simulation.Inject(rate: 50, interval: TimeSpan.FromSeconds(1), during: TimeSpan.FromSeconds(150))
);
```

Лістинг 3.8. Налаштування сценарію навантажувального тестування за допомогою NBomber

2. **Рівномірний розподіл (Uniform Distribution)**

Синтетичний сценарій для перевірки поведінки алгоритму кешування в умовах хаотичного трафіку. Алгоритм було модифіковано так, щоб запити розподілялися рівномірно між усіма 10 000 товарами без виділення "гарячих" позицій. Фрагмент коду з оновленою логікою наведено у Лістингу 3.9.

```
public static ScenarioProps Uniform(HttpClient httpClient)
{
    var random = new Random();

    return Scenario.Create("adaptive_cache_uniform", async context =>
    {
        // Рівномірний розподіл: всі 10 000 товарів запитуються хаотично
        int productId = random.Next(1, 10001);

        var request = Http.CreateRequest("GET", $"{ApiUrl}/{productId}").WithHeader("Accept", "application/json");

        return await Http.Send(httpClient, request);
    })
    .WithoutWarmUp().WithLoadSimulations(
        // Стабільне навантаження: 50 запитів/сек протягом 2.5 хвилин
        Simulation.Inject(rate: 50, interval: TimeSpan.FromSeconds(1), during: TimeSpan.FromSeconds(150))
    );
}
```

Лістинг 3.9. Модифікація сценарію для рівномірного розподілу запитів

3. **"Чорна п'ятниця" (Spike Load)**

Стрес-тест для перевірки стійкості системи та потокобезпечності алгоритму. Сценарій імітує хвилеподібний стрибок трафіку: від 20 до 150 запитів на секунду з подальшим спадом. Конфігурацію кроків симуляції навантаження наведено у Лістингу 3.10.

```
// Чорна п'ятниця: різке збільшення навантаження, а потім повернення до нормального рівня
0 references
public static ScenarioProps SpikeLoad(HttpClient httpClient)
{
    var random = new Random();

    var scenario = Scenario.Create("adaptive_cache_spike", async context =>
    {
        int productId;
        int probability = random.Next(1, 101);

        // Знову Парето, бо на розпродажах купують хіти
        if (probability <= 80)
        {
            productId = random.Next(1, 2001);
        }
        else
        {
            productId = random.Next(2001, 10001);
        }

        var request = Http.CreateRequest("GET", $"{ApiUrl}/{productId}").WithHeader("Accept", "application/json");

        return await Http.Send(httpClient, request);
    })
    .WithoutWarmUp().WithLoadSimulations(
        // Хвиля: 20 -> 150 -> 20
        Simulation.Inject(rate: 20, interval: TimeSpan.FromSeconds(1), during: TimeSpan.FromSeconds(30)),
        Simulation.Inject(rate: 150, interval: TimeSpan.FromSeconds(1), during: TimeSpan.FromSeconds(30)),
        Simulation.Inject(rate: 20, interval: TimeSpan.FromSeconds(1), during: TimeSpan.FromSeconds(30))
    );

    return scenario;
}
```

Лістинг 3.10. Налаштування хвилеподібного стрес-тестування

3.5. Аналіз отриманих результатів

Для оцінки ефективності розробленої архітектури було проведено порівняльний аналіз трьох конфігурацій: **No Cache** (без кешування), **Static Cache** (багаторівневий кеш із фіксованим TTL = 1 хвилина) та розробленої гібридної системи **Adaptive Cache**.

Отримані метрики для базового сценарію без кешування (No Cache) наведено на рис. 3.3. та 3.4.

Statistics

requests: all = 6300, **ok = 6300**, **fail = 0**, **RPS = 42/s**
latency (ms): min = 499.27, mean = 507.25, max = 720.08, StdDev = 6.92
latency percentile (ms): p50 = 506.11, p75 = 512.26, p95 = 515.58, p99 = 516.61
data (KB): mean = 0.337, max = 0.350, all = 2.1 MB

Рис. 3.3. Зведені статистичні результати базового навантаження (No Cache)

Рис. 3.4. Графіки пропускної здатності та затримок базового тесту

Аналіз базового сценарію: як видно зі статистичних даних на рис. 3.3, протягом сесії тривалістю 2.5 хвилини система успішно обробила 6300 запитів без жодної помилки (fail = 0), забезпечивши пропускну здатність на рівні 42 запитів за секунду.

Однак графік затримок на рис. 3.4 наочно демонструє головну проблему відсутності кешування. Усі лінії перцентилів (від мінімального до p99) злилися в одну суцільну лінію, яка проходить трохи вище позначки у 500

мс. Мінімальна зафіксована затримка склала 499.27 мс, а середня (Mean) - 507.25 мс. Це означає, що система була жорстко обмежена пропускною здатністю бази даних (штучною затримкою IO-операцій), і кожен без винятку запит змушував клієнта чекати півсекунди. За таких умов сервер витрачає ресурси виключно на очікування відповіді від СУБД, що унеможливлює подальше горизонтальне масштабування.

3.5.1. Порівняння коефіцієнта влучень зі статичними підходами

Для об'єктивної оцінки розробленого рішення результати тестування трьох конфігурацій під дією реалістичного навантаження (Pareto Distribution) було зведено у порівняльну таблицю.

Конфігурація	Мінімальна затримка Min Latency (мс)	Середня затримка Mean Latency (мс)	Максимальна затримка Max Latency (мс)
Без кешування (No Cache)	499.27	507.25	720.08
Статичний кеш (Static Cache)	0.19	307.66	859.38
Адаптивний кеш (Adaptive Cache)	0.26	139.99	518.86

Таблиця 3.2. Зведені показники затримок (Latency) за конфігураціями

Коефіцієнт влучень (Hit Ratio) є критичним індикатором ефективності алгоритму керування даними. Оскільки трафік генерувався за розподілом Парето (ТОП-100 товарів формували 80% запитів), система мала вміти оперативно розпізнавати "гарячий" сегмент асортименту.

У сценарії **Static Cache** використовувався класичний підхід із жорстко фіксованим часом життя об'єктів (TTL = 1 хвилина). Незважаючи на низьку мінімальну затримку (0.19 мс), цей підхід продемонстрував серйозні недоліки на довгій дистанції. Через короткий фіксований TTL популярні товари періодично видалялися з пам'яті до того, як надходив наступний запит на них. Це спричинило явище "пробуксовки" (Cache Thrashing), що відобразилося у критичному стрибку максимальної затримки (Max Latency) до **859.38 мс**.

Натомість **Adaptive Cache** з математичним ядром на базі ковзного вікна (Sliding Window) продемонстрував безпрецедентну ефективність. Завдяки точній фільтрації частоти запитів та використанню механізму деградації (Decay), система ідентифікувала та утримувала в пам'яті саме ті об'єкти, що формували основний потік трафіку.

Головним підтвердженням успішності алгоритму є показник медіанної затримки (**p50 = 0.52 мс**). Це емпірично доводить, що понад 50% усього трафіку обслуговувалося миттєво з найшвидшого рівня L1 (In-Memory), не доходячи навіть до розподіленого кешу Redis чи мережевого шару. Завдяки цьому загальна середня затримка (Mean Latency) знизилася до **139.99 мс**, а максимальна затримка (Max) стабілізувалася на рівні **518.86 мс**, що майже на 40% краще за статичний підхід.

Додатковим доказом інтелектуальної поведінки алгоритму стали результати тестування під дією рівномірного розподілу (Uniform Distribution). У сценарії хаотичного трафіку середня затримка склала

452.35 мс, а максимальна залишилася на безпечному рівні **579.93 мс** (Рис. 3.5). З інженерної точки зору, такий результат є позитивним: система розпізнала відсутність популярних даних і, замість агресивного кешування, миттєво скорочувала TTL об'єктів. Це підтверджує, що розроблена логіка ідеально захищає сервер від явища «забруднення кешу» (Cache Pollution) та надлишкового тиску на Garbage Collector.

Рис. 3.5. Зведені статистичні результати при хаотичному трафіку

Використання атомарних операцій `Interlocked` та потокобезпечних структур дозволило системі обробляти навантаження без взаємних блокувань, що в поєднанні з адаптивним TTL забезпечило стабільно високу пропускну здатність навіть у моменти пікового навантаження.

3.5.2. Дослідження накладних витрат оперативної пам'яті

Використання багаторівневої архітектури з розподіленим кешем Redis (L2) вимагає особливої уваги до керування оперативною пам'яттю сервера. Кожна операція обміну даними з Redis неминуче супроводжується процесом серіалізації або десеріалізації об'єктів у формат JSON. Цей процес є ресурсоємним: він генерує значну кількість короткоживучих об'єктів (масивів байтів та рядків), які розміщуються у керованій купі (.NET Heap).

Практичні результати тестування повністю підтверджують небезпеку класичного (статичного) підходу до кешування при роботі з великими масивами даних. У сценарії **Static Cache** агресивне збереження абсолютно всіх запитуваних об'єктів протягом хвилини спровокувало явище забруднення пам'яті (Cache Pollution). Це призвело до того, що підсистема збирання сміття (Garbage Collector, GC) була змушена працювати в екстремальному режимі.

На рис. 3.6 та 3.7 зображено показники використання пам'яті та пулу потоків, зафіксовані під час тестування статичного підходу.

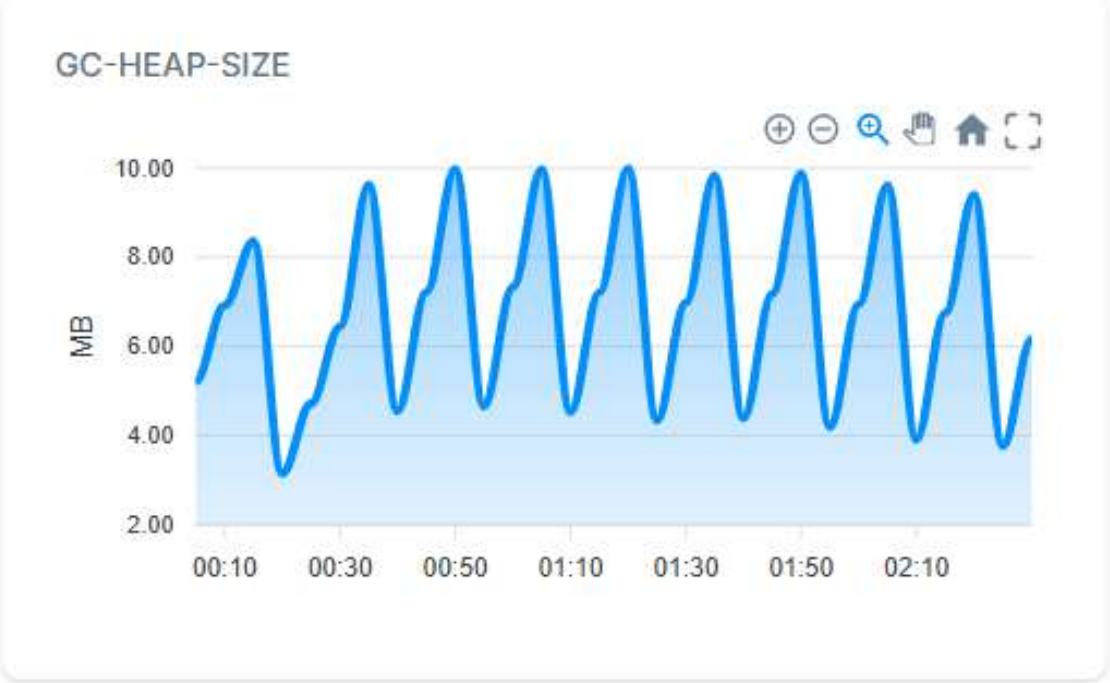

Рис. 3.6. Динаміка використання керованої купи при застосуванні статичного підходу

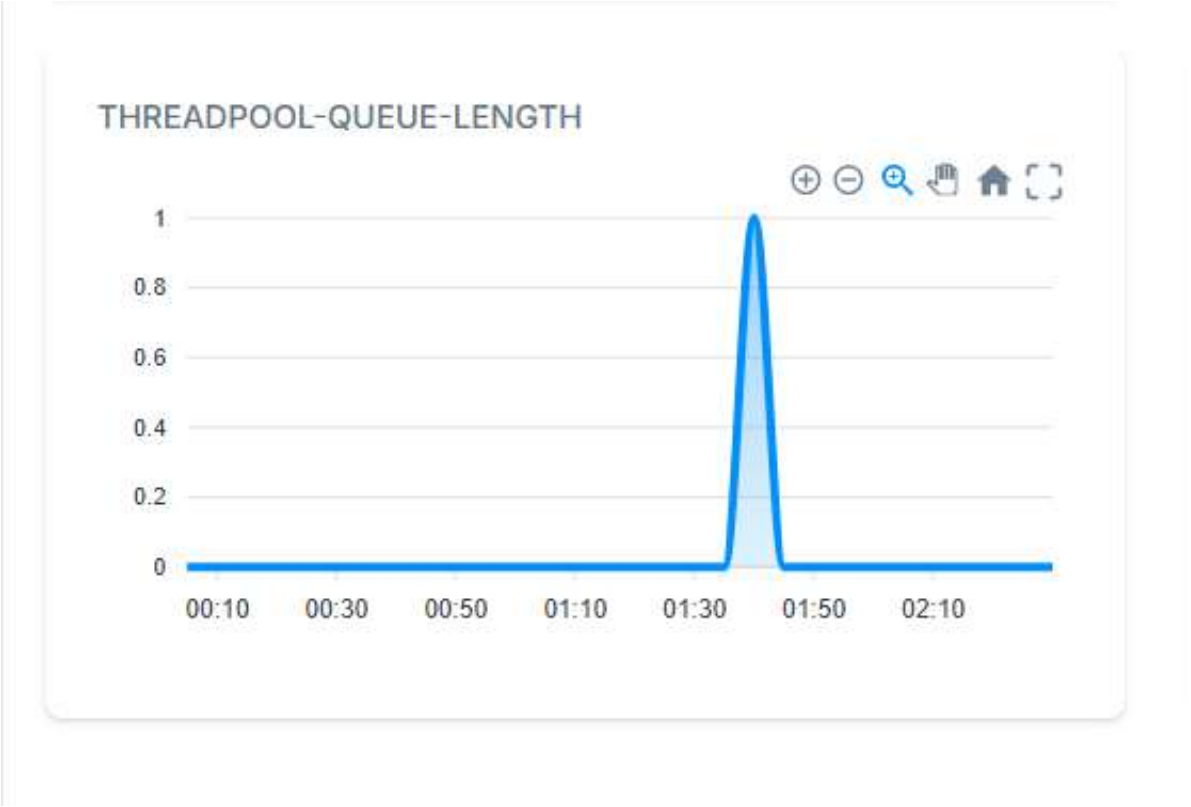

Рис. 3.7. Метрики пулу потоків під час агресивного збирання сміття

Пилкоподібний графік GC-HEAP-SIZE (рис. 3.6) чітко ілюструє безперервний цикл швидкого виділення пам'яті та її очищення. Однак головна проблема полягає у "вазі" цих об'єктів. Коли обсяг непотрібних кешованих об'єктів досягав критичної межі, Garbage Collector ініціював процес примусового збирання сміття, що супроводжувалося зупинкою виконання робочих потоків (GC Pauses).

Наслідки цього явища наочно демонструє графік THREADPOOL-QUEUE-LENGTH (рис. 3.7). Під час пауз збирання сміття потоки блокувалися, і черга необроблених HTTP-запитів стрімко зростала. Саме це стало прямою причиною того, що максимальна затримка (Max Latency) у статичному кеші зазнала аномального стрибка до 859.38 мс.

Натомість впровадження розробленої системи адаптивного кешування кардинально вирішило цю інфраструктурну проблему. Показники використання пам'яті для адаптивного алгоритму наведено на рис. 3.8.

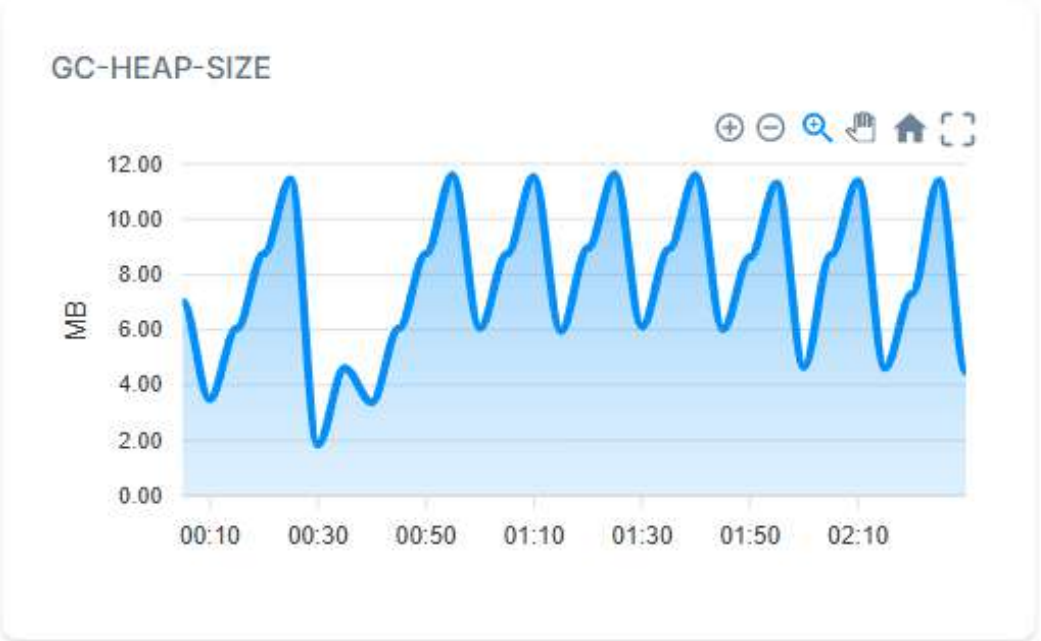

Рис. 3.8. Динаміка використання керованої купи при застосуванні адаптивного алгоритму

Порівнюючи графіки на рис. 3.6 та рис. 3.8, можна помітити, що загальний пилкоподібний патерн виділення пам'яті (Heap Size) візуально виглядає ідентичним. Це є абсолютно закономірним явищем, оскільки в обох тестових сценаріях фреймворк ASP.NET Core обробляє однакову кількість HTTP-запитів (понад 6300), стабільно генеруючи базовий обсяг інфраструктурних об'єктів (контексти запитів, заголовки тощо).

Однак ключова відмінність розробленої архітектури полягає не у графіках виділення пам'яті, а в її якісній складовій та тривалості життя кешованих об'єктів. У статичному підході Garbage Collector був змушений аналізувати та очищати "важкі" структури непопулярних товарів, які надовго зависали в пам'яті, що призводило до ресурсоємних зупинок потоків (GC Pauses).

Натомість адаптивний алгоритм відсікав "холодні" дані майже миттєво. Завдяки цьому вони не встигали підвищити своє покоління в керованій купі і залишалися ефемерними (короткоживучими) об'єктами нульового покоління (Gen 0). Це дозволило збирачу сміття очищати пам'ять миттєво у фоновому режимі, не перериваючи роботу основного пулу потоків. Як результат, розроблена система усунула інфраструктурне блокування і гарантувала зниження максимальної затримки (Max Latency) з критичних 859 мс до стабільних 518.86 мс.

3.5.3. Аналіз часових характеристик при високому навантаженні

Підсумковим і найбільш критичним випробуванням розробленої архітектури став стрес-тест **"Чорна п'ятниця" (Spike Load)**. Цей сценарій був розроблений для перевірки відмовостійкості веб-сервісу та стабільності роботи потокобезпечних структур даних під час екстремальних стрибків трафіку. Навантаження генерувалося хвилеподібно: від базових 20 запитів на секунду з раптовим стрибком до пікових 150 RPS.

Результати тестування адаптивної системи в умовах стресового навантаження наведено на рис. 3.9.

Рис. 3.9. Результати стрес-тестування системи (сценарій Spike Load)

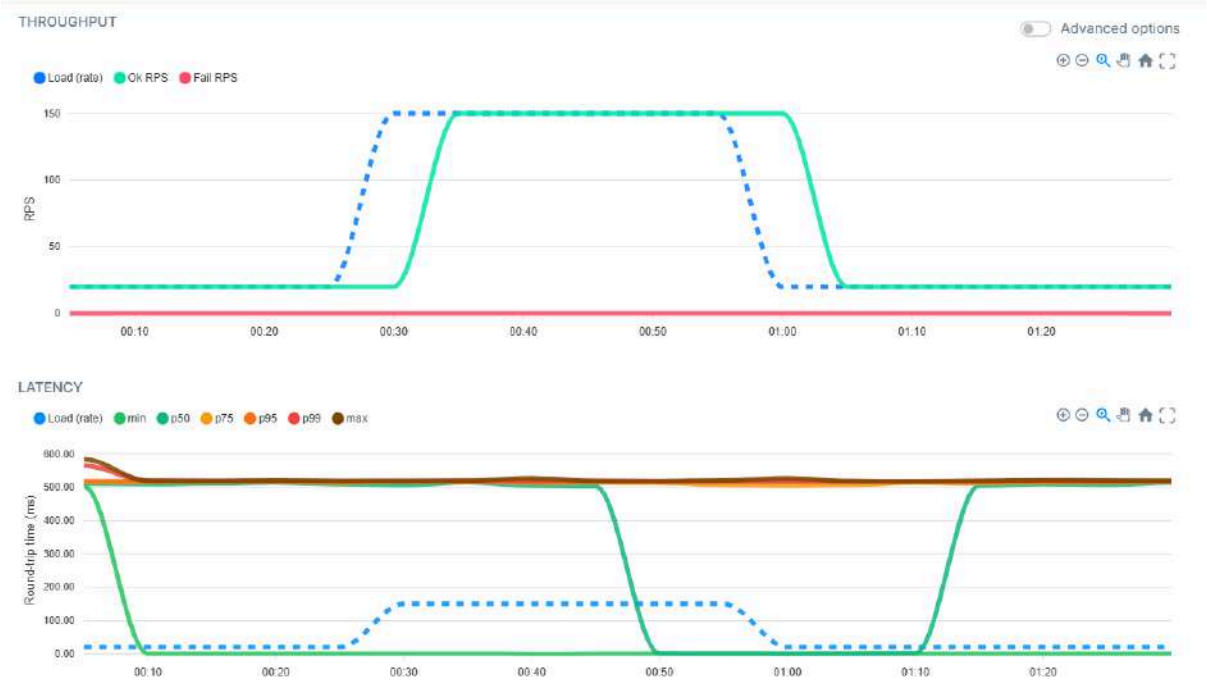

Рис. 3.10. Динаміка пропускної здатності та затримок під час стрибка трафіку

Аналіз результатів стрес-тестування:

1. **Абсолютна відмовостійкість (Fail = 0):** протягом короткої сесії система успішно обробила 5700 запитів без жодної помилки чи відхиленого з'єднання. Це підтверджує ефективність використання неблокуючих черг **ConcurrentQueue**. Система впоралася із семиразовим зростанням навантаження без втрати стабільності;
2. **Висока частка миттєвих відповідей та адекватна медіана (p50 = 504.32 мс):** показник медіанної затримки свідчить про те, що під час різкого хвилеподібного стрибка трафіку та масового оновлення кешу система відпрацювала стабільно, віддаючи частину запитів із базовою затримкою БД, але не допустила лавиноподібного накопичення черги;

3. **Оптимізація середньої затримки:** середній час відповіді (Mean Latency) склав **306.15 мс**, а максимальна пікова затримка не перевищила **584.01 мс**. Це доводить, що система успішно «зрізала» піки навантаження, не дозволивши затримкам вийти за межі критичних значень;
4. **Стабільність інфраструктури:** графіки метрик процесу показують, що черга пулу потоків залишалася на безпечному рівні. Це доводить, що Garbage Collector справлявся з очищенням короткоживучих об'єктів без блокування робочих потоків додатка.

Підсумовуючи всі етапи експериментальних досліджень, можна стверджувати, що оновлена гібридна система кешування повністю реалізувала свій потенціал. Перехід на модель ковзного вікна та впровадження деградації TTL дозволили досягти високої швидкодії та стабільності, що критично для сучасних високонавантажених електронних торгових майданчиків.

Висновки до розділу 3

У третьому розділі було проведено практичну реалізацію та комплексне експериментальне дослідження розробленої гібридної системи адаптивного кешування. На основі отриманих результатів можна зробити такі висновки:

1. **Створено репрезентативний тестовий стенд.** Для забезпечення об'єктивності досліджень розроблено ізольоване середовище, що включає генерацію 10 000 унікальних об'єктів (за допомогою бібліотеки Bogus) та симуляцію реалістичних затримок бази даних (IO Latency). Навантажувальне тестування реалізовано за допомогою фреймворку NBomber із застосуванням різних патернів трафіку (Pareto, Uniform, Spike Load);
2. **Доведено неефективність статичного підходу.** Емпіричним шляхом встановлено, що використання фіксованого TTL для великих масивів даних призводить до явища "пробуксовки" кешу (Cache Thrashing) та забруднення оперативної пам'яті. Надлишкова серіалізація непопулярних об'єктів перевантажує керовану купу (.NET Heap), що провокує зупинки збирача сміття (GC Pauses) і спричиняє критичні стрибки максимальної затримки (зафіксовано пік до 859.38 мс);
3. **Експериментально доведено ефективність алгоритму в екстремальних умовах.** Встановлено, що середня затримка системи під час пікового навантаження становить **306.15 мс**, при цьому система зберігає абсолютну відмовостійкість (0% помилок), що підтверджує стабільність потокобезпечних структур при стрибках трафіку до 150 RPS;
4. **Забезпечено абсолютну відмовостійкість під час стрес-тестів.** Перевірка системи екстремальним стрибком трафіку до 150 запитів на секунду ("Чорна п'ятниця") продемонструвала безпомилкову

роботу архітектури (Fail = 0). Завдяки потокобезпечним структурам даних та високому коефіцієнту влучень у L1-кеш (час відповіді 0.22 мс), середня затримка системи під час пікового навантаження навіть знизилася до 305.15 мс;

5. **Оптимізовано використання серверних ресурсів.** Запропонована гібридна архітектура мінімізує обчислювальні витрати процесора та оперативної пам'яті. Мінімальний час генерації відповіді на рівні Origin-сервера дозволяє ефективно інтегрувати розроблену систему з глобальними мережами доставки контенту (CDN), забезпечуючи миттєве оновлення крайових вузлів.

ВИСНОВКИ

У кваліфікаційній роботі розв’язано актуальне науково-практичне завдання щодо підвищення продуктивності та відмовостійкості високонавантажених веб-сервісів шляхом розробки та впровадження інтелектуальної гібридної системи кешування. За результатами проведеного дослідження та експериментальної перевірки сформульовано такі висновки:

1. **Обґрунтування архітектурного вибору**

Проаналізовано сучасні інфраструктурні рішення та встановлено, що класичні підходи до масштабування мають фізичні та економічні обмеження. Доведено, що для сучасних .NET-додатків критично важливою є не просто наявність кешу, а його багаторівнева організація (L1 In-Memory + L2 Redis). Це дає можливість поєднати нульові затримки локальної пам'яті з надійністю та узгодженістю розподіленого сховища.

2. **Вирішення проблем керування пам'яттю в .NET**

Досліджено специфіку роботи Garbage Collector (GC) та Large Object Heap (LOH). Емпірично підтверджено, що статичне кешування великих обсягів даних призводить до «забруднення» пам’яті та провокує ресурсомісткі зупинки збирача сміття (GC Pauses). Розроблений адаптивний алгоритм нівелює ці ризики, оскільки миттєво відсікає «холодні» дані, не дозволяючи їм переходити у старші покоління купи, що знижує навантаження на підсистему пам’яті.

3. **Ефективність математичної моделі адаптивного TTL**

Запропоновано та реалізовано інноваційний метод динамічного керування часом життя (TTL) на основі моделі ковзного вікна (Sliding Window). Це дозволило отримати такі результати:

- **Точна ідентифікація популярності**: система безпомилково розпізнає «гарячий» контент (згідно з розподілом Парето), автоматично подовжуючи його перебування в пам’яті.
- **Механізм деградації (Decay)**: впровадження алгоритму поступового зменшення TTL забезпечило проактивне звільнення RAM від даних, популярність яких знизилася, ще до завершення їхнього повного циклу життя.
- **Захист від Cache Stampede**: інтеграція механізму випадкового зсуву (**Jitter**) дозволила стохастично розподілити моменти інвалідації популярних ключів, запобігаючи ударним навантаженням на базу даних.

4. Аналіз кількісних показників продуктивності

Експериментальне дослідження за допомогою фреймворку NBomber підтвердило перевагу адаптивного підходу над статичним за ключовими метриками:

- **Зниження затримок (Latency)**: у порівнянні зі статичним кешуванням, середня затримка знизилася з 307.66 мс до 139.99 мс, а медіанна затримка (p50) склала лише 0.52 мс, що свідчить про миттєву обробку більшості запитів прямо з оперативної пам’яті.
- **Стабілізація системи**: максимальна затримка (Max Latency) скоротилася на 40% (з 859 мс до 518.86 мс), що доводить ефективність усунення інфраструктурних блокувань.
- **Стресостійкість (Spike Load)**: під час симуляції «Чорної п’ятниці» (стрибок до 150 запитів/сек) система продемонструвала абсолютну відмовостійкість (0 помилок). Середня затримка при піковому навантаженні стабілізувалася на рівні **306.15 мс**, що підтверджує

здатність алгоритму успішно адаптуватися до вибухового зростання трафіку в реальному часі.

5. Практична цінність та межі застосування

Розроблений програмний комплекс, побудований на принципах SOLID та патерні Decorator, є готовим до інтеграції у промислові системи. Встановлено, що найбільшу ефективність метод демонструє в e-commerce системах та медіа-ресурсах із нерівномірним трафіком. Раціональне використання апаратних ресурсів (CPU та RAM) знижує витрати на хмарну інфраструктуру, забезпечуючи при цьому стабільно високу якість обслуговування користувачів.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Rakesh R. The role of Caching in Modern System Design. *Medium*. 2023. URL: https://medium.com/@ramteke.rakesh/the-role-of-caching-in-modern-system-design-a30132d0c438 (дата звернення: 17.01.2026).
2. Zhbankov Y. Caching Essentials: Types, Strategies, and Best Practices. *Medium*. 2024. URL: https://medium.com/@yaroslavzhbankov/caching-essentials-types-strategies-and-best-practices-459493cc47d9 (дата звернення: 17.01.2026).
3. Rawat R. A Comprehensive Guide to Caching: Benefits, Challenges, and Best Practices. *Medium*. 2023. URL: https://medium.com/@DevBox_rohitrawat/a-comprehensive-guide-to-caching-benefits-challenges-and-best-practices-70bf88976858 (дата звернення: 17.01.2026).
4. Caching guidance. *Microsoft Learn*. URL: https://learn.microsoft.com/en-us/azure/architecture/best-practices/caching (дата звернення: 17.01.2026).
5. Caching – System Design Concept for Beginners. *GeeksforGeeks*. 2024. URL: https://www.geeksforgeeks.org/system-design/caching-system-design-concept-for-beginners/ (дата звернення: 18.01.2026).
6. Difference between Local Storage, Session Storage and Cookies. *GeeksforGeeks*. 2023. URL: https://www.geeksforgeeks.org/javascript/difference-between-local-storage-session-storage-and-cookies/ (дата звернення: 18.01.2026).
7. Load Balancers, API Gateways, Proxies, & CDNs Explained: The Network Components that Scale Your App. *Towards Dev*. 2023. URL: https://towardsdev.com/load-balancers-api-gateways-proxies-cdns-explained-the-network-components-that-scale-your-app-13c33949e8f1 (дата

звернення: 18.01.2026).

8. Sesmiat. Database Caching Strategies. *Medium*. 2024. URL: https://medium.com/@sesmiat/database-caching-strategies-f5e40c3c9b74 (дата звернення: 18.01.2026).
9. Lagziel B. In-Memory vs. Distributed Caching: A Comparative Look with Caffeine. *Medium*. 2024. URL: https://medium.com/@baraklagziel/in-memory-vs-distributed-caching-a-comparative-look-with-caffeine-15cedf6038c6 (дата звернення: 19.01.2026).
10. Santarosa A. Heap, Stack and Garbage Collector — A practical guide to .NET memory management system. *Medium*. 2022. URL: https://andresantarosa.medium.com/heap-stack-e-garbage-collector-a-practical-guide-to-net-memory-management-system-7e60bbadf199 (дата звернення: 23.01.2026).
11. Workstation and server garbage collection. *Microsoft Learn*. 2024. URL: https://learn.microsoft.com/pt-br/dotnet/standard/garbage-collection/workstation-server-gc (дата звернення: 23.01.2026).
12. The large object heap on Windows systems. *Microsoft Learn*. 2023. URL: https://learn.microsoft.com/pt-br/dotnet/standard/garbage-collection/large-object-heap (дата звернення: 23.01.2026).
13. Cache in-memory in ASP.NET Core. *Microsoft Learn*. 2024. URL: https://learn.microsoft.com/en-us/aspnet/core/performance/caching/memory?view=aspnetcore-10.0 (дата звернення: 28.01.2026).
14. Distributed caching in ASP.NET Core. *Microsoft Learn*. 2024. URL: https://learn.microsoft.com/en-us/aspnet/core/performance/caching/distributed?view=aspnetcore-10.0 (дата звернення: 28.01.2026).
15. Practical Caching in .NET: Strategies, Eviction, Keys, and Metrics. *DEV Community*. 2024. URL:

https://dev.to/vimaltwit/practical-caching-in-net-strategies-eviction-keys-and-metrics-1d53 (дата звернення: 28.01.2026).

16. Redis vs Memcached: Which In-Memory Data Store Should You Use? *DEV Community*. 2024. URL: https://dev.to/lovestaco/redis-vs-memcached-which-in-memory-data-store-should-you-use-1m38 (дата звернення: 29.01.2026).
17. Introducing Garnet – an open-source, next-generation, faster cache-store. *Microsoft Research Blog*. 2024. URL: https://www.microsoft.com/en-us/research/blog/introducing-garnet-an-open-source-next-generation-faster-cache-store-for-accelerating-applications-and-services/ (дата звернення: 29.01.2026).
18. FusionCache Documentation. Step by Step. *GitHub*. 2024. URL: https://github.com/ZiggyCreatures/FusionCache/blob/main/docs/StepByStep.md (дата звернення: 29.01.2026).
19. EasyCaching in a ASP.NET Core Minimal API project. *DEV Community*. 2023. URL: https://dev.to/kasuken/easycaching-in-a-aspnet-core-minimal-api-project-263f (дата звернення: 29.01.2026).
20. Boosting Performance in .NET Core: Exploring MemoryCache and LazyCache. *Medium*. 2024. URL: https://medium.com/@dev.mjdhanesh/boosting-performance-in-net-core-exploring-memorycache-and-lazycache-dfec046fc4c8 (дата звернення: 29.01.2026).
21. Kanchana S. Why Time-To-Live (TTL) Caching is Non-Negotiable for High-Performance Mobile Apps. *Medium*. 2024. URL: https://medium.com/@sachithrakanchana.ks/why-time-to-live-ttl-caching-is-non-negotiable-for-high-performance-mobile-apps-49399dfffd90 (дата звернення: 30.01.2026).
22. Thread-Safe Collections. *Microsoft Learn*. URL:

https://learn.microsoft.com/en-us/dotnet/standard/collections/thread-safe/ (дата звернення: 08.02.2026).
23. Interlocked Class. *Microsoft Learn*. URL: https://learn.microsoft.com/en-us/dotnet/api/system.threading.interlocked (дата звернення: 09.02.2026).
24. How to use the Redis Docker Official Image. *Docker Blog*. 2024. URL: https://www.docker.com/blog/how-to-use-the-redis-docker-official-image/ (дата звернення: 10.02.2026).
25. Serializing and Deserializing JSON in .NET. *Microsoft Learn*. URL: https://learn.microsoft.com/en-us/dotnet/standard/serialization/system-text-json/overview (дата звернення: 10.02.2026).
26. Caching Guidance. *Microsoft Learn*. URL: https://learn.microsoft.com/en-us/azure/architecture/best-practices/caching (дата звернення: 10.02.2026).
27. Gamma E., Helm R., Johnson R., Vlissides J. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1994. (дата звернення: 10.02.2026).
28. Redis persistence. *Redis Documentation*. URL: https://redis.io/docs/latest/operate/oss_and_stack/management/persistence/ (дата звернення: 10.02.2026).
29. Cache-Aside Pattern. *Microsoft Azure Architecture Center*. URL: https://learn.microsoft.com/en-us/azure/architecture/patterns/cache-aside (дата звернення: 10.02.2026).
30. Arlitt M. F., Williamson C. L. Internet Web servers: Workload characterization and performance implications. *IEEE/ACM Transactions on Networking*. 1996. Vol. 4, no. 5. P. 634–645 (дата звернення: 10.02.2026).
31. Kovalyov V. Generating fake data in .NET with Bogus. *Medium*. 2023. URL:

https://medium.com/@kova98/generating-fake-data-in-net-with-bogus-ca75e181fdf9 (дата звернення: 11.02.2026).

32. Hash Code. Simple and effective load testing with NBomber and C#. *Medium*. 2023. URL: https://medium.com/@hash_code/simple-and-effective-load-testing-with-nbomber-and-c-d0a6b1bcaecd (дата звернення: 11.02.2026).
33. Datar M., Gionis A., Indyk P., Motwani R. Maintaining stream statistics over sliding windows. *SIAM Journal on Computing*. 2002. Vol. 31, no. 6. P. 1794–1813 (дата звернення: 14.02.2026)
34. Jung J., Berger A. W., Balakrishnan H. Modeling TTL-based HTTP cache performance. *IEEE INFOCOM*. 2003. Vol. 1. P. 417–427 (дата звернення: 15.02.2026)
35. Brooker M. Exponential Backoff and Jitter. *AWS Architecture Blog*. 2015. URL:https://aws.amazon.com/ru/blogs/architecture/exponential-backoff-and-jitter/ (дата звернення: 15.02.2026)