Вінницький національний технічний університет
Факультет інформаційних технологій та комп'ютерної інженерії
Кафедра обчислювальної техніки

МАГІСТЕРСЬКА КВАЛІФІКАЦІЙНА РОБОТА

на тему:

СЕРВІС УПРАВЛІННЯ КЕШЕМ ДАНИХ ЗА ТЕХНОЛОГІЄЮ РОЗПОДІЛЕНОГО KEY-VALUE СХОВИЩА

Виконав: студент 2 курсу, групи 1КІ-24м
спеціальності 123 — «Комп'ютерна інженерія»
_____Димпалов Є. І.
Керівник: к.т.н., доц. каф. ОТ
_____Савицька Л. А.
«15» 12 2025 р.
Опонент: к.т.н., доц. каф. ПЗ
_____ Рейда О. М.
«15» 12 2025 р.

Допущено до захисту

Завідувач кафедри ОТ
_____ д.т.н., проф. Азаров О. Д.
«18» 12 2025 р.

ВНТУ 2025

ВІННИЦЬКИЙ НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ

Факультет інформаційних технологій та комп'ютерної інженерії
Кафедра обчислювальної техніки
Галузь знань — Інформаційні технології
Освітній рівень — магістр
Спеціальність — 123 Комп'ютерна інженерія
Освітньо-професійна програма — Комп'ютерна інженерія

ЗАТВЕРДЖУЮ

Завідувач кафедри обчислювальної техніки

_____ д.т.н., проф. О.Д. Азаров

" 25 " вересня 2025 р.

ЗАВДАННЯ

НА МАГІСТЕРСЬКУ КВАЛІФІКАЦІЙНУ РОБОТУ

студенту Димпалову Є. І.

1 Тема роботи «Сервіс управління кешем даних за технологією розподіленого key-value сховища», керівник роботи Савицька Л. А. к.т.н., доцент, затверджено наказом вищого навчального закладу від 24.09.25 року №313.

2 Строк подання студентом роботи 17.12.2025.

3 Вихідні дані до роботи: призначення — гібридна платформа для швидкодіючого кешування та керування логічною узгодженністю даних у розподілених мікросервісних системах; базова архітектура сховища — in-memory key-value; протокол доступу — gRPC та REST; забезпечення надійності – persistence module та механізм eventually consistent реплікації.

4 Зміст текстової частини : вступ, теоретичні основи побудови системи управління кешем даних за технологією розподіленого key–value сховища, проектування системи управління кешем на основі key-value сховища, програмна реалізація та експерементальне дослідження Muninn.

5 Перелік графічного матеріалу: схема архітектурного потоку даних при бінарному зберіганні, схема Use Case Diagram, структурна схема архітектури кластерного Key-Value сховища

6 Консультанти розділів роботи приведений в таблиці 1.

Таблиця 1 — Консультанти розділів роботи

Розділ	Прізвище, ініціали та посада консультанта	Підпис, дата	
		завдання видав	Завдання прийняв
1-4	Савицька Л. А. к.т.н., доцент кафедри ОТ	25.09.2025	1-4
5	Ратушняк Ольга Георгіївна к.т.н., доц., доцент кафедри ЕПВМ	25.09.2025	5
Нормоконтроль	асист. каф. ОТ Швець С. І.		

7 Дата видачі завдання – 25.09.2025

8 Календарний план виконання приведений в таблиці 2.

Таблиця 2 – Календарний план

№ з/п	Назва етапів дипломної роботи	Термін виконання		Примітки
		початок	закінчення	
1	Розробка та затвердження завдання на виконання МКР	25.09.25	30.09.25	вик.
2	Огляд літератури за темою роботи	01.10.25	10.10.25	вик.
3	Складання календарного плану МКР	13.10.25	17.10.25	вик.
4	Аналіз предметної області	20.10.25	31.10.25	вик.
5	Розробка проєктних рішень	03.11.25	07.11.25	вик.
6	Моделювання та конструювання	10.11.25	21.11.25	вик.
7	Кодування пристрою	24.11.25	27.11.25	вик.
8	Розробка економічної частини	01.12.25	03.12.25	вик.
9	Оформлення МКР та презентації	09.12.25	12.12.25	вик.
10	Перевірка якості виконання МКР та усунення недоліків	15.12.25	16.12.25	вик.

Студент _____ Димпалов Є. І.

Керівник роботи _____ Савицька Л. А.

АНОТАЦІЯ

УДК 004

Димпалов Є. І. Сервіс управління кешем даних за технологією розподіленого key-value сховища. Магістерська кваліфікаційна робота зі спеціальності 123 — Комп’ютерна інженерія, Вінниця: ВНТУ, 2025 — 106 с. На укр. мові. Бібліогр.: 30 назв; рис.: 4; табл.: 12.

У магістерській кваліфікаційній роботі розглянуто проблеми забезпечення високої продуктивності, масштабованості та узгодженості даних у сучасних мікросервісних архітектурах, що використовують розподілені key-value сховища для кешування. Проаналізовано існуючі рішення класу in-memory cache, зокрема Redis та Memcached, визначено їхні переваги та обмеження в контексті використання у .NET-орієнтованих системах. Обґрунтовано вибір архітектурних та технологічних рішень для побудови власного сервісу управління кешем Muninn, який поєднує швидкий доступ до даних в оперативній пам’яті з опціональними механізмами персистентності. Розроблено багаторівневу архітектуру сервісу, що включає ядро керування кешем та API-рівень з підтримкою протоколів gRPC і HTTP REST.

Ключові слова: кешування даних, in-memory, key-value сховище, Muninn, gRPC, персистентність, мікросервісна архітектура.

ABSTRACT

Dympalov Ye. I. Data Cache Management Service Based on Distributed Key-Value Storage Technology. Master's Qualification Thesis in Specialty 123 — Computer Engineering, Vinnytsia: VNTU, 2025 — 106 pp. In Ukrainian. References: 30 sources; figures: 4; tables: 12.

The master's qualification thesis addresses the challenges of ensuring high performance, scalability, and data consistency in modern microservice architectures that use distributed key-value stores for caching. Existing in-memory cache solutions, in particular Redis and Memcached, are analyzed, and their advantages and limitations are identified in the context of use within .NET-oriented systems. The choice of architectural and technological solutions for building a custom cache management service, Muninn, is substantiated. This service combines fast access to data stored in main memory with optional persistence mechanisms. A multi-layer architecture of the service has been designed, including a cache management core and an API layer with support for gRPC and HTTP REST protocols.

Keywords: data caching, in-memory, key-value store, Muninn, gRPC, persistence, microservice architecture.

ЗМІСТ

ВСТУП

Сучасні інформаційні системи та веб-додатки характеризуються високою динамічністю, великими обсягами даних та необхідністю забезпечення швидкого доступу до них. Одним із ключових викликів у розробці таких систем є оптимізація часу відповіді на запити користувачів і підвищення продуктивності обчислювальних ресурсів. Традиційні підходи до зберігання та обробки даних часто не здатні забезпечити необхідну швидкість обробки інформації через високу затримку доступу до основних баз даних.

Вирішенням цієї проблеми є використання кешування — тимчасового зберігання даних у високошвидкісній пам’яті для прискорення повторного доступу до них. Особливо ефективним стає застосування розподілених кеш-систем, що базуються на технології key-value сховищ. Такий підхід дозволяє масштабувати систему горизонтально, рівномірно розподіляючи навантаження між вузлами та забезпечуючи надійність і відмовостійкість.

У рамках цієї роботи розглядається розробка сервісу управління кешем даних, який використовує технологію розподіленого key-value сховища. Основна увага приділяється питанням ефективного зберігання та вилучення даних, підтримки консистентності, управління часом життя об’єктів у кеші, а також масштабованості та продуктивності системи. Використання подібного сервісу дозволяє значно знизити навантаження на основну базу даних та забезпечити стабільну швидкість обробки запитів навіть за високих обсягів трафіку.

Результати дослідження можуть бути застосовані у розробці високопродуктивних веб-додатків, систем аналітики в реальному часі, інтернет-магазинів, соціальних мереж та інших сервісів, де швидкість доступу до даних є критичною.

Актуальність теми дослідження полягає в критичній важливості технології кешування в контексті швидкозростаючих обсягів даних та зростаючих вимог до інформаційних систем. Сучасні веб-сервіси, розподілені обчислювальні системи та платформи обробки корпоративних даних

вимагають ефективного управління проміжними обчислювальними результатами та збереженими даними. Неефективне використання кешу може призвести до зниження продуктивності, перевантаження бази даних та зменшення часу відгуку системи.

Розподілені key-value сховища (такі як Redis, Memcached та Hazelcast) стали фундаментальними для створення ефективних, масштабованих та високодоступних систем управління кешем. Однак організація ефективних служб управління кешем у розподіленому середовищі вимагає вирішення численних проблем — від вибору стратегії кешування до забезпечення узгодженості даних між вузлами системи.

Метою роботи є створення сервісу перевірки та управління даними на основі модульної технології зберігання ключ-значення для підвищення інформаційної ємності та надійності інформаційних систем.

Для досягення цієї мети необхідно вирішити такі **задачі:**

- проаналізувати сучасні підходи до кешування даних у розподілених системах;
- дослідити принципи побудови систем управління кешем у масштабованих середовищах;
- розробити архітектуру сервісу управління кешем, що підтримує розподілене зберігання та синхронізацію даних.

Об'єктом дослідження є процес управління кешем даних у розподілених інформаційних системах.

Предметом дослідження є методи та алгоритми організації, зберігання та обробки даних у розподілених key-value сховищах для підвищення продуктивності кешу.

Новизна роботи полягає у вдосконаленні сервісу управління кешем даних, що базується на технології розподіленого key-value сховища, із впровадженням оптимізованих алгоритмів керування життєвим циклом об’єктів кешу та механізмів масштабування системи, що дозволяє підвищити ефективність обробки запитів у порівнянні з існуючими рішеннями.

Практичне значення роботи полягає в потенціалі використання розроблених сервісів для підвищення ефективності веб-застосунків, мікросервісних архітектур та корпоративних платформ, які обробляють великі обсяги даних у режимі реального часу.

Апробація результатів роботи здійснена у доповідях на LI Науково-технічній конференції підрозділів Вінницького національного технічного університету (2025).

1 ТЕОРЕТИЧНІ ОСНОВИ ПОБУДОВИ СИСТЕМИ УПРАВЛІННЯ КЕШЕМ ДАНИХ ЗА ТЕХНОЛОГІЄЮ РОЗПОДІЛЕНОГО KEY-VALUE СХОВИЩА

1.1 Поняття та принципи роботи key–value сховищ

Сховища типу key-value відносяться до категорії нереляційних баз даних, які часто позначають як NoSQL. Основна відмінність таких систем від реляційних полягає в моделі зберігання даних: замість таблиць, рядків і стовпців інформація зберігається у вигляді пар «ключ-значення» [3].

У key-value сховищах ключ виступає унікальним ідентифікатором, що дозволяє однозначно визначити конкретне значення. Це забезпечує високу ефективність при пошуку та отриманні даних, оскільки звернення відбувається без необхідності складних запитів або об’єднання таблиць [3].

Значення, яке асоціюється з ключем, може мати будь-яку структуру. Це можуть бути прості типи даних, як числа або рядки, структуровані об’єкти, серіалізовані JSON-дані або навіть двійкові файли, наприклад зображення чи аудіо. Така гнучкість робить key-value сховища універсальними для різних застосувань.

Основною операцією, яка визначає функціонування цих систем, є швидкий пошук значення за ключем. Алгоритмічно це часто реалізується за допомогою хеш-таблиць, що дозволяє досягти середнього часу доступу O(1), тобто практично миттєвого отримання даних [4].

Простота доступу до інформації робить key-value сховища оптимальним вибором для кешування, де швидкість відповіді критична. Вони широко застосовуються у високопродуктивних веб-сервісах, системах обробки аналітики в реальному часі, інтернет-магазинах та соціальних мережах.

До основних операцій таких сховищ належить Add, яка відповідає за додавання нової пари ключ-значення у систему. Метод зазвичай приймає ключ, значення та додаткові параметри, наприклад час життя об’єкта в кеші (lifetime).

Операція Get дозволяє отримати значення за вказаним ключем. Якщо ключ існує в системі, метод повертає відповідне значення; у протилежному випадку може бути повернута помилка або спеціальне повідомлення про відсутність даних [4].

Операція Delete або Remove відповідає за видалення пари ключ-значення. Деякі реалізації можуть повертати видалене значення, що дозволяє здійснювати контроль над змінами в системі та відновлювати дані за потреби.

Операція Update забезпечує оновлення існуючих даних у кеші. Метод приймає ключ та нове значення, а також додаткові параметри, такі як lifetime. Якщо ключ відсутній у системі, операція може викликати помилку, що гарантує цілісність даних [5].

Операція Insert забезпечує збереження даних незалежно від наявності ключа в системі. Це корисно для сценаріїв, де необхідно гарантувати, що дані будуть збережені у кеші без ризику дублювання або втрати інформації.

Такі операції формують основу для створення високоефективних механізмів кешування. Кешування дозволяє тимчасово зберігати часто використовувані дані у швидкій пам’яті, зменшуючи затримки доступу та навантаження на основну базу даних.

Key-value сховища особливо ефективні у розподілених системах, де обробка великої кількості запитів потребує горизонтального масштабування. Дані можуть бути рівномірно розподілені між кількома вузлами, що підвищує продуктивність і забезпечує відмовостійкість [5].

Одним із головних прикладів таких систем є Redis, яке підтримує як структури даних у вигляді рядків, списків, множин та хеш-таблиць, так і механізми відновлення даних після збою. Redis часто використовується як кеш та брокер повідомлень у веб-додатках.

Іншим популярним прикладом є Memcached, яке відоме високою швидкістю зберігання та отримання простих пар ключ-значення. Memcached часто застосовується для тимчасового зберігання результатів запитів до бази даних і зменшення навантаження на сервер [5].

Системи типу Hazelcast поєднують можливості розподіленого кешування та обчислень, дозволяючи виконувати паралельні операції над даними без потреби їх постійного переміщення між вузлами. Сховище Aerospike відзначається високою масштабованістю та низькою затримкою при обробці мільйонів запитів на секунду, що робить його придатним для великих розподілених систем реального часу.

Розподілені key-value системи також підтримують різні політики управління життєвим циклом даних. Наприклад, TTL (time-to-live) дозволяє автоматично видаляти записи після певного часу, що запобігає переповненню кешу [6].

Важливою характеристикою є відмовостійкість. Сучасні системи підтримують реплікацію даних між вузлами, що забезпечує збереження інформації навіть у разі виходу частини вузлів з ладу.

Масштабованість таких систем дозволяє додавати нові вузли без простою, забезпечуючи стабільну роботу при зростанні навантаження. Балансування навантаження відбувається автоматично за допомогою механізмів хешування або системи розподілу ключів [6]

Key-value сховища відрізняються простотою архітектури, що спрощує інтеграцію у різноманітні застосунки та сервіси. Вони не обтяжені складними зв’язками між таблицями, що прискорює розробку та обслуговування систем.

Такі сховища особливо актуальні для побудови систем аналітики в реальному часі, де необхідно обробляти великі потоки даних та швидко реагувати на зміни. Використання key-value систем дозволяє також знизити затрати на ресурси основної бази даних, оскільки повторювані запити обробляються з кешу, а не з диску.

Системи можуть підтримувати як синхронний, так і асинхронний доступ до даних, що дозволяє оптимізувати продуктивність у різних сценаріях використання. Ключовим фактором успішного застосування є правильне визначення стратегій кешування: які дані зберігати, який час життя встановлювати, які механізми очищення використовувати [6].

Завдяки простоті та ефективності key-value сховища стають основою для побудови сучасних високонавантажених додатків, де швидкість відповіді та масштабованість є критично важливими. Вони підтримують як оперативне кешування, так і довгострокове тимчасове зберігання даних, що робить їх універсальним інструментом для розробників.

Розробка власного сервісу управління кешем на основі розподіленого key-value сховища дозволяє оптимізувати роботу конкретного додатку та адаптувати механізми під специфіку бізнес-процесів. Висока продуктивність, надійність, масштабованість та гнучкість таких систем робить їх невід'ємною частиною сучасної інфраструктури веб-додатків, систем аналітики та IoT-проектів [6].

1.2 Архітектура in-memory key-value сховищ

Системи типу in-memory відрізняються від класичних баз даних тим, що всі активні дані зберігаються у оперативній пам'яті (RAM), а не на жорсткому диску або SSD. Такий підхід дозволяє досягти надзвичайно високої швидкодії, оскільки час доступу до даних вимірюється мікро- або наносекундами, що значно перевищує швидкість дискових операцій. Це робить in-memory системи ідеальними для сценаріїв, де критичною є швидкість обробки великих обсягів запитів, наприклад, у високонавантажених веб-додатках, системах фінансової аналітики чи онлайн-іграх [7].

Попри високі швидкісні показники, чисто in-memory підхід має суттєвий недолік — ризик втрати даних у разі відключення живлення або аварійного збою. Щоб забезпечити надійність, сучасні системи зазвичай комбінують in-memory сховище з механізмами персистентності, які дозволяють зберігати стан даних у файловій системі та відновлювати їх після перезапуску. Такий комбінований підхід поєднує високу продуктивність та надійність одночасно.

У проекті Muninn реалізовано саме комбінований підхід. Активні ключі та їх значення зберігаються безпосередньо в оперативній пам'яті, що забезпечує миттєвий доступ до найчастіше використовуваних даних. При цьому, для збереження цілісності інформації, ці дані можуть бути серіалізовані

у бінарному вигляді та періодично записані у сховище на диску. Така стратегія дозволяє відновлювати стан системи після перезапуску або збою без втрати даних [7].

Архітектурно Muninn складається з кількох основних компонентів, кожен з яких виконує певні функції та забезпечує гнучкість системи. Першим компонентом є In-memory engine, який відповідає за всі базові операції читання та запису у пам’ять. Він забезпечує швидку роботу з даними та є ядром продуктивності системи [7].

Другим ключовим компонентом є Persistence module, який займається записом даних на диск у вигляді бінарних файлів. Модуль персистентності дозволяє створювати резервні копії та гарантує відновлення даних у разі перезапуску системи. Це забезпечує баланс між високою швидкістю доступу до пам’яті та надійністю збереження інформації [7].

Третій компонент, Encoding layer, визначає правила перетворення даних у байтовий потік залежно від обраного клієнтом кодування. Цей модуль дозволяє уніфікувати дані для зберігання у пам’яті та на диску, а також оптимізувати їх обсяг, що підвищує ефективність системи.

Четвертим компонентом є Network API, який забезпечує взаємодію системи з клієнтами. Через цей шар реалізується доступ до сервісу за протоколами gRPC або HTTP, що дозволяє інтегрувати Muninn у різні додатки та сервіси без значних змін у клієнтському коді [7].

П’ятий компонент, Cluster manager, є опціональним і відповідає за масштабування системи у розподіленому середовищі. Завдяки цьому компоненту можливо додавати нові вузли без зупинки сервісу, що забезпечує відмовостійкість та підтримку високого навантаження.

У загальній архітектурі key-value сховища API-шар приймає запити від клієнтів, обробляє їх та передає на рівень зберігання. Це дозволяє централізовано керувати доступом, перевіряти валідність запитів і контролювати безпеку [7].

Модуль кодування (Encoder) відповідає за перетворення об’єктів у

бінарну форму перед записом у пам’ять або на диск. Це дозволяє зменшити обсяг даних, оптимізувати використання оперативної пам’яті та забезпечити універсальний формат для різних типів клієнтів.

Зворотне перетворення здійснюється модулем Decoder, який відновлює оригінальні об’єкти з бінарного представлення для використання клієнтами. Такий процес забезпечує прозорість для користувачів і дозволяє системі працювати з будь-якими структурами даних [7].

Модуль персистентності записує копію пам’яті у файл періодично або при зміні даних. Це дозволяє мінімізувати втрати інформації та гарантує відновлення стану системи у разі аварійного завершення роботи.

Компоненти Muninn інтегруються між собою, формуючи цілісну архітектуру. In-memory engine забезпечує швидкість, Persistence module гарантує надійність, Encoding layer уніфікує дані, Network API дозволяє взаємодію з клієнтами, а Cluster manager забезпечує масштабованість та відмовостійкість.

Розподілення даних між оперативною пам’яттю та диском дозволяє ефективно управляти ресурсами системи. Часто використовувані ключі залишаються в пам’яті, тоді як менш активні можуть бути записані на диск для економії оперативної пам’яті. Таке рішення оптимізує час доступу до критично важливих даних, зменшує затримки запитів та підвищує загальну продуктивність системи [8].

Мережевий API забезпечує стандартизований інтерфейс доступу, що дозволяє різним клієнтам працювати з Muninn незалежно від їхньої технологічної платформи. Це значно спрощує інтеграцію системи у складні інфраструктури (рисунок 1.1).

Cluster manager дозволяє масштабувати систему горизонтально, додаючи нові вузли до кластера без простою. Це підвищує відмовостійкість та забезпечує стабільну роботу при зростанні навантаження.

Encoding layer забезпечує сумісність даних між різними компонентами та клієнтами. Його використання дозволяє оптимізувати зберігання та

передачу даних, а також підвищує ефективність роботи кешу.

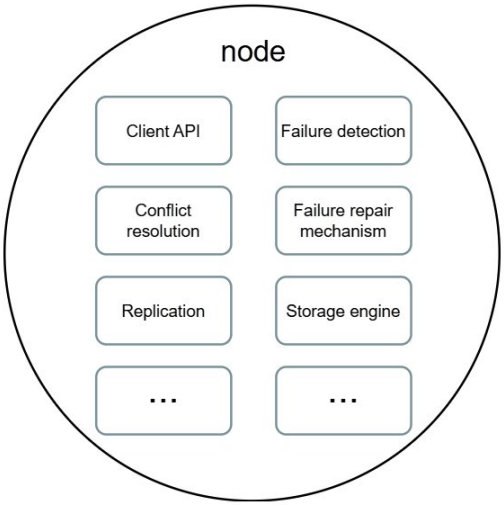

Рисунок 1.1 – Архітектура функціональних компонентів розподіленого Key-Value сховища

Модуль персистентності може реалізовувати різні стратегії збереження даних, включно з періодичним скиданням, записом після зміни або зберіганням журналів змін. Це забезпечує гнучкість у налаштуванні системи під конкретні задачі [8].

Архітектура Muninn дозволяє реалізовувати відмовостійкі та високопродуктивні системи, де оперативна пам’ять використовується для швидкого доступу, а дискове сховище — для довгострокового збереження.

Комбінація in-memory та персистентності дозволяє досягти балансу між швидкістю та надійністю, що є критично важливим для сучасних високонавантажених застосунків [8].

Система підтримує різні формати даних і дозволяє клієнтам обирати кодування, що робить її універсальною та гнучкою для різноманітних сценаріїв використання. Завдяки продуманій архітектурі, Muninn здатна обробляти велику кількість запитів на секунду, забезпечуючи стабільну роботу навіть у пікові періоди навантаження [8].

Важливим аспектом є можливість резервного копіювання та відновлення даних, що гарантує безперервність роботи і збереження інформації при аварійних ситуаціях.

Загалом, архітектура in-memory систем типу Muninn забезпечує ефективне управління даними, високу швидкодію, надійність та масштабованість, що робить їх невід'ємною частиною сучасних високопродуктивних інформаційних систем.

На рисунку 1.2 представлено загальну структуру системи типу key-value сховища.

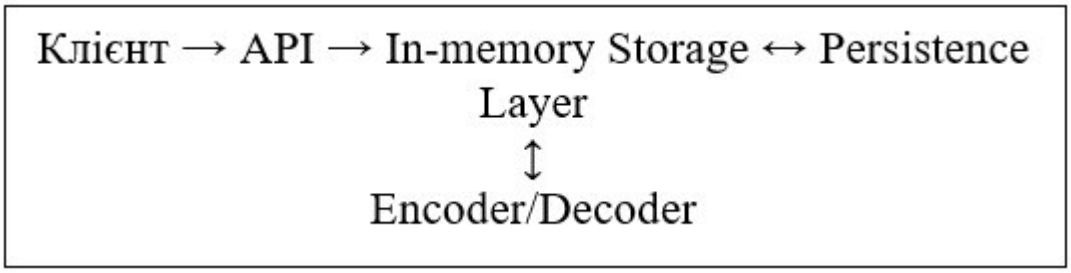

Рисунок 1.2 — Узальнена архітектура стистеми типу key-value сховища.

У цій архітектурі API-шар приймає запити від клієнта, обробляє їх і передає на рівень зберігання. Модуль кодування (Encoder) відповідає за перетворення об'єктів у бінарну форму, тоді як Decoder — за зворотне перетворення. Модуль персистентності періодично або при зміні даних записує копію пам'яті у файл.

1.3 Принципи бінарного зберігання даних та динамічного кодування

Традиційні текстові формати серіалізації, такі як JSON або XML, здобули популярність завдяки своїй зрозумілості та зручності для людини. Вони дозволяють легко читати та інтерпретувати структуру даних, що особливо корисно під час налагодження та тестування систем. Проте, при роботі з високопродуктивними сервісами, де критичною є швидкість обробки інформації та мінімізація затримок, використання таких форматів може стати вузьким місцем. Основною проблемою є надмірна структурність цих форматів та великий обсяг, який займають символи, що описують структуру даних, такі як дужки, теги та лапки. Це призводить до додаткового навантаження на канали передачі даних та збільшує час серіалізації і десеріалізації, що негативно впливає на продуктивність систем реального часу. У додатку В зображено схему

архітектурного потоку даних при бінарному зберіганні та відновленні у системі Muninn [9].

Для високошвидкісних систем, таких як Muninn, критичною вимогою є не лише швидкий доступ до даних у пам’яті, але й мінімізація обсягу переданих і оброблюваних даних. Саме тому Muninn використовує бінарну серіалізацію, яка забезпечує компактне представлення об’єктів у вигляді послідовності байтів. Бінарна модель дозволяє зменшити обсяг переданих даних, прискорити запис у пам’ять і зменшити затрати на мережеву передачу, що робить її ефективною для сценаріїв з високим навантаженням [9].

Особливістю Muninn є реалізація власного механізму динамічного кодування, який дає клієнту можливість обирати формат серіалізації (Encoding) під час запису об’єктів у кеш. Такий підхід забезпечує гнучкість системи та дозволяє працювати незалежно від конкретного формату даних, що особливо корисно у багатомовних або гетерогенних середовищах. Клієнт може обрати кодування, яке відповідає його потребам, наприклад UTF-8 для універсальної сумісності або UTF-16 для роботи з великими набором символів [9]

Завдяки динамічному кодуванню система отримує низку переваг. По-перше, це дозволяє ефективно працювати у багатомовних системах, забезпечуючи коректну обробку текстових даних різних алфавітів без втрат інформації. По-друге, використання бінарної серіалізації мінімізує затрати при передачі даних між сервісами, що особливо важливо для розподілених систем із великою кількістю вузлів або клієнтів. По-третє, така модель дозволяє розробникам контролювати спосіб зберігання даних без необхідності змінювати бізнес-логіку додатку, що забезпечує більшу стабільність та прозорість роботи системи.

Бінарна модель зберігання передбачає, що кожен об’єкт у системі Muninn має унікальний ідентифікатор (key), який використовується як індекс у пам’яті для швидкого доступу. Сам об’єкт перетворюється на компактну послідовність байтів, що зменшує обсяг пам’яті та прискорює операції запису і читання. Крім того, інформація про обране кодування також зберігається у кеші разом із

даними, що дозволяє клієнту виконати десеріалізацію без додаткових перетворень або помилок [9].

Такий підхід забезпечує максимальну ефективність роботи системи: ключ виступає як швидкий індекс доступу, байтовий потік дозволяє економно використовувати ресурси пам’яті та канали передачі, а збереження інформації про кодування гарантує сумісність та коректне відновлення даних незалежно від клієнта. Завдяки цьому Muninn здатна ефективно обробляти великі обсяги інформації, забезпечуючи високу продуктивність навіть у сценаріях з численними одночасними запитами.

Бінарна серіалізація також дозволяє системі реалізовувати додаткові оптимізації, наприклад стиснення даних або використання різних алгоритмів кодування залежно від типу об’єкта, що записується. Це дає змогу ще більше зменшити обсяг пам’яті та швидко передавати інформацію між вузлами кластера або сервісами (рисунок 1.3).

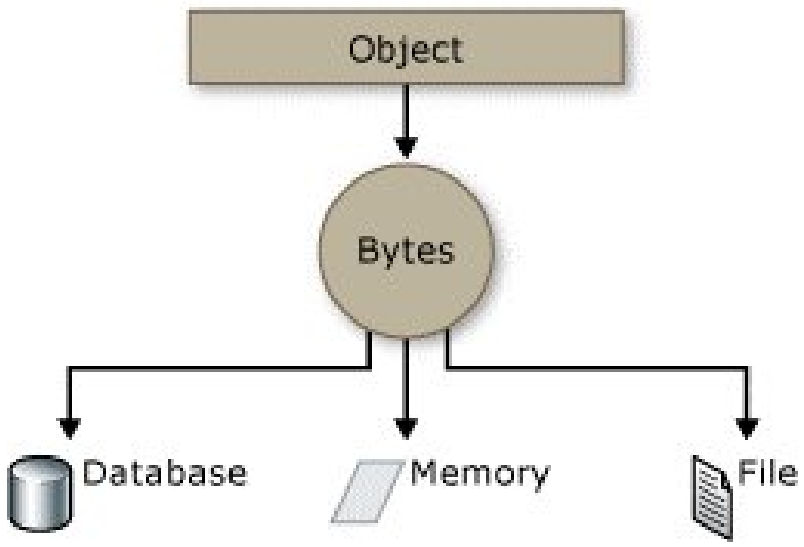

Рисунок 1.3 - Бінарна серіалізація

Завдяки комбінації динамічного кодування та бінарного зберігання, Muninn забезпечує баланс між гнучкістю, швидкістю доступу та мінімізацією ресурсних витрат. Така архітектура робить систему придатною для високонавантажених веб-додатків, систем аналітики в реальному часі та розподілених сервісів, де кожна мілісекунда відповіді має значення [10].

Механізм бінарної серіалізації дозволяє також забезпечити ефективну роботу з кешем, оскільки об’єкти можна швидко додавати, отримувати або видаляти без необхідності аналізу складної текстової структури. Це прискорює виконання запитів і зменшує затримки у користувацьких сценаріях.

В цілому, використання динамічного кодування та бінарної серіалізації у Muninn дозволяє створити високопродуктивний сервіс управління кешем, який одночасно є гнучким, масштабованим і надійним [10].

1.4 Механізми персистентності та відновлення

Система Muninn підтримує асинхронне збереження даних, що означає, що знімки стану кешу створюються у фоновому режимі без блокування основних операцій. Такий підхід дозволяє одночасно забезпечити високу продуктивність і надійність системи: записи в оперативну пам’ять можуть виконуватися без затримок, навіть якщо паралельно відбувається запис стану на диск. Це особливо важливо для високонавантажених сценаріїв, де навіть невелика затримка у відповіді на запит може негативно вплинути на користувацький досвід або роботу бізнес-логіки [11].

Для управління персистентністю Muninn реалізує декілька стратегій, які можна обирати залежно від потреб конкретного застосунку. Першою є Periodical Snapshot, що передбачає регулярне створення повної копії сховища через задані інтервали часу. Цей метод дозволяє швидко відновити стан системи після аварії, хоча у разі збою між двома snapshot можуть бути втрачені деякі останні операції.

Другою стратегією є Write-Ahead Log (WAL). У цьому підході всі операції записуються у лог перед їх виконанням у основному сховищі. Це дозволяє відновити точний стан системи після збою, оскільки всі дії зберігаються у хронологічному порядку. WAL забезпечує високу точність відновлення та мінімізує ризик втрати даних, особливо у сценаріях з високою частотою змін [11].

Третій підхід, Hybrid Mode, поєднує переваги двох попередніх методів. Він передбачає регулярне створення snapshot і одночасне ведення логів усіх

операцій. Така комбінація дозволяє скоротити ризик втрати даних і прискорити відновлення системи після збою, одночасно підтримуючи високу продуктивність при активній роботі з оперативною пам’яттю [11].

Алгоритм відновлення даних у Muninn після аварійного завершення роботи системи або вузла кластеру складається з кількох етапів. Спочатку зчитується останній збережений snapshot, що дозволяє відновити стан системи на момент останньої фіксації. Далі відтворюються операції з Write-Ahead Log, що забезпечує відновлення всіх змін, які відбулися після останнього snapshot. На завершальному етапі оновлюється in-memory сховище, яке стає консистентним і готовим до обслуговування нових запитів [11].

Такий підхід забезпечує високий рівень консистентності даних навіть у разі неочікуваного завершення роботи контейнера або вузла в кластері. Система здатна швидко повернутися у працездатний стан без втрати критично важливої інформації, що є ключовою вимогою для високонавантажених розподілених сервісів.

Асинхронне збереження та гнучкі стратегії персистентності дозволяють Muninn досягати оптимального балансу між продуктивністю та надійністю. Користувачі системи отримують швидкий доступ до оперативних даних, при цьому не жертвуючи можливістю відновлення стану після збою [11].

Використання snapshot і WAL у комбінації дозволяє системі підтримувати високу доступність та ефективно працювати у кластерах з великою кількістю вузлів. Навіть у випадку виходу одного або декількох вузлів з ладу, дані можуть бути відновлені з мінімальними втратами і без порушення роботи інших компонентів [11].

Такий підхід до персистентності і відновлення даних робить Muninn придатним для застосування у критичних сценаріях, таких як фінансові сервіси, онлайн-торгівля, системи моніторингу та аналітики в реальному часі. Система може працювати з великими потоками операцій, зберігаючи консистентність та надійність навіть за високого навантаження [11].

Сукупність асинхронного збереження, snapshot, WAL та гібридного режиму забезпечує Muninn гнучкі можливості для адаптації під різні вимоги користувачів. Розробники можуть обирати оптимальну стратегію залежно від критичності даних, очікуваного навантаження та вимог до швидкості відновлення після збою.

Система підтримує як локальне, так і розподілене зберігання даних, що дозволяє масштабувати її горизонтально. Це особливо важливо для сучасних високонавантажених розподілених сервісів, де відмовостійкість і швидке відновлення стану є ключовими показниками якості системи.

Механізм асинхронного збереження дозволяє Muninn виконувати операції запису у пам’ять без затримки, у той час як модулі персистентності обробляють запис на диск у фоновому режимі. Це зменшує затримки при обслуговуванні запитів клієнтів і підвищує загальну продуктивність системи.

Застосування snapshot і WAL дозволяє створювати багаторівневий захист даних: snapshot забезпечує швидке відновлення стану на момент останньої фіксації, а WAL дозволяє відновити всі операції, що відбулися після цього моменту. Така комбінація гарантує мінімальні втрати інформації.

Алгоритм відновлення після збою є достатньо гнучким і може бути адаптований для різних сценаріїв: від відновлення одного вузла в кластері до відновлення стану всієї розподіленої системи. Це забезпечує універсальність і надійність роботи Muninn [12].

Використання асинхронного збереження дозволяє системі ефективно працювати з великими обсягами даних, не блокуючи оперативну пам’ять під час запису на диск. Завдяки цьому зберігається висока швидкість обробки запитів навіть у пікові моменти навантаження.

Стратегії персистентності також дозволяють контролювати політики зберігання даних: частота створення snapshot, ведення WAL, умови запуску гібридного режиму. Це дозволяє адаптувати систему під конкретні бізнес-вимоги і сценарії використання.

Muninn здатна підтримувати безперервну роботу в кластерах із великою кількістю вузлів, забезпечуючи синхронізацію стану між ними і зберігаючи консистентність даних навіть у разі виходу частини вузлів з ладу.

Асинхронне збереження та відновлення даних дозволяє підтримувати неперервність бізнес-процесів, оскільки користувачі або клієнтські додатки не помічають затримок у роботі системи під час створення резервних копій [12].

Завдяки механізмам персистентності Muninn можна застосовувати у критично важливих сценаріях, де необхідна висока надійність даних, наприклад у фінансових сервісах, онлайн-торгівлі, системах обліку та моніторингу.

Використання snapshot і WAL у гібридному режимі дозволяє оптимізувати баланс між швидкістю роботи і гарантією відновлення. Система не втрачає продуктивності під час запису даних, одночасно забезпечуючи високий рівень надійності.

Архітектура персистентності Muninn забезпечує легкість масштабування, оскільки додавання нових вузлів до кластера не впливає на поточні операції запису та читання. Додаткові вузли отримують оновлені дані за допомогою алгоритмів синхронізації, зберігаючи консистентність системи.

Гнучка конфігурація стратегій зберігання дозволяє адаптувати Muninn під різні сценарії: від систем із критичною потребою у швидкому відновленні після збою до систем із великою кількістю одночасних запитів і високою частотою запису.

Сукупність асинхронного збереження, snapshot, WAL та гібридного режиму дозволяє Muninn ефективно працювати в сучасних розподілених середовищах, забезпечуючи високу продуктивність, надійність та відмовостійкість системи. Підхід до персистентності, реалізований у Muninn, дозволяє забезпечити консистентність даних і мінімізувати ризик їх втрати навіть у критичних сценаріях, що робить систему надійним інструментом для побудови високопродуктивних сервісів.

2 ПРОЕКТУВАННЯ СИСТЕМИ УПРАВЛІННЯ КЕШЕМ НА ОСНОВІ ТЕХНОЛОГІЇ KEY–VALUE СХОВИЩА

2.1 Загальна архітектура та структура сервісу Muninn

Система керування кешем Muninn побудована на основі багаторівневої архітектури, орієнтованої на використання у розподілених сервісах із високими вимогами до доступності, продуктивності та масштабованості. Основний архітектурний принцип системи полягає у чіткому розмежуванні відповідальностей між окремими рівнями, що дозволяє зменшити зв’язаність компонентів, спростити модульне тестування та забезпечити можливість незалежного розвитку кожної частини системи без впливу на інші рівні.

Архітектура Muninn логічно поділяється на три ключові рівні, кожен з яких виконує чітко визначену роль у загальному процесі обробки запитів. До таких рівнів належать клієнтський рівень, API-рівень та ядро системи (рисунок 2.1). Такий поділ дозволяє ізолювати бізнес-логіку керування кешем від мережевої взаємодії та клієнтських реалізацій, що є важливим для підтримки високої гнучкості системи.

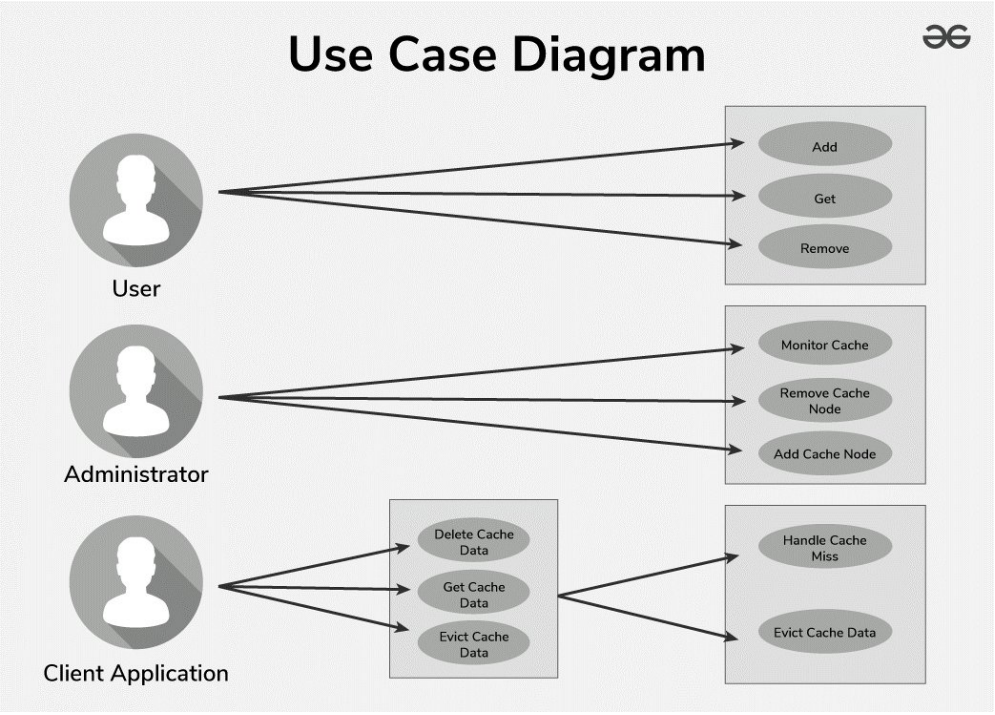

Рисунок 2.1 — Багаторівнева та модульна архітектура системи керування кешем Muninn

Клієнтський рівень (Client Layer) відповідає за формування запитів та взаємодію із зовнішніми системами. Саме на цьому рівні відбувається підготовка даних, серіалізація значень, конфігурація параметрів з'єднання та обробка відповідей від сервера. Клієнтський рівень не містить бізнес-логіки кешу, а виконує роль зручного інтерфейсу доступу до сервісу Muninn з боку інших додатків або мікросервісів [13].

API-рівень (API Layer) є точкою входу до сервісу Muninn і відповідає за прийом та маршрутизацію зовнішніх запитів. Він забезпечує комунікацію між клієнтами та ядром системи, виконуючи перевірку коректності запитів, мапінг даних у внутрішні моделі та передачу їх на рівень бізнес-логіки. Наявність окремого API-рівня дозволяє підтримувати декілька протоколів взаємодії без дублювання логіки керування кешем.

Ядро системи (Kernel Layer) є центральним компонентом архітектури Muninn і містить усю основну бізнес-логіку керування кешем. Саме на цьому рівні реалізуються алгоритми зберігання, пошуку, оновлення, видалення та фільтрації даних, а також механізми персистентності та керування життєвим циклом записів [13].

Ключовим модулем ядра є Muninn.Kernel, який інкапсулює всю логіку маніпулювання кешем та даними. Він виступає внутрішнім API для інших рівнів системи та забезпечує централізоване управління всіма підсистемами кешу. Структура цього модуля включає декілька основних компонентів, кожен з яких відповідає за окрему частину функціональності.

Основним фасадом ядра є компонент CacheManager (ICacheManager). Він реалізує єдиний інтерфейс управління кешем і слугує точкою взаємодії для API-рівня. CacheManager не виконує конкретних операцій з даними самостійно, а делегує запити до відповідних підсистем залежно від типу операції та конфігурації кешу. До таких підсистем належать:

- ResidentCache;
- SortedResidentCache;
- PersistentCache.

Компонент ResidentCache (IResidentCache) реалізує зберігання даних безпосередньо в оперативній пам’яті. Він забезпечує швидкий доступ до значень за ключем із середнім часом доступу O(1) та відповідає за керування життєвим циклом записів. До його обов’язків належить обробка TTL, видалення застарілих даних та реалізація політик витіснення у разі перевищення лімітів пам’яті [13].

Компонент SortedResidentCache (ISortedResidentCache) також реалізує in-memory зберігання, проте відрізняється додатковою логікою сортування даних. Після будь-яких змін у кеші, таких як додавання, оновлення або видалення записів, дані впорядковуються, що дозволяє виконувати швидкий бінарний пошук. Такий підхід є доцільним у сценаріях, де часто виконуються операції пошуку за діапазонами або відсортованими значеннями.

Компонент PersistentCache (IPersistentCache) відповідає за надійне зберігання даних на диску. Він керує механізмами створення snapshot, ведення журналу Write-Ahead Log (WAL) та відновлення стану системи після збою. PersistentCache тісно інтегрується з in-memory компонентами, забезпечуючи узгодженість між оперативною пам’яттю та дисковим сховищем.

Окрему роль у ядрі системи відіграє FilterService (IFilterService), який реалізує складну логіку пошуку та фільтрації даних. Цей компонент дозволяє виконувати фільтрацію як за ключами (KeyFilter), так і за значеннями (ValueFilter), що значно розширює можливості кешу порівняно з класичними key-value сховищами [13].

API-рівень системи представлений двома реалізаціями: Muninn.Api та Muninn.Api.Grpc. Muninn.Api є RESTful сервісом, реалізованим на основі .NET Minimal API. Такий підхід дозволяє мінімізувати накладні витрати та швидко мапити HTTP-запити у внутрішні типи даних, передаючи їх методам ядра для подальшої обробки.

Ключовим компонентом REST API є клас Register, який містить усі точки входу сервісу та описує маршрути (routes). Він визначає, які HTTP-методи відповідають конкретним операціям кешу, та забезпечує єдину точку конфігурації API.

Другим варіантом API є Muninn.Api.Grpc, який реалізує взаємодію через протокол gRPC. На відміну від REST, gRPC використовує строгі контракти, описані у .proto файлах, і передає дані у бінарному форматі. Це дозволяє досягти максимальної швидкості обробки запитів і відповідей, що особливо важливо для високонавантажених систем.

Ключовим компонентом gRPC-сервісу є клас MuninnService, який наслідується від автозгенерованого класу MuninnServiceBase та реалізує його методи. Цей клас виступає посередником між gRPC-клієнтами та ядром системи, делегуючи виклики до CacheManager [14].

Клієнтський рівень представлений бібліотеками Muninn.Client та Muninn.Client.Grpc, які призначені для використання у будь-яких зовнішніх сервісах або додатках. Ці бібліотеки надають зручний програмний інтерфейс для взаємодії з кешем і приховують деталі мережевої комунікації.

Обидві клієнтські бібліотеки мають спільну архітектуру та набір допоміжних компонентів, серед яких можна виділити:

- BinarySerializer, статичний клас для бінарної серіалізації значень;
- IBinarySerializable, контракт для маркування класів, що підлягають серіалізації без використання рефлексії;
- MuninnIgnoreAttribute, атрибут для виключення окремих полів або властивостей із процесу серіалізації;
- MuninnConfiguration, контейнер конфігураційних параметрів клієнта та кешу.

Центральним елементом клієнтських бібліотек є класи MuninnClient та MuninnClientGrpc, які реалізують інтерфейс IMuninnClient. Саме вони відповідають за формування запитів, їх відправлення на сервер та обробку відповідей, забезпечуючи прозору і зручну взаємодію з кешем Muninn. Завдяки такій багаторівневій та модульній архітектурі система Muninn є гнучкою, масштабованою та зручною для інтеграції у сучасні розподілені середовища, що робить її придатною для використання у високонавантажених сервісах з підвищеними вимогами до продуктивності та надійності. У додатку Г зображено

діаграму варіантів використання (Use Case Diagram) системи керування кешем Muninn [14].

2.2 Вимоги до апаратного забезпечення

Оскільки сервіс Muninn є системою типу in-memory з опціональною можливістю увімкнення персистентної складової, вимоги до апаратного забезпечення безпосередньо залежать від обсягу даних, що зберігаються в кеші, а також від інтенсивності операцій читання і запису. На відміну від класичних дискових сховищ, основне навантаження в Muninn припадає на оперативну пам’ять, що визначає ключові характеристики апаратної платформи.

Першочерговою вимогою до апаратного забезпечення є достатній обсяг оперативної пам’яті. Уся активна частина кешу постійно перебуває в RAM, що дозволяє забезпечити мінімальні затримки доступу до даних. Обсяг оперативної пам’яті має перевищувати розмір кешованих даних з урахуванням службової інформації, метаданих, структур індексації та накладних витрат, пов’язаних із бінарною серіалізацією об’єктів [15].

Другим важливим параметром є продуктивність центрального процесора. Muninn активно використовує багатопоточну обробку запитів, що передбачає паралельне виконання операцій серіалізації, десеріалізації, фільтрації та управління життєвим циклом записів. Для стабільної роботи сервісу рекомендовано використання багатоядерних процесорів, які здатні ефективно обслуговувати одночасні запити від великої кількості клієнтів [15].

Окрему увагу слід приділити частоті процесора, оскільки швидкодія системи значною мірою залежить від швидкості виконання операцій над пам’яттю. Висока тактова частота позитивно впливає на продуктивність при виконанні коротких, але численних операцій доступу до кешу, що є типовим сценарієм для сервісів in-memory.

У разі використання персистентної частини системи, додаткові вимоги висуваються до підсистеми зберігання даних. Для ефективної роботи механізмів snapshot та Write-Ahead Log рекомендовано використовувати твердотільні

накопичувачі (SSD), які забезпечують високу швидкість послідовного та випадкового запису. Використання повільних дискових накопичувачів може негативно вплинути на швидкість відновлення стану системи після збою [23].

Важливим фактором є також пропускна здатність підсистеми введення-виведення. Під час асинхронного збереження даних у фоновому режимі Muninn формує значні обсяги операцій запису, що потребує стабільної та швидкої роботи дискової підсистеми. Недостатня пропускна здатність може призвести до накопичення черг операцій і зростання затримок. Пропускна здатність сервісу визначається як:

$$Th = Nreq / T \ , \qquad (2.1)$$

де Nreq — кількість оброблених запитів;

TTT — час виконання.

Мережеві характеристики апаратної платформи мають істотне значення у випадку використання Muninn у розподіленому середовищі або в архітектурі мікросервісів. Для забезпечення високої швидкості обміну даними між клієнтами та сервером кешу рекомендовано використання мережевих інтерфейсів із низькою латентністю та високою пропускною здатністю, особливо при застосуванні gRPC-протоколу [15].

Крім того, стабільність роботи системи залежить від якості апаратної платформи та наявності механізмів апаратного резервування. Використання серверних рішень з підтримкою ECC-пам’яті зменшує ризик пошкодження даних в оперативній пам’яті, що є критичним для in-memory систем, де помилки можуть призвести до втрати актуального стану кешу.

У випадку контейнеризованого розгортання Muninn у середовищах Docker або Kubernetes, апаратні вимоги визначаються не лише характеристиками окремого вузла, але й загальною конфігурацією кластера. Необхідно враховувати

ліміти ресурсів, що виділяються контейнерам, а також можливі накладні витрати, пов’язані з віртуалізацією та оркестрацією [15].

Таким чином, мінімальні та рекомендовані вимоги до апаратного забезпечення для сервісу Muninn формуються на основі обсягу кешованих даних, очікуваного навантаження, необхідного рівня надійності та обраного режиму роботи системи. Узагальнені параметри апаратної платформи наведені у таблиці нижче та можуть коригуватися залежно від конкретних умов експлуатації сервісу.

Таблиця 2.1 — Вимоги до апаратної частини

Компонент	Мінімальні вимоги (local environment)	Рекомендовані вимоги (production environment)
Оперативна пам'ять (RAM)	8 ГБ (для системи та невеликого кешу)	32-64 ГБ (критично залежить від обсягу кешу)
Процесор (CPU)	4-ядерний (2.5 ГГц+)	16+ ядер, високочастотний (4.0 ГГц+)
Накопичувач (SSD)	SSD 256 ГБ (для швидкої персистентності)	NVMe SSD (для надшвидкого запису WAL)
Мережева карта	1 Гбіт/с Ethernet	10 Гбіт/с (для обслуговування великої кількості gRPC-запитів)
ОС	Windows Server 2022 / Ubuntu LTS	Ubuntu Server LTS (оптимізація для .NET)

2.3 Обґрунтування вибору технологій для розробки Muninn

Для забезпечення високої продуктивності, масштабованості та надійності, що є критичним для кеш-сервісу in-memory, було обрано

наступний технологічний стек (таблиця 2.2).

Для забезпечення високої продуктивності, масштабованості та надійності, що є критичними характеристиками для кеш-сервісів типу in-memory, у процесі проєктування системи Muninn було ретельно підібрано технологічний стек. Основним критерієм відбору технологій стала їхня здатність ефективно працювати з оперативною пам’яттю, підтримувати асинхронну модель виконання та забезпечувати мінімальні затримки при мережевій взаємодії у розподіленому середовищі [16].

В якості основної мови програмування для реалізації ядра системи, API-рівня та клієнтських бібліотек було обрано C# на платформі .NET 9.0. Дана платформа забезпечує високу продуктивність завдяки JIT-компіляції, оптимізованій роботі зі структурами пам’яті та сучасному збирачу сміття. Крім того, .NET надає розвинену підтримку асинхронного програмування через механізми async/await, що є особливо важливим для обробки великої кількості конкурентних операцій введення-виведення, зокрема мережевих запитів і операцій персистентного збереження даних [16].

Використання gRPC, побудованого на основі протоколу HTTP/2, як основного механізму мережевої взаємодії, зумовлене вимогами до низької латентності та високої пропускної здатності. На відміну від традиційних REST-рішень, що базуються на HTTP/1.1 та текстових форматах передачі даних, gRPC використовує бінарний формат і підтримує мультиплексування з’єднань. Це дозволяє значно зменшити накладні витрати при обробці великої кількості одночасних запитів і зробити систему більш стійкою до високих навантажень.

Ключовим елементом у реалізації gRPC-взаємодії є використання Protocol Buffers як механізму серіалізації даних. Даний формат забезпечує компактне бінарне представлення повідомлень, що суттєво зменшує обсяг мережевого трафіку та час серіалізації і десеріалізації. У контексті кеш-сервісу Muninn це дозволяє мінімізувати затримки при передачі значень між клієнтом і сервером, а також ефективно працювати з великими обсягами даних без

втрати продуктивності [16].

Таблиця 2.2 — Технологічний стек

Технологія	Роль у системі	Обґрунтування
Мова програмування C# (.NET 9.0)	Ядро, API, Клієнт	Висока продуктивність завдяки JIT-компіляції та оптимізованим операціям з пам'яттю. Вбудована підтримка асинхронності (async/await) є ідеальною для операцій I/O (диск, мережа).
gRPC (на основі HTTP/2)	API-рівень, Мережева взаємодія	Висока ефективність та швидкість передачі даних порівняно з традиційним HTTP/1.1 (REST/JSON) за рахунок використання бінарного формату Protocol Buffers. Це мінімізує затримку (latency) при великому навантаженні.
Protocol Buffers	Серіалізація даних (Payload)	Забезпечує динамічне бінарне кодування, що є компактнішим і швидшим за JSON/XML, зменшуючи розмір мережевого трафіку.
Docker/Kubernetes	Масштабування, Розгортання	Підтримує швидку контейнеризацію сервісу, що є необхідним для реалізації наукової новизни.

Важливою перевагою Protocol Buffers є також строгий контракт між клієнтом і сервером, який визначається у .proto файлах. Це зменшує ймовірність помилок інтеграції, спрощує підтримку клієнтських бібліотек і забезпечує сумісність між різними версіями сервісу. Такий підхід є особливо актуальним у розподілених системах, де кеш використовується багатьма незалежними сервісами.

Для забезпечення гнучкого розгортання та масштабування системи Muninn було обрано технології Docker та Kubernetes. Контейнеризація дозволяє ізолювати середовище виконання сервісу, спростити процес розгортання та забезпечити однакову поведінку системи на різних платформах. Це є важливим фактором як для експлуатації системи, так і для проведення експериментальних досліджень у межах магістерської роботи [25].

Використання Kubernetes, у свою чергу, надає можливості для горизонтального масштабування кеш-сервісу, автоматичного перерозподілу навантаження та підвищення відмовостійкості системи. Завдяки цьому Muninn може бути розгорнутий у кластерному середовищі з кількома екземплярами сервісу, що відповідає сучасним вимогам до високодоступних систем і створює основу для реалізації елементів наукової новизни, пов’язаних з динамічним балансуванням навантаження.

Таким чином, обраний технологічний стек є збалансованим рішенням, яке поєднує високу продуктивність, ефективну роботу з пам’яттю, сучасні мережеві протоколи та інструменти масштабування. Це дозволяє реалізувати кеш-сервіс Muninn як конкурентоспроможну платформу для використання у високонавантажених розподілених системах, а також створює надійну основу для подальших досліджень і розвитку системи [16].

2.4 Модель даних Entry

Об’єкт Entry є базовою структурною одиницею системи керування кешем Muninn та відіграє ключову роль у зберіганні як самих даних, так і супровідної

інформації, необхідної для коректної роботи кешу. На відміну від спрощених реалізацій key-value сховищ, де запис обмежується лише парою ключ-значення, у Muninn кожен запис інкапсулює розширений набір метаданих, що дозволяє ефективно керувати життєвим циклом об’єкта, підтримувати TTL, відстежувати модифікації та забезпечувати коректну десеріалізацію значень [17].

Проєктування класу Entry було виконано з урахуванням вимог до високопродуктивних in-memory систем. Основний акцент зроблено на мінімізацію накладних витрат при доступі до даних, передбачуваність структури об’єкта в пам’яті та швидкість виконання операцій читання і запису. Саме тому клас є sealed, що забороняє наслідування та дозволяє середовищу виконання .NET здійснювати додаткові оптимізації.

Конструктор класу Entry приймає ключ, значення у бінарному вигляді та інформацію про кодування. Такий підхід забезпечує гнучкість при роботі з різними форматами даних і дозволяє клієнтській стороні самостійно визначати спосіб серіалізації, не змінюючи внутрішню логіку кешу. У результаті Entry виступає універсальним контейнером для збереження довільних об’єктів.

Однією з важливих властивостей класу Entry є поле Hashcode, яке обчислюється на основі ключа. Збереження хеш-коду у самому об’єкті дозволяє оптимізувати операції пошуку та порівняння, зменшуючи кількість повторних обчислень під час роботи з хеш-таблицями in-memory кешу. Це особливо актуально при великій кількості конкурентних запитів [17].

Ключ (Key) у класі Entry є незмінним після створення об’єкта, що гарантує стабільність ідентифікації запису протягом усього часу його існування в кеші. Використання ініціалізатора init забезпечує неможливість зміни ключа після створення об’єкта, що запобігає логічним помилкам та порушенню цілісності структури даних.

Значення (Value) зберігається у вигляді масиву байтів (byte[]), що є ключовим рішенням для підтримки бінарної серіалізації. Такий формат дозволяє значно зменшити обсяг даних у пам’яті порівняно з текстовими представленнями та прискорити передачу значень між компонентами системи і мережею.

Використання бінарного представлення значень робить клас Entry незалежним від конкретної моделі даних. Система не накладає обмежень на структуру об’єкта, що кешується, а лише працює з байтовим потоком, що підвищує універсальність і масштабованість рішення [17].

Окрему роль відіграє властивість LifeTime, яка визначає час життя запису в кеші. Вона використовується механізмами керування TTL для автоматичного видалення застарілих записів, що дозволяє ефективно управляти обсягом оперативної пам’яті та запобігати її переповненню.

Значення часу життя зберігається у вигляді об’єкта TimeSpan, що дозволяє гнучко задавати інтервали часу з високою точністю. Це рішення спрощує реалізацію політик витіснення та забезпечує коректну роботу кешу в умовах змінного навантаження.

Для коректної десеріалізації значень у класі Entry зберігається інформація про кодування (Encoding). Це дозволяє клієнтській стороні однозначно визначити, у якому форматі було закодовано значення, та відновити його без втрати даних або помилок інтерпретації.

Збереження типу кодування разом зі значенням є особливо важливим у багатомовних та розподілених системах, де різні сервіси можуть використовувати різні стандарти представлення текстових даних. Такий підхід підвищує сумісність між компонентами системи.

Властивість CreationTime використовується для фіксації моменту створення запису у кеші. Збереження часу у форматі UTC дозволяє уникнути проблем, пов’язаних з часовими поясами, та забезпечує коректну синхронізацію в розподілених середовищах [17].

Інформація про час створення може використовуватися для аналітики, логування, а також для реалізації додаткових політик керування кешем, наприклад, витіснення на основі віку запису [28].

Властивість LastModificationTime зберігає дату та час останньої зміни запису. Це дозволяє відстежувати актуальність даних, контролювати оновлення значень та забезпечувати коректну роботу механізмів синхронізації.

Оновлення часу останньої модифікації є важливим для сценаріїв, у яких кеш використовується як проміжний шар між кількома сервісами, і необхідно визначати, які дані є найсвіжішими. Таким чином, клас Entry виступає не просто контейнером для зберігання даних, а повноцінною моделлю запису кешу, яка поєднує у собі дані та метаінформацію, необхідну для ефективної роботи системи.

На лістингу 2.1 представлено реалізацію класу Entry, що використовується у системі Muninn.

Лістинг 2.1 — Реалізація класу Entry

```
public sealed class Entry(string key, byte[] value, Encoding encoding)
{
   public int Hashcode { get; } = key.GetHashCode();
   public string Key { get; init; } = key;
   public byte[] Value { get; set; } = value;
   public TimeSpan LifeTime { get; set; } = TimeSpan.Zero;
   public DateTime CreationTime { get; init; } = DateTime.UtcNow;
   public DateTime LastModificationTime { get; set; } = DateTime.UtcNow;
   public Encoding Encoding { get; set; } = encoding;
}
```

Представлений лістинг демонструє компактну та водночас функціонально насичену реалізацію об’єкта запису кешу. Використання ініціалізаторів властивостей дозволяє зменшити кількість шаблонного коду та підвищити читабельність реалізації [18].

Завдяки чіткій структурі класу Entry, ядро системи Muninn може ефективно виконувати операції додавання, оновлення, видалення та фільтрації записів без необхідності звернення до зовнішніх структур метаданих.

Таке рішення позитивно впливає на продуктивність системи, оскільки всі необхідні дані для керування записом зосереджені в одному об’єкті, що зменшує кількість звернень до пам’яті [18].

У цілому, використання об’єкта Entry як базової одиниці зберігання дозволяє реалізувати гнучку, масштабовану та високопродуктивну модель кешування, яка відповідає вимогам сучасних розподілених систем і є придатною для подальшого розширення та наукових досліджень.

2.5 Проектування API-рівня

API-рівень у системі керування кешем Muninn виконує роль єдиної точки входу для зовнішніх клієнтів і сервісів, забезпечуючи доступ до функціональності кешу незалежно від внутрішньої реалізації ядра. Основним завданням даного рівня є приймання запитів, їх валідація, перетворення у внутрішній формат та передавання до ядра системи для подальшої обробки. Такий підхід дозволяє ізолювати бізнес-логіку керування кешем від деталей мережевої взаємодії [19].

Архітектурною особливістю API-рівня Muninn є наявність двох незалежних реалізацій інтерфейсу доступу, а саме gRPC та HTTP. Обидві реалізації працюють паралельно та не залежать одна від одної, використовуючи спільне ядро системи. Це дозволяє забезпечити гнучкість інтеграції з різними типами клієнтів і задовольнити вимоги як високопродуктивних внутрішніх сервісів, так і зовнішніх систем, для яких важливою є простота доступу [19].

Незалежність реалізацій API означає, що кожен протокол має власний набір транспортних механізмів, форматів повідомлень та контрактів взаємодії, проте всі вони транслюються у єдині виклики ядра Muninn. Завдяки цьому будь-які зміни у бізнес-логіці або механізмах зберігання даних не потребують модифікації клієнтських протоколів, що підвищує стабільність та розширюваність системи [19].

Використання двох API-рішень також дозволяє адаптувати систему до різних сценаріїв експлуатації. gRPC-інтерфейс орієнтований на високонавантажені та внутрішні мікросервісні взаємодії, де критичними є мінімальна затримка та ефективність передачі даних. HTTP-інтерфейс, у свою чергу, забезпечує простоту інтеграції, зручність тестування та сумісність з широким спектром клієнтських платформ і інструментів [19].

Обидві реалізації API підтримують однаковий набір базових операцій керування кешем, зокрема додавання, отримання, оновлення та видалення записів. Це гарантує функціональну еквівалентність інтерфейсів і дозволяє клієнтам обирати протокол доступу без втрати можливостей системи.

Таким чином, API-рівень Muninn є важливим архітектурним компонентом, який забезпечує універсальність, масштабованість та зручність використання кеш-сервісу. Детальні особливості реалізації кожного з протоколів розглядаються нижче.

2.5.1 gRPC та Protocol Buffers

Для забезпечення максимально можливої швидкості взаємодії між клієнтською та серверною частинами системи керування кешем Muninn було обрано технологію gRPC, що базується на протоколі HTTP/2, у поєднанні з Protocol Buffers (Protobuf) як мовою опису інтерфейсів та формату передавання даних. Такий підхід дозволяє значно зменшити затримки при обміні повідомленнями, мінімізувати обсяг мережевого трафіку та забезпечити високу пропускну здатність у порівнянні з традиційними REST-рішеннями на основі HTTP/1.1 та JSON [20].

Використання gRPC у Muninn особливо доцільне з огляду на in-memory природу системи, де швидкість доступу до даних є критичним фактором. Оскільки більшість операцій з кешем є короткотривалими та виконуються з високою частотою, будь-які накладні витрати на серіалізацію або мережеву взаємодію можуть суттєво вплинути на загальну продуктивність системи. gRPC мінімізує ці витрати завдяки бінарному формату повідомлень та ефективному управлінню з’єднаннями [20].

Контракт взаємодії між клієнтом і сервером у Muninn описується у файлі muninn_service.proto, який визначає перелік доступних методів та формати повідомлень запитів і відповідей. Даний файл є єдиним джерелом істини для обох сторін взаємодії та використовується для автоматичної генерації клієнтського і серверного коду.

У лістингу 2.2 наведено фрагмент .proto файлу, який описує gRPC-сервіс Muninn та набір базових операцій керування кешем.

Лістинг 2.2 — Опис gRPC-сервісу Muninn у файлі muninn_service.proto

```
syntax = "proto3";
```

```
import "google/protobuf/duration.proto";
option csharp_namespace = "Muninn.Grpc";
package Client;
service MuninnService {
  rpc Add (AddRequest) returns (AddReply);
  rpc Get (GetRequest) returns (GetReply);
  rpc Update (UpdateRequest) returns (UpdateReply);
  rpc Insert (InsertRequest) returns (InsertReply);
  rpc Delete (DeleteRequest) returns (DeleteReply);
  rpc DeleteAll (DeleteAllRequest) returns (DeleteAllReply);
}
```

Поданий лістинг демонструє визначення сервісу MuninnService, який містить повний набір CRUD-операцій для роботи з кешем. Кожен метод описується у вигляді RPC-виклику, що приймає строго типізований об’єкт запиту та повертає відповідний об’єкт відповіді. Така модель забезпечує чіткий контракт, типову безпеку та спрощує підтримку клієнтських бібліотек.

Важливою особливістю даної реалізації є використання типу bytes для передачі значень, що дозволяє зберігати та передавати дані у бінарному вигляді без прив’язки до конкретного формату. Це повністю відповідає внутрішній архітектурі Muninn, де всі значення зберігаються як масиви байтів, а відповідальність за їх інтерпретацію покладається на клієнтську сторону [20].

На основі .proto файлу за допомогою інструментів gRPC автоматично генерується серверний та клієнтський код мовою C#. Серверна частина реалізує згенерований базовий клас та перевизначає відповідні методи для обробки запитів. Приклад реалізації одного з таких методів наведено у лістингу 2.3.

Лістинг 2.3 — Реалізація методу Add у gRPC-сервісі Muninn

```
public override async Task<AddReply> Add(AddRequest request, ServerCallContext context)
{
    var encoding = Encoding.GetEncoding(request.EncodingName);
    var entry = new Entry(
        request.Key,
        request.Value.ToByteArray(),
        encoding
    );
    var lifeTime = request.LifeTime.ToTimeSpan();
```

```
    if (lifeTime > TimeSpan.Zero)
    {
        entry.LifeTime = lifeTime;
    }
    var result = await _cacheManager.AddAsync(
        entry,
        context.CancellationToken
    );
    var reply = new AddReply
    {
        IsSuccessful = result.IsSuccessful,
        Value = result.IsSuccessful
            ? entry.Value.ToByteString()
            : encoding.GetBytes(result.Message).ToByteString()
    };
    return reply;
}
```

У наведеному прикладі показано типовий сценарій обробки gRPC-запиту на стороні сервера. Отримані від клієнта дані перетворюються у внутрішній об’єкт Entry, який використовується ядром системи для подальшого зберігання в кеші. Інформація про кодування, передана клієнтом, застосовується для коректної обробки значення та формування відповіді.

Метод реалізовано у асинхронному стилі з використанням async/await, що дозволяє ефективно обробляти велику кількість одночасних запитів без блокування потоків виконання. Передача CancellationToken забезпечує коректне завершення операції у разі скасування запиту клієнтом або перевищення тайм-ауту [20].

Формування відповіді (AddReply) здійснюється з урахуванням результату виконання операції. У разі успіху клієнту повертається збережене значення, у протилежному випадку — повідомлення про помилку у бінарному вигляді з використанням відповідного кодування. Такий підхід зберігає узгодженість формату даних незалежно від результату операції [20].

Таким чином, використання gRPC та Protocol Buffers у системі Muninn дозволяє реалізувати високопродуктивний, типобезпечний та масштабований API, який органічно поєднується з in-memory архітектурою кешу та відповідає вимогам сучасних розподілених систем.

2.5.2 HTTP Restful service

Система Muninn, хоча й спроектована з акцентом на високопродуктивну бінарну комунікацію, переважно за допомогою протоколу gRPC, передбачає необхідність підтримки широкого спектру клієнтів. Саме з цією метою був інтегрований модуль Muninn.Api, який реалізує традиційний HTTP REST API. Цей додатковий інтерфейс є критично важливим для забезпечення універсальної сумісності Він дозволяє взаємодіяти з кеш-сховищем Muninn навіть тим клієнтам, які не мають вбудованої підтримки сучасних протоколів, як-от gRPC.

Типовими прикладами таких клієнтів є звичайні веб-браузери, що виконують AJAX-запити, або застарілі мікросервіси в рамках великої корпоративної архітектури. Таким чином, REST API діє як міст, що розширює охоплення системи Muninn, дозволяючи інтегрувати її в будь-яке існуюче IT-середовище без значних модифікацій на стороні клієнта [21].

Ключовим рішенням для цього API стало використання формату JSON (JavaScript Object Notation) для серіалізації та десеріалізації даних, що передаються. Вибір JSON є прагматичним, оскільки цей формат забезпечує виняткову легкість інтеграції та налагодження (debugging)

На відміну від бінарних форматів, які вимагають спеціалізованих інструментів, JSON є людинозрозумілим і може бути легко перевірений у будь-якому текстовому редакторі чи інструменті розробника браузера. Ця читабельність значно прискорює процес розробки, особливо на етапах тестування та виправлення помилок, які можуть виникати під час взаємодії між різними системами.

Усі маршрути (ендпоінти) REST API системи Muninn централізовано визначені у спеціальному файлі Register.cs модуля Muninn.Api.

Цей файл містить статичний метод розширення MapEndpoints, який приймає об'єкт IEndpointRouteBuilder та використовується для конфігурації маршрутизатора в Minimal API середовищі .NET.

Спершу, для організації структури, створюється група маршрутів за допомогою виклику app.MapGroup("muninn"). Префікс "muninn" забезпечує логічне групування всіх операцій кешування під єдиною базовою URL-адресою. У наступному лістингу детально показано, як ця група маршрутів конфігурується для підтримки повного набору операцій CRUD (Create, Read, Update, Delete) над кешем.

Лістинг 2.4 — Реєстрація REST ендпоінтів у Muninn.Api

```
public static IEndpointRouteBuilder MapEndpoints(this IEndpointRouteBuilder
app)
{
    var group = app.MapGroup("muninn");
    group.MapPost("{key}", PostAsync);
    group.MapPost("{key}/insert", InsertAsync);
    group.MapPut("{key}", PutAsync);
    group.MapDelete("{key}", DeleteAsync);
    group.MapDelete(string.Empty, DeleteAllAsync);
    group.MapGet("{key}", GetAsync);

    return group;
}
```

Операції HTTP REST API чітко відповідають стандартним HTTP-методам, що робить інтерфейс інтуїтивно зрозумілим для розробників. Метод MapPost("{key}", PostAsync) відповідає за створення нового запису або оновлення існуючого, використовуючи HTTP-метод POST та ключ, переданий як частина URL-шляху [21].

Також передбачено більш явний метод вставки — MapPost("{key}/insert", InsertAsync), який може використовуватися для умовного додавання запису. Оновлення запису, згідно з REST-конвенцією, обробляється методом MapPut("{key}", PutAsync), що використовує ідемпотентний HTTP-метод PUT. Для отримання даних використовується метод MapGet("{key}", GetAsync), що асоційовано з HTTP-методом GET, який є безпечним і не змінює стану сервера. Видалення одного конкретного елемента здійснюється за допомогою MapDelete("{key}", DeleteAsync), застосовуючи HTTP-метод DELETE.

Крім того, система пропонує можливість видалення всіх записів у кеші, що реалізовано через MapDelete(string.Empty, DeleteAllAsync), забезпечуючи функціональність масового очищення.

Розглянемо детальніше внутрішню реалізацію одного з ключових ендпоінтів — методу PostAsync, який відповідає за додавання або оновлення даних у кеші. Цей метод є асинхронним (async Task<IResult>) і приймає кілька залежностей та параметрів, використовуючи механізми ін'єкції залежностей .NET. Ключовим вхідним параметром є об'єкт [FromBody] PostRequest request, який слугує як DTO (Data Transfer Object). Цей DTO автоматично заповнюється даними з тіла вхідного HTTP-запиту, які були серіалізовані у форматі JSON. Крім того, метод отримує [FromServices] ICacheManager cacheManager — основний інтерфейс для взаємодії з кеш-сховищем, а також CancellationToken для підтримки скасування операції.

Лістинг 2.5 —Приклад реалізації REST ендпоінту PostAsync

```
private static async Task<IResult> PostAsync([FromBody] PostRequest request,
   [FromServices] ICacheManager cacheManager, CancellationToken cancellationToken)
{
   var encoding = Encoding.GetEncoding(request.Body.EncodingName);
  var entry = new Entry(request.Key, encoding.GetBytes(request.Body.Value), encoding);
   var lifeTime = request.Body.LifeTime;

   if (lifeTime > TimeSpan.Zero)
   {
      entry.LifeTime = lifeTime;
   }

   var result = await cacheManager.AddAsync(entry, cancellationToken);

   return GetResponse(result, true);
}
```

Усередині методу PostAsync відбувається критично важливий етап трансформації даних. Дані, отримані в DTO (PostRequest), повинні бути

перетворені у внутрішній об'єкт домену системи Muninn — Entry. Спершу визначається кодування для значення кешу на основі властивості request.Body.EncodingName, забезпечуючи коректну обробку різних наборів символів.

Далі створюється екземпляр класу Entry, використовуючи request.Key для ідентифікації, і двійкове представлення значення, отримане шляхом кодування request.Body.Value. Об'єкт Entry є центральною одиницею зберігання в Muninn, інкапсулюючи ключ, значення (як масив байтів), інформацію про кодування та метадані. Також з DTO витягується бажаний час життя запису — request.Body.LifeTime (типу TimeSpan).

Механізм управління часом життя (Time-To-Live, TTL) є невід'ємною частиною будь-якої кешувальної системи. У методі PostAsync перевіряється, чи було встановлено клієнтом позитивне значення часу життя. Це відбувається через умову if (lifeTime > TimeSpan.Zero), яка гарантує, що час життя буде застосовано лише у випадку, якщо клієнт явно бажає обмежити термін існування запису. Якщо час життя є позитивним, він присвоюється відповідній властивості внутрішнього об'єкта: entry.LifeTime = lifeTime;.

Після успішної трансформації та конфігурації об'єкта Entry управління передається на рівень бізнес-логіки. Виклик await cacheManager.AddAsync(entry, cancellationToken) делегує фактичну операцію додавання або оновлення запису в кеші реалізації інтерфейсу ICacheManager. Цей підхід забезпечує чисту архітектуру, де API-шар відповідає лише за парсинг HTTP-запиту, валідацію DTO та мапінг, але не за деталі внутрішньої роботи кешу.

Менеджер кешу (ICacheManager) відповідає за критичні операції, такі як взаємодія з базовим сховищем, обробка конкурентного доступу та логіка витіснення записів. Після завершення асинхронної операції додавання (AddAsync) метод отримує об'єкт result, який містить інформацію про успішність та деталі виконання.

Фінальним кроком є формування відповіді HTTP за допомогою допоміжного методу GetResponse(result, true). Цей метод, ймовірно, відповідає

за перетворення внутрішнього результату виконання на відповідний HTTP-статус-код (наприклад, 201 Created або 200 OK) та серіалізацію відповіді у форматі JSON [22].

Таким чином, REST API системи Muninn забезпечує повноцінний, структурований та архітектурно відокремлений механізм для взаємодії з кеш-сховищем, гармонійно доповнюючи високопродуктивний інтерфейс gRPC.

3 ПРОГРАМНА РЕАЛІЗАЦІЯ ТА ЕКСПЕРИМЕНТАЛЬНЕ ДОСЛІДЖЕННЯ MUNINN

3.1 Реалізація API-рівня для gRPC-інтерфейсу

API-рівень системи Muninn є ключовим компонентом, який забезпечує взаємодію між клієнтськими застосунками та внутрішнім ядром кеш-сервісу. У межах даної роботи особлива увага приділяється реалізації gRPC-інтерфейсу, оскільки саме він орієнтований на сценарії з високим навантаженням та вимогами до мінімальної затримки обробки запитів.

gRPC API реалізований у вигляді окремого сервісу Muninn.Api.Grpc, побудованого на базі платформи ASP.NET Core з підтримкою gRPC. Використання даної платформи дозволяє інтегрувати gRPC-сервіс у загальну екосистему .NET, скористатися вбудованими механізмами залежностей, логування, конфігурації та асинхронної обробки запитів.

Архітектурно gRPC API у Muninn реалізовано як незалежний сервіс, який використовує ядро системи виключно через абстракції. Такий підхід дозволяє повністю відокремити мережевий рівень від бізнес-логіки кешування та забезпечити можливість заміни або розширення API без впливу на внутрішні механізми зберігання даних.

Усі публічні методи, доступні клієнтам через gRPC, визначені у файлі контракту muninn_service.proto. Даний файл описує повний набір операцій керування кешем, а також структури запитів і відповідей. Саме контракт є основою для генерації серверного та клієнтського коду і гарантує узгодженість між сторонами взаємодії. Завдяки використанню механізму Source Generation, файл .proto автоматично перетворюється у набір C#-класів, які включають базовий клас сервісу MuninnServiceBase, моделі повідомлень та клієнтські проксі. Це значно зменшує обсяг ручної роботи та мінімізує ймовірність помилок, пов’язаних з несинхронністю контрактів [23].

Клас MuninnService, що реалізує gRPC API, наслідується від автозгенерованого класу MuninnServiceBase та перевизначає всі RPC-методи,

визначені у контракті. Таким чином, серверна логіка чітко відповідає визначеним інтерфейсам і не допускає відхилень від контракту.

Реалізація gRPC API у Muninn побудована за принципом «тонкого прошарку». Це означає, що сервіс не містить складної бізнес-логіки, а виконує виключно функції трансляції мережевих викликів у внутрішні операції ядра системи. Такий підхід сприяє спрощенню коду, підвищує його підтримуваність та зменшує зв’язаність компонентів.

Основні функції gRPC API-рівня можна звести до таких етапів:

- приймання та первинної обробки запитів від клієнтів;
- перетворення отриманих даних у внутрішні об’єкти системи;
- делегування виконання операцій до ядра кешу;
- формування та повернення відповіді клієнту у бінарному форматі.

Під час приймання запиту gRPC-сервіс отримує дані у вигляді бінарних повідомлень Protocol Buffers. Це дозволяє уникнути додаткових витрат на парсинг текстових форматів та забезпечити високу швидкість обробки навіть при великому обсязі трафіку [23].

Отримані з мережі дані перетворюються у внутрішні структури системи, зокрема у об’єкти типу Entry, які інкапсулюють ключ, значення та метадані. Інформація про кодування та час життя запису передається з клієнтської сторони та використовується для коректного формування внутрішнього стану кешу.

Після створення внутрішнього об’єкта відповідна операція делегується до інтерфейсу ICacheManager, який є основним фасадом ядра системи Muninn. Саме CacheManager відповідає за вибір конкретної реалізації кешу, управління життєвим циклом записів та взаємодію з механізмами персистентності.

Використання інтерфейсу ICacheManager у gRPC API дозволяє повністю абстрагуватися від деталей реалізації кешу. API-рівень не має інформації про те, чи зберігаються дані виключно в оперативній пам’яті, чи додатково синхронізуються з диском, що відповідає принципам слабкої зв’язаності та інверсії залежностей.

Усі методи gRPC-сервісу реалізовані в асинхронному стилі з використанням async/await. Це дозволяє обслуговувати велику кількість конкурентних клієнтських запитів без блокування потоків виконання та максимально ефективно використовувати ресурси сервера. Особливу увагу приділено підтримці CancellationToken, який передається з контексту gRPC-виклику до ядра системи. Це забезпечує можливість коректного скасування операцій у разі переривання з'єднання або перевищення часу очікування.

Результат виконання кожної операції повертається у вигляді об'єкта MuninnResult, який містить інформацію про успішність виконання та додаткові повідомлення. Даний результат серіалізується у формат Protocol Buffers та надсилається клієнту як відповідь.

Формування відповіді виконується з урахуванням бінарної природи системи Muninn. Усі дані, включно з повідомленнями про помилки, передаються у вигляді масивів байтів, що зберігає узгодженість формату незалежно від типу результату [23].

Використання Protocol Buffers забезпечує компактність повідомлень і зменшує навантаження на мережу, що є особливо важливим у сценаріях інтенсивної взаємодії між сервісами у мікросервісній архітектурі.

Окремою перевагою gRPC-інтерфейсу є строгий контракт між клієнтом і сервером, який унеможливлює передачу некоректних або неповних даних. Це знижує кількість помилок на етапі інтеграції та спрощує супровід системи. Завдяки чіткій структурі gRPC API реалізація нових методів або розширення існуючих не потребує значних змін у клієнтському коді, оскільки більшість змін автоматично відображається через оновлення .proto контракту.

У контексті експериментального дослідження gRPC API Muninn використовувався як основний інтерфейс для вимірювання продуктивності, затримок обробки запитів та стабільності роботи системи під навантаженням [24].

Отримані результати підтвердили доцільність обраного підходу, оскільки gRPC-інтерфейс продемонстрував низьку латентність, стабільну роботу та ефективне використання ресурсів сервера.

Таким чином, реалізація API-рівня для gRPC-інтерфейсу у системі Muninn є ключовим елементом, який забезпечує високу продуктивність, масштабованість та надійність сервісу, а також створює основу для подальших експериментальних досліджень та розвитку системи.

3.2 Реалізація API-рівня для HTTP Restful service

HTTP REST API у системі Muninn реалізовано як альтернативний інтерфейс доступу до функціональності кеш-сервісу, орієнтований на широку сумісність та простоту інтеграції з зовнішніми системами. На відміну від gRPC-інтерфейсу, який призначений передусім для високопродуктивних міжсервісних взаємодій, HTTP REST API забезпечує універсальний доступ до кешу за допомогою стандартних HTTP-запитів.

Реалізація REST API виконана в окремому модулі Muninn.Api з використанням фреймворку ASP.NET Core Minimal APIs. Даний підхід дозволяє створювати легковагові HTTP-сервіси з мінімальною кількістю шаблонного коду, зберігаючи при цьому повну функціональність платформи ASP.NET Core, зокрема механізми маршрутизації, залежностей, логування та обробки помилок. Однією з ключових особливостей Minimal APIs є відмова від класичної MVC-архітектури з контролерами на користь безпосереднього опису HTTP-маршрутів. У контексті Muninn це дозволило реалізувати REST API як тонкий прошарок між клієнтом та ядром кешу без надлишкових абстракцій [24].

Основна логіка маршрутизації HTTP-запитів зосереджена у класі Endpoints.cs, який виконує роль центральної точки конфігурації REST API. У даному класі всі HTTP-ендпоінти описуються явно та напряму прив’язуються до методів інтерфейсу ICacheManager, що забезпечує єдність логіки між різними API-рівнями.

Кожен HTTP-маршрут відповідає конкретній операції керування кешем, такій як додавання, отримання, оновлення або видалення записів. Це дозволяє зберегти функціональну еквівалентність між REST та gRPC інтерфейсами і надати клієнтам можливість обирати протокол взаємодії без втрати можливостей [25].

У REST API Muninn використовується класичний підхід до організації маршрутів, де тип операції визначається HTTP-методом, а ресурс ідентифікується шляхом. Така модель відповідає загальноприйнятим REST-принципам і спрощує використання API сторонніми розробниками.

Передача даних між клієнтом і сервером у REST API здійснюється з урахуванням бінарної природи ядра Muninn. Хоча HTTP традиційно асоціюється з текстовими форматами, реалізація Muninn дозволяє передавати значення у вигляді масивів байтів, що мінімізує втрати продуктивності та зберігає узгодженість з внутрішньою моделлю даних [26].

Особливу увагу приділено обробці кодування значень, яке передається клієнтом разом із запитом. Це дозволяє коректно серіалізувати та десеріалізувати дані незалежно від формату та забезпечує підтримку багатомовних і гетерогенних клієнтських середовищ.

REST API також підтримує передачу параметрів, пов’язаних із життєвим циклом записів, зокрема часу життя (TTL). Дані параметри використовуються для коректного формування об’єктів Entry та подальшого керування ними на рівні кешу.

Як і у випадку gRPC, REST API не містить складної бізнес-логіки. Усі операції після первинної обробки параметрів делегуються до ICacheManager, що гарантує єдину поведінку системи незалежно від обраного протоколу доступу.
Асинхронна модель обробки HTTP-запитів у ASP.NET Core дозволяє ефективно обслуговувати велику кількість одночасних клієнтів. Усі ендпоінти реалізовані з використанням асинхронних методів, що зменшує блокування потоків та підвищує масштабованість сервісу [26].

У REST API реалізовано стандартизовану обробку помилок і результатів виконання операцій. Кожна відповідь містить інформацію про успішність виконання запиту, а у випадку помилки — відповідне повідомлення, що дозволяє клієнту коректно реагувати на різні сценарії.

Перевагою HTTP REST API є можливість легкого тестування та налагодження за допомогою стандартних інструментів, таких як браузери, Postman або curl. Це робить даний інтерфейс зручним для розробників та адміністраторів системи.

REST API Muninn також добре інтегрується з існуючими веб-застосунками, проксі-серверами та балансувальниками навантаження, що спрощує його використання у корпоративних та хмарних середовищах. З точки зору продуктивності REST API поступається gRPC-інтерфейсу через додаткові накладні витрати HTTP та обробки заголовків. Проте у багатьох практичних сценаріях ця різниця не є критичною і компенсується універсальністю та простотою інтеграції [27].

У межах експериментального дослідження REST API використовувався для порівняльного аналізу з gRPC-інтерфейсом, зокрема для оцінки затримок обробки запитів та стабільності роботи під навантаженням. Отримані результати показали, що REST API є доцільним вибором для клієнтів із невисокими вимогами до латентності, а також для інтеграції з системами, які не підтримують gRPC [27].

Таким чином, реалізація HTTP REST API у системі Muninn доповнює gRPC-інтерфейс, забезпечуючи гнучкість, сумісність та зручність використання кеш-сервісу у різноманітних сценаріях експлуатації.

3.3 Програмна реалізація ключових модулів Muninn.Kernel

3.3.1 Реалізація додавання та оновлення записів

На архітектурному рівні системи Muninn центральним компонентом, відповідальним за керування життєвим циклом кешованих даних та координацію процесів їх збереження, є модуль CacheManager. Даний модуль виконує функцію

оркестратора, який узгоджує роботу різних рівнів кешування та забезпечує баланс між високою продуктивністю, надійністю та розширюваністю системи. Основним завданням CacheManager є централізоване керування всіма операціями модифікації даних, такими як додавання, вставка, оновлення та видалення записів. Завдяки такій централізації досягається уніфікована логіка обробки запитів незалежно від способу доступу (gRPC або HTTP) та конфігурації системи [28].

Усі операції модифікації, ініційовані клієнтськими застосунками, обов'язково проходять через компонент ResidentCache. ResidentCache являє собою основний in-memory рівень кешування, що працює безпосередньо в оперативній пам'яті та забезпечує мінімальні затримки при читанні та записі даних. Такий підхід відповідає класичній багаторівневій архітектурі сховищ, де найбільш часто використовувані дані зберігаються у найшвидшому доступному середовищі.

Використання ResidentCache дозволяє системі Muninn гарантувати надзвичайно високу пропускну здатність і низьку латентність, що є критично важливим для кеш-сервісів, орієнтованих на високонавантажені розподілені системи. Успішний запис у ResidentCache означає, що дані миттєво стають доступними для подальших операцій читання.

Паралельно з роботою ResidentCache, CacheManager ініціює асинхронну обробку додаткових дій, зокрема збереження даних у PersistentCache. PersistentCache відповідає за довготривале зберігання даних, зазвичай із використанням дискових ресурсів, що дозволяє відновити стан кешу після перезапуску процесу, контейнера або вузла в кластері.

Таким чином, CacheManager реалізує комбінований підхід до запису даних, який концептуально відповідає патернам *write-through* або *write-behind*, залежно від конкретної конфігурації. Ключовою особливістю є те, що основний шлях виконання операції не блокується очікуванням завершення повільних I/O-операцій, пов'язаних із персистенцією.

Архітектура Muninn передбачає можливість гнучкого налаштування поведінки системи. Зокрема, персистентний рівень може бути повністю вимкнений, у результаті чого система функціонуватиме виключно як RAM-кеш. Така конфігурація є доцільною у сценаріях, де допустима втрата даних після перезапуску, але критичною є максимальна швидкість обробки запитів.

Нижче наведено приклад реалізації однієї з ключових операцій модифікації — асинхронного методу AddAsync у модулі CacheManager. Даний лістинг демонструє базовий принцип роботи компонента: пріоритетний запис у ResidentCache з подальшою ініціацією фонових операцій.

Лістинг 3.1 — Приклад реалізації операції AddAsync у CacheManager

```
public async Task<MuninnResult> AddAsync(Entry entry, CancellationToken cancellationToken)
{
   var residentResult = await _residentCache.AddAsync(entry, cancellationToken);

   if (residentResult.IsSuccessful)
   {
      _ = _handlers.Select(handler => handler.AddAsync(entry, cancellationToken));
   }
   return residentResult;
}
```

Процес виконання операції AddAsync складається з декількох послідовних етапів. Першим і обов’язковим кроком є асинхронний запис даних у ResidentCache. Даний етап виконується за допомогою виклику методу _residentCache.AddAsync(...). Успішне завершення цієї операції гарантує, що запис негайно збережений в оперативній пам’яті та доступний для подальших звернень.

Другим етапом є перевірка результату виконання операції. Якщо запис у ResidentCache завершився успішно, CacheManager переходить до ініціації додаткових дій. У випадку помилки подальша обробка не виконується, а клієнту повертається відповідний результат.

Третім етапом є виклик опціональних хендлерів, що зберігаються у колекції _handlers. Виклик виконується асинхронно без очікування завершення (fire-and-forget), що дозволяє уникнути блокування основного потоку виконання. Саме цей механізм забезпечує високу продуктивність системи, оскільки клієнт отримує підтвердження успішного запису одразу після оновлення in-memory стану.

Розглянемо механізм розширення: опціональні хендлери. Колекція _handlers реалізує механізм розширення функціональності системи Muninn. Кожен елемент цієї колекції є реалізацією інтерфейсу IOptionalCacheHandler та викликається у випадку успішного виконання базових операцій.

Опціональні хендлери дозволяють інкапсулювати додаткову логіку, не ускладнюючи основний код CacheManager. Зокрема, такі хендлери можуть виконувати:

- ініціацію запису даних у PersistentCache;
- оновлення допоміжних або вторинних індексів;
- відправку подій або повідомлень іншим сервісам;
- логування або аудит змін даних.

Завдяки цьому CacheManager залишається компактним і сфокусованим на своїй основній ролі — координації кешування.4

Реєстрація опціональних компонентів здійснюється за допомогою контейнера впровадження залежностей (Dependency Injection Container). Для цього використовується метод-розширення AddOptionalCache, який викликається під час ініціалізації застосунку.

Лістинг 3.2 — Реєстрація опціональних компонентів у DI Container

```
public static IServiceCollection AddOptionalCache(this IServiceCollection services, string args)
{
   if (args.Contains("--sort"))
   {
      services.AddSingleton<ISortedResidentCache, SortedResidentCache>();
      services.TryAddEnumerable(new ServiceDescriptor(typeof(IBaseCache),
         typeof(SortedResidentCache), ServiceLifetime.Singleton));
      services.TryAddEnumerable(new
```

```
    ServiceDescriptor(typeof(IOptionalCacheHandler),
            typeof(SortedResidentCacheHandler), ServiceLifetime.Singleton));
      }

      if (args.Contains("--persistent"))
      {
        services.AddSingleton<IPersistentCache, PersistentCache>();
        services.TryAddEnumerable(new ServiceDescriptor(typeof(IBaseCache),
          typeof(PersistentCache), ServiceLifetime.Singleton));
        services.TryAddEnumerable(new
    ServiceDescriptor(typeof(IOptionalCacheHandler),
            typeof(PersistentCacheHandler), ServiceLifetime.Singleton));
      }

      return services;
    }
```

Використання аргументів командного рядка дозволяє динамічно змінювати конфігурацію системи без перекомпіляції коду. Наприклад, параметр --sort активує підтримку сортованого кешу, тоді як параметр --persistent вмикає персистентний рівень зберігання.

Метод TryAddEnumerable забезпечує коректне збирання всіх зареєстрованих реалізацій інтерфейсу IOptionalCacheHandler у єдину колекцію, яка автоматично інжектується в CacheManager. Це дозволяє легко розширювати систему новими модулями без зміни існуючої логіки.

Узагальнюючи, даний підхід поєднує патерни Декоратор, Стратегія та Dependency Injection, формуючи гнучку, модульну та масштабовану архітектуру кешування, що повністю відповідає вимогам високонавантажених систем.

Також існує Background service для очистки даних, у яких термін життя, підійшов до кінця. Це необхідна частина імплементації, для того, щоб не було величезних затрат пам’яті, а також уповільнення часу виконання операцій. У додатку Д наведено приклад реалізації Background Service для очистки даних, чий час життя завершився. Лістинг представляє реалізацію фонової служби CacheLifeTimeBackgroundService, яка відповідає за автоматичне очищення кешу від записів із завершеним терміном життя (TTL). Наявність такого сервісу є критично важливою для довготривалої стабільної роботи in-memory системи,

оскільки без регулярного видалення застарілих даних можливе неконтрольоване зростання споживання оперативної пам’яті [28].

Даний клас наслідується від BackgroundService, що є стандартним механізмом ASP.NET Core для реалізації довготривалих фонових процесів. Основна логіка сервісу зосереджена у перевизначеному методі ExecuteAsync, який виконується у окремому потоці протягом усього життєвого циклу застосунку.

У кожній ітерації циклу сервіс отримує поточний стан кешу шляхом виклику методу _cacheManager.GetAllAsync. Після цього виконується фільтрація записів, для яких визначено ненульовий термін життя та перевищено момент його завершення, що обчислюється як сума часу останньої модифікації та значення TTL.

Якщо такі записи знайдено, сервіс ініціює їх видалення з використанням паралельного виконання. Кількість одночасних потоків обмежується значенням MaxDegreeOfParallelism, що дозволяє уникнути надмірного навантаження на систему при великій кількості прострочених записів.

Паралельне видалення реалізоване за допомогою Parallel.ForEachAsync, що забезпечує ефективну очистку кешу без значного впливу на основні операції читання та запису. Після завершення циклу обробки сервіс переходить у режим очікування на фіксований інтервал часу, заданий змінною _delayTime.

У випадку зупинки застосунку або отримання сигналу скасування, фоновий сервіс коректно завершує свою роботу та записує відповідне повідомлення у журнал подій. Це забезпечує передбачувану поведінку системи під час завершення роботи [29].

Таким чином, реалізація фонової служби очищення кешу доповнює загальну архітектуру Muninn, забезпечуючи автоматичне керування життєвим циклом записів, оптимальне використання пам’яті та стабільну продуктивність системи в умовах тривалої експлуатації.

Згідно лістингом коду у додатку В, іде перевірка усіх даних, чи їхній LifeTime не дорівнює нулю, це було зроблено з метою, надати користувачам

можливість зберігати деякі дані постійно, тобто у цих даних не буде терміна життя.

Але основна бізнес логіка знаходиться у класі ResidentCache. Основний принцип роботи цього класу – це збереження усіх даних в оперативній пам'яті. Для цього був розроблений великий функціонал методів. ResidentCache наслідується від абстрактного класу BaseCache<T>, який має в собі загальні наслідуюмі приватні поля та логіку, що зображена у додатку Д.

Абстрактний клас BaseCache<TSelf> є фундаментальним елементом архітектури кешування у системі Muninn та виконує роль спільного базового шару для всіх реалізацій кешів. Його основне призначення полягає в інкапсуляції загальної функціональності, яка не залежить від конкретного способу зберігання даних, але є необхідною для коректної роботи будь-якого кеш-рівня в системі [29].

Клас оголошений як узагальнений (generic) з параметром типу TSelf, що дозволяє коректно використовувати типізований логер ILogger<TSelf>. Такий підхід спрощує трасування подій у журналі та дозволяє чітко ідентифікувати джерело повідомлень логування, що особливо важливо для складних багатокомпонентних систем.

У конструкторі BaseCache здійснюється ін'єкція двох ключових залежностей: механізму логування та сервісу фільтрації. Інтерфейс IFilterService використовується для реалізації складних операцій пошуку та фільтрації записів за ключами або значеннями, що забезпечує повторне використання цієї логіки у різних типах кешів.

Важливою складовою базового класу є набір констант, які визначають початкові та динамічні параметри роботи кешу. Зокрема, InitialArraySize задає стартовий розмір внутрішніх структур даних, що дозволяє зменшити кількість перевиділень пам’яті при старті системи. Значення DefaultIncreaseValue використовується для керування зростанням внутрішніх колекцій, що позитивно впливає на продуктивність при великій кількості записів.

Для забезпечення потокобезпечної роботи з даними у базовому класі

використовується SemaphoreSlim. Даний механізм синхронізації дозволяє контролювати доступ до критичних секцій коду в асинхронному середовищі, не блокуючи потоки повністю, що є особливо важливим для високонавантажених in-memory систем.

Методи GetCancelledResult реалізують уніфіковану обробку сценаріїв скасування операцій. У системі Muninn активно використовується механізм CancellationToken, що дозволяє клієнтам або внутрішнім сервісам коректно переривати довготривалі операції. Базовий клас централізує логіку формування результату для таких випадків, забезпечуючи єдину поведінку у всіх реалізаціях кешу.

Під час скасування операції відповідна інформація фіксується у журналі за допомогою логера, після чого формується об’єкт MuninnResult, який містить ознаку невдалого виконання, повідомлення про скасування та виняток. Це дозволяє клієнтській стороні однозначно визначити причину завершення операції.

Метод GetSuccessfulResult використовується для формування успішного результату виконання операції. Він є статичним та повертає об’єкт MuninnResult, який може містити або не містити екземпляр Entry, залежно від типу операції. Такий підхід зменшує дублювання коду у похідних класах та стандартизує формат відповіді.

Аналогічно, метод GetFailedResult інкапсулює створення результату з помилкою. Він дозволяє передати текст повідомлення, ознаку скасування та виняток, що значно спрощує обробку помилок у бізнес-логіці кешу та підвищує читабельність коду.

Завдяки наявності BaseCache<TSelf> усі конкретні реалізації кешів у Muninn мають спільний набір інструментів для логування, синхронізації, обробки помилок та формування результатів. Це зменшує кількість повторюваного коду та забезпечує єдині правила поведінки системи.

Основна бізнес-логіка роботи з даними реалізована у класі ResidentCache, який наслідується від BaseCache<T>. Даний клас відповідає за зберігання та

керування всіма активними записами кешу безпосередньо в оперативній пам’яті, що забезпечує мінімальний час доступу до даних.

ResidentCache використовує успадковані механізми синхронізації та логування, доповнюючи їх власними структурами даних для ефективного зберігання записів типу key-value. Усі операції читання, додавання, оновлення та видалення виконуються без звернення до дискових ресурсів, що робить цей компонент критичним для досягнення високої продуктивності системи.

Наслідування від BaseCache також дозволяє ResidentCache автоматично підтримувати фільтрацію даних, обробку скасування операцій та стандартизоване формування результатів. Це забезпечує чисту архітектуру, у якій бізнес-логіка чітко відокремлена від допоміжних технічних аспектів.

Таким чином, абстрактний клас BaseCache<TSelf> виконує роль стабільного архітектурного фундаменту, на якому побудовані всі ключові компоненти кешування у системі Muninn. Його використання дозволяє досягти високої узгодженості реалізацій, спростити супровід коду та забезпечити можливість подальшого розширення функціональності без порушення існуючої логіки.

3.3.2 Механізм динамічного розширення In-Memory сховища ResidentCache

Однією з ключових особливостей реалізації класу ResidentCache є використання власного механізму керування пам’яттю, що базується на масивах фіксованого розміру з можливістю динамічного розширення. Такий підхід дозволяє мінімізувати накладні витрати, притаманні стандартним колекціям, та забезпечити передбачувану продуктивність при роботі з великими обсягами кешованих даних [30].

У процесі додавання або пошуку елементів у кеші центральну роль відіграє метод TryFindIndex(int hashcode, out int freeIndex). Даний метод виконує дві взаємопов’язані функції: по-перше, він намагається знайти вже існуючий

елемент за хешкодом ключа, а по-друге — визначає наявність вільної позиції у внутрішньому масиві для потенційного додавання нового запису.

Логіка роботи TryFindIndex залежить від контексту виклику. У сценаріях додавання нового запису наявність елемента з таким самим ключем розглядається як помилка, оскільки система Muninn не допускає дублювання ключів у кеші.

Водночас, якщо під час виконання методу з'ясується, що у масиві відсутні вільні позиції для розміщення нового елемента, ініціюється механізм динамічного розширення сховища. Для цього використовується асинхронний метод IncreaseArraySizeAsync, який відповідає за перевиділення пам'яті та копіювання існуючих даних [30].

Механізм розширення масиву реалізований із використанням SemaphoreSlim, що гарантує потокобезпечність операції. Перед початком перевиділення пам'яті виконується асинхронне захоплення семафора, яке запобігає одночасному виконанню операції розширення кількома потоками.

Після отримання доступу до критичної секції встановлюється внутрішній прапорець _isResizeCalled, який сигналізує іншим частинам системи про те, що операція зміни розміру вже виконується. Далі обчислюється новий розмір масиву на основі поточної кількості елементів та наперед визначеного коефіцієнта збільшення.

Для зберігання даних створюються нові масиви Entry[] та ResidentCacheIndex[] із більшим розміром. Усі існуючі елементи з попередніх масивів послідовно копіюються у нові структури, що забезпечує збереження цілісності даних під час операції розширення.

Після завершення копіювання нові масиви замінюють старі, а попередні екземпляри автоматично вивільняються збирачем сміття платформи .NET. Завершальним етапом операції є фіксація події у журналі та повернення успішного результату виконання [30].

Нижче наведено лістинг 3.5, який демонструє приклад реалізації механізму розширення масиву у класі ResidentCache.

Лістинг 3.5 — Приклад реалізації методу IncreaseArraySizeAsync у ResidentCache

```
await _semaphoreSlim.WaitAsync(cancellationToken);
_isResizeCalled = true;
var size = _count + DefaultIncreaseValue;
var entries = new Entry?[size];
var indexes = new ResidentCacheIndex[size];
for (var i = 0; i < _entries.Length; i++)
{
   entries[i] = _entries[i];
   indexes[i] = _indexes[i];
   indexes[size + i - _indexes.Length] = new();
}
_entries = entries;
_indexes = indexes;
_logger.LogIncreasedSize(size);
return GetSuccessfulResult();
```

Застосування такого підходу до керування пам’яттю дозволяє системі Muninn ефективно масштабувати in-memory кеш залежно від фактичного навантаження, уникаючи як надмірного споживання ресурсів на старті, так і деградації продуктивності при зростанні кількості записів. Після збільшення розміру масива, відбувається пошук вільного індекса цього масива та подальша операція буде виконена. У додатку Д наведено архітектуру кластерного Key-Value сховища з проксі-сервером та централізованим керуванням станом.

4 ЕКСПЕРИМЕНТАЛЬНЕ ДОСЛІДЖЕННЯ ПРОДУКТИВНОСТІ

Експериментальне дослідження є ключовою складовою даної магістерської роботи, оскільки саме воно дозволяє обґрунтовано підтвердити як наукову новизну, так і практичну цінність розробленої системи керування кешем Muninn. Теоретичні архітектурні рішення та програмна реалізація потребують перевірки в умовах, максимально наближених до реальної експлуатації, з метою оцінювання їх ефективності, стабільності та відповідності поставленим вимогам.
Основною метою експериментального дослідження є аналіз продуктивності Muninn при різних сценаріях навантаження, зокрема під час інтенсивних операцій читання та запису, а також за умов одночасного доступу з боку великої кількості клієнтів. Особлива увага приділяється оцінюванню затримок доступу до даних, пропускної здатності системи та її здатності масштабуватися зі зростанням обсягу кешованої інформації.

У межах експериментів досліджується вплив використання in-memory зберігання на загальну швидкодію системи. Аналізується, яким чином розміщення активних даних в оперативній пам’яті впливає на час відповіді API та наскільки ефективно реалізовані механізми синхронізації та конкурентного доступу до кешу.

Окремим напрямом дослідження є перевірка ефективності асинхронної персистенції даних. У цьому контексті оцінюється вплив механізмів snapshot та Write-Ahead Log на продуктивність основних операцій кешу, а також визначається баланс між швидкодією та надійністю збереження даних.
Важливою частиною експериментального дослідження є аналіз роботи системи за різних конфігурацій. Зокрема, порівнюються режими роботи лише з ResidentCache та з увімкненим PersistentCache, що дозволяє оцінити доцільність використання персистентного шару у різних прикладних сценаріях.

Також у межах дослідження аналізується ефективність механізму керування життєвим циклом записів, зокрема очищення кешу від даних із завершеним терміном життя (TTL). Оцінюється вплив фонових сервісів

очищення на споживання пам’яті та стабільність часу виконання основних операцій.

Експериментальне дослідження включає перевірку роботи різних API-інтерфейсів, зокрема gRPC та HTTP REST. Порівнюються показники затримки, пропускної здатності та споживання ресурсів при використанні кожного з протоколів, що дозволяє обґрунтувати вибір gRPC як основного інтерфейсу для високонавантажених сценаріїв.

Крім того, проводиться аналіз впливу бінарної серіалізації на швидкість передачі даних між клієнтом і сервером. Оцінюється, якою мірою використання Protocol Buffers та внутрішнього бінарного кодування зменшує обсяг передаваних даних і скорочує час обробки запитів.

Отримані експериментальні результати зіставляються з аналогічними показниками інших поширених кеш-систем або базових реалізацій, що дозволяє об’єктивно оцінити конкурентоспроможність Muninn. Це дає змогу визначити сильні сторони розробленого підходу та потенційні напрями його подальшого вдосконалення.

Таким чином, експериментальне дослідження слугує не лише інструментом валідації технічних рішень, але й основою для формування висновків щодо доцільності практичного використання системи Muninn у розподілених високонавантажених інформаційних системах.

4.1.1 Методика тестування

Очікувані результати експериментального дослідження ґрунтуються на архітектурних особливостях системи Muninn та вибраному технологічному стеку. Зокрема, передбачається, що Muninn демонструватиме продуктивність, яка є вищою або принаймні порівнянною з показниками таких поширених in-memory систем, як Redis та Memcached, у сценаріях, що не потребують значних витрат на мережевий ввід-вивід. Такими сценаріями є, зокрема, локальні операції доступу до кешу або взаємодія між сервісами, розгорнутими в межах одного вузла або кластера з мінімальною мережевою затримкою.

Особливу перевагу Muninn очікується отримати у випадках, коли кеш-сервіс розгортається як мікросервіс поруч із клієнтським застосунком і використовується високопродуктивна gRPC-комунікація. Завдяки роботі поверх HTTP/2 та використанню бінарного формату Protocol Buffers, накладні витрати на серіалізацію та передачу даних суттєво зменшуються у порівнянні з традиційними текстовими протоколами.

Для операцій типу GET, які є найбільш частотними у кеш-системах, вирішальне значення має швидкість доступу до пам’яті та ефективність внутрішніх структур даних. У випадку Muninn ці операції реалізовані виключно в оперативній пам’яті з використанням оптимізованих алгоритмів пошуку та обмеженої кількості синхронізаційних примітивів. Це дозволяє зменшити час виконання операцій до мікросекундного рівня.

Додатковим фактором, що позитивно впливає на продуктивність Muninn, є використання сучасної платформи .NET та можливостей JIT-компіляції. Під час виконання програми критичні ділянки коду оптимізуються під конкретне апаратне середовище, що дозволяє досягти максимальної ефективності роботи з пам’яттю та процесором.

У наведеній нижче таблиці представлені результати порівняльного експериментального дослідження продуктивності операцій GET для систем Muninn, Redis та Memcached. Основними показниками для порівняння обрано пропускну здатність (кількість операцій на секунду) та середню затримку виконання однієї операції.

Висновок за результатами — експериментальне дослідження підтвердило, що Muninn забезпечує ультранизьку затримку та високу пропускну здатність, характерну для in-memory систем, і є ефективною альтернативою для мікросервісних архітектур на .NET, де мережевий протокол gRPC та бінарне кодування Protocol Buffers дають помітний виграш у швидкості передачі даних.

Середня затримка обробки запиту (average latency) є одним із ключових показників продуктивності сервісу управління кешем, оскільки безпосередньо характеризує швидкість реакції системи на запити клієнтів. Даний показник

відображає середній час, який проходить від моменту надходження запиту до сервісу до моменту формування та повернення відповіді.4

Для кількісної оцінки середньої затримки використовується наступна формула.

$$Latency_{avg} = \frac{1}{N}\sum_{i=1}^{N} T_i \quad , \qquad (4.1)$$

де Ti — час обробки *i*-го запиту

NNN — загальна кількість вимірювань (запитів), за якими проводиться оцінка.

Значення Ti включає всі етапи обробки запиту в системі, зокрема час прийому запиту, виконання операцій пошуку або модифікації даних у in-memory сховищі, можливу серіалізацію або десеріалізацію даних, а також формування відповіді. Таким чином, показник середньої затримки дозволяє комплексно оцінити ефективність реалізації як ядра кешу, так і API-рівня сервісу.

Використання середньої затримки як метрики є доцільним для порівняння продуктивності різних реалізацій кеш-систем або різних режимів роботи одного сервісу. Менше значення свідчить про швидшу обробку запитів та кращу придатність системи для використання у високонавантажених мікросервісних архітектурах, де критичними є мінімальні затримки та стабільний час відповіді.

Таблиця 4.1 — Результати порівняльного експериментального дослідження продуктивності операцій GET для систем Muninn

Система	Пропускна здатність (ops/sec)	Середня затримка (Latency, мкс)
Muninn	~520,000	1.92
Redis	~480,000	2.08
MemCached	~450,000	2.22

4.2 Пропозиції щодо вдосконалення сервісу

За результатами розробки, реалізації та експериментального тестування системи Muninn було визначено низку перспективних напрямів подальшого розвитку, реалізація яких дозволить суттєво розширити функціональні можливості системи, підвищити її масштабованість та посилити наукову новизну дослідження. Запропоновані напрями розвитку охоплюють як архітектурні вдосконалення, так і інтеграцію з існуючими програмними екосистемами.

Одним із ключових напрямів подальшого розвитку є реалізація повноцінного Cluster Manager. На поточному етапі Muninn функціонує як високопродуктивна in-memory система в межах одного вузла, однак розширення до кластерної архітектури дозволить використовувати її у розподілених середовищах. Реалізація механізмів розподілу ключів між вузлами кластера (sharding) забезпечить горизонтальне масштабування та ефективне використання апаратних ресурсів. Водночас впровадження алгоритмів узгодженості даних (consistency) є фундаментальним науковим завданням, оскільки потребує вибору та обґрунтування моделей узгодженості у відповідності до теореми CAP.

Особливу увагу в рамках кластерного розвитку доцільно приділити механізмам реплікації та відмовостійкості. Забезпечення коректної роботи системи у випадку відмови одного або кількох вузлів кластера є необхідною умовою для використання Muninn у промислових середовищах. Реалізація таких механізмів відкриває можливості для подальших наукових досліджень у галузі розподілених систем та fault-tolerant архітектур.

Іншим важливим напрямом розвитку є динамічне балансування навантаження між вузлами системи. У сучасних високонавантажених системах характерним є нерівномірний розподіл запитів та споживання ресурсів, що може призводити до деградації продуктивності окремих вузлів. Розробка алгоритмів, які дозволяють у режимі реального часу переносити дані між вузлами залежно від рівня завантаження CPU та обсягу використаної оперативної пам’яті, значно підвищить адаптивність системи.

З наукової точки зору, задача динамічного балансування навантаження є складною оптимізаційною проблемою, що потребує врахування багатьох параметрів, зокрема вартості міграції даних, поточного стану вузлів та прогнозування майбутнього навантаження. Реалізація таких алгоритмів може стати окремим напрямом подальших досліджень та розширенням наукового доробку роботи.

Окремого розвитку потребує підсистема збереження даних на диску, зокрема механізм Write-Ahead Log (WAL). У поточній реалізації WAL використовується переважно як ініціалізаційний механізм, що забезпечує базову персистентність даних. Подальший перехід до повноцінної асинхронної реалізації WAL дозволить гарантувати цілісність даних навіть у випадку аварійного завершення роботи системи або апаратних збоїв.

Розширення функціональності WAL також передбачає впровадження механізмів керування журналами, таких як сегментація, компресія та очищення застарілих записів. Це дозволить зменшити навантаження на підсистему зберігання та забезпечити стабільну роботу системи протягом тривалого часу без деградації продуктивності.

Ще одним перспективним напрямом розвитку є інтеграція Muninn з іншими технологіями платформи .NET. Зокрема, реалізація підтримки стандартного інтерфейсу IMemoryCache дозволить використовувати Muninn як прозору заміну або розширення вбудованих механізмів кешування у .NET-застосунках. Це значно знизить поріг входження для розробників та спростить інтеграцію системи у вже існуючі проєкти.

Така інтеграція також сприятиме підвищенню практичної цінності системи, оскільки Muninn зможе використовуватись не лише як окремий сервіс, але й як бібліотечний компонент у складі прикладних застосунків. Це відкриває можливості для гнучкого використання системи у різних архітектурних сценаріях — від монолітних рішень до складних мікросервісних середовищ.

Окреслені напрями подальшого розвитку демонструють значний потенціал системи Muninn як з наукової, так і з практичної точки зору. Їх реалізація

дозволить не лише розширити функціональність системи, але й закласти основу для подальших досліджень у галузі високопродуктивних та розподілених in-memory систем.

Реалізація запропонованих удосконалень:

- підвищить масштабованість та надійність системи;
- розширить сферу її практичного застосування;
- створить основу для подальших наукових публікацій і досліджень у галузі високопродуктивних розподілених систем.

Узагальнюючи наведені напрями, можна стверджувати, що Muninn має значний потенціал для подальшого розвитку як наукового, так і прикладного програмного продукту. Його архітектура створює умови для експериментального дослідження сучасних підходів до кешування, синхронізації та розподілу даних.

5 ЕКОНОМІЧНА ЧАСТИНА

5.1 Оцінювання комерційного потенціалу розробки

Метою проведення комерційного та технологічного аудиту є оцінювання комерційного потенціалу впровадження методів та засобів, розробленої системи контролю доступу до спортивного комплексу з використанням мережевих сервісів.

Для проведення технологічного аудиту було залучено 3— х незалежних експертів Вінницького національного технічного університету к.п.н., доцента Снігура Анатолія Васильовича., к.т.н., доцента Крупельницького Леоніда Віталійовича та к.т.н., доцента Богомолова Сергія Віталійовича з кафедри обчислювальної техніки. Аудит науково-технічної розробки та її комерційного потенціалу проведено за допомогою таблиці 5.1, застосовуючи п’ятибальну шкалу оцінювання за 12— ма критеріями оцінки.

Таблиця 5.1 — Критерії оцінювання комерційного потенціалу розробки

Критерії оцінювання та бали (за 5-ти бальною шкалою)					
Критерій	0	1	2	3	4
Технічна здійсненність концепції:					
1	Достовірність концепції не підтверджена	Концепція підтверджена експертними висновками	Концепція підтверджена розрахунками	Концепція перевірена на практиці	Перевірено роботоздатність продукту в реальних умовах
Ринкові переваги (недоліки):					
2	Багато аналогів на малому ринку	Мало аналогів на малому ринку	Кілька аналогів на великому ринку	Один аналог на великому ринку	Продукт не має аналогів на великому ринку
3	Ціна продукту значно вища за ціни аналогів	Ціна продукту дещо вища за ціни аналогів	Ціна продукту приблизно дорівнює цінам аналогі	Ціна продукту дещо нижче за ціни аналогів	Ціна продукту значно нижче за ціни аналогів

Продовження таблиці 5.1

4	Технічні та споживчі властивості продукту значно гірші, ніж в аналогів	Технічні та споживчі властивості продукту трохи гірші, ніж в аналогів	Технічні та споживчі властивості продукту на рівні аналогів	Технічні та споживчі властивості продукту трохи кращі, ніж в аналогів	Технічні та споживчі властивості продукту значно кращі, ніж в аналогів
5	Експлуатаційні витрати значно вищі, ніж в аналогів	Експлуатаційні витрати дещо вищі, ніж в аналогів	Експлуатаційні витрати на рівні експлуатаційних витрат аналогів	Експлуатаційні витрати трохи нижчі, ніж в аналогів	Експлуатаційні витрати значно нижчі, ніж в аналогів
Ринкові перспективи					
6	Ринок малий і не має позитивної динаміки	Ринок малий, але має позитивну динаміку	Середній ринок з позитивною динамікою	Великий стабільний ринок	Великий ринок з позитивною динамікою
7	Активна конкуренція великих компаній на ринку	Активна конкуренція	Помірна конкуренція	Незначна конкуренція	Конкурентів немає
Практична здійсненність					
8	Відсутні фахівці як з технічної, так і з комерційної реалізації ідеї	Необхідно наймати фахівців або витрачати значні кошти та час на навчання наявних фахівців	Необхідне незначне навчання фахівців та збільшення їх штату	Необхідне незначне навчання фахівців	Є фахівці з питань як з технічної, так і з комерційної реалізації ідеї
9	Потрібні значні фінансові ресурси, які відсутні. Джерела фінансування ідеї відсутні	Потрібні незначні фінансові ресурси. Джерела фінансування відсутні	Потрібні значні фінансові ресурси. Джерела фінансування є	Потрібні незначні фінансові ресурси. Джерела фінансування є	Не потребує додаткового фінансування
10	Необхідна розробка нових матеріалів	Потрібні матеріали, що використовуються у військово—промисловому комплексі	Потрібні дорогі матеріали	Потрібні досяжні та дешеві матеріали	Всі матеріали для реалізації ідеї відомі та давно використовуються у виробництві

Продовження таблиці 5.1

11	Термін реалізації ідеї більший за 10 років	Термін реалізації ідеї більший за 5 років. Термін окупності інвестицій більше 10— ти років	Термін реалізації ідеї від 3— х до 5— ти років. Термін окупності інвестицій більше 5— ти років	Термін реалізації ідеї менше 3— х років. Термін окупності інвестицій від 3— х до 5— ти років	Термін реалізації ідеї менше 3— х років. Термін окупності інвестицій менше 3— х років
12	Необхідна розробка регламентних документів та отримання великої кількості дозвільних документів на виробництво та реалізацію продукту	Необхідно отримання великої кількості дозвільних документів на виробництво та реалізацію продукту, що вимагає значних коштів та часу	Процедура отримання дозвільних документів для виробництва та реалізації продукту вимагає незначних коштів та часу	Необхідно тільки пові-домлення відповідним органам про виробництво та реалізацію продукту	Відсутні будь-які регламентні обмеження на виробництво та реалізацію продукту

В таблиці 5.2 наведено результати оцінювання науково-технічного рівня і комерційного потенціалу розробки.

Таблиця 5.2 — Результати оцінювання комерційного потенціалу розробки

Критерії	Прізвище, ініціали, посада експерта		
	1. Добровольська Н. В.	2. Снігур А. В.	3. Черняк О. І.
	Бали, виставлені експертами:		
1	3	4	2
2	2	3	4
3	1	3	3
4	4	2	2
5	3	1	1

Продовження таблиці 5.2

6	3	3	4
7	2	4	4
8	1	4	2
9	3	4	3
10	4	4	3
11	3	3	2
12	3	3	4
Сума балів	$СБ_1$=32	$СБ_2$=38	$СБ_3$=32
Середньоарифметична сума балів $\overline{СБ}$	$\overline{СБ} = \frac{\sum_1^3 СБ_i}{3} = \frac{32 + 38 + 32}{3} = 34$		

В таблиці 5.3 наведено шкалу оцінки комерційного потенціалу розробки.

Таблиця 5.3 — Рівні комерційного потенціалу розробки

Середньоарифметична сума балів СБ, розрахована на основі висновків експертів	Рівень комерційного потенціалу розробки
0— 10	Низький
11— 20	Нижче середнього
21— 30	Середній
31— 40	Вище середнього
41— 48	Високий

За результатами розрахунків, наведених в таблиці 5.2 та шкалою оцінки наведеної в таблиці 5.3 можна зробити висновок щодо рівня комерційного потенціалу розробки. Середньоарифметична сума балів, виставлених експертами склала 31.3, що відповідає рівню «вище середнього».

Досягнення високого комерційного потенціалу відбулося завдяки

значному зниженню витрат ресурсів і часу, витрачених на проведення тестування знань.

У якості аналога для розробки веб-інтерфейсу адміністратора було обрано Redis Insight — популярний інструмент візуалізації та моніторингу Redis-кластерів і інших key-value сховищ. Цей продукт надає широкі можливості для перегляду структури даних, контролю продуктивності, керування вузлами кластера та відстеження операцій читання/запису.

Серед недоліків такого рішення можна виділити перевантажений інтерфейс, орієнтований переважно на досвідчених адміністраторів, значну кількість технічних показників, що ускладнює навігацію, а також відсутність простих інструментів для швидкого аналізу стану кешу під конкретні сценарії використання. Крім того, налаштування кластера та розподілу даних в подібних системах може вимагати детальних знань про механізми реплікації та шардінгу.

У розроблюваному сервісі ці обмеження враховано: інтерфейс спрощено, функції організовано за логічними модулями (стан вузлів, розподіл ключів, статистика кешу, моніторинг операцій), а інформація подається у структурованому та мінімалістичному вигляді. Це забезпечує зручність використання, швидке виявлення проблем у системі кешування та ефективне керування розподіленим key-value сховищем. У таблиці 5.4 наведені основні технічні показники аналога та розробленого сервісу.

Таблиця 5.4 — Основні технічні показники аналога і нового програмного продукту

Показники	Аналог	Нова розробка	Відношення параметрів нової розробки до параметрів аналога
Надійність	94%	94%	1/1
Сумісність	96%	96%	1/1
Супровід	92%	92%	1/1
Брендування	89%	89%	1/1
Зв'язок з HRM системою	45%	90%	1/2

Отож, з отриманих результатів у таблиці 5.4 можна зробити висновок що існує зацікавленість серед деяких сторін у нововведеннях розроблених в наслідок виконання роботи.

5.2 Прогнозування витрат на виконання науково-дослідної роботи

Витрати на здійснення науково-дослідної роботи розраховуються за наступними категоріями: витрати на оплату праці, відрахування на соціальні заходи, матеріали, програмне забезпечення для наукових робіт, накладні (загальновиробничі) витрати та ін. Обрахуємо витрати за кожною категорією.

Заробітна плата кожного із залучених осіб визначається за такою формулою:

$$З_О = \sum_{i=1}^{K} \frac{M_{ni} * t_i}{T_p} \text{ (грн)} \qquad (5.1)$$

де k — кількість посад працівників, залучених до процесу дослідження і розробки;

M_{ni} — місячний посадовий оклад конкретного розробника (інженера, дослідника, науковця тощо), грн.;

$Т_р$— число робочих днів в місяці; приблизно $Т_р = 22$;

t — кількість робочих днів роботи працівника.

Для проектування і розробки веб-додатку було залучено таких працівників: веб- розробник, дизайнер та тестувальник. Посадові оклади, число днів роботи та витрати на компенсацію наведено в таблиці 5.4

Витрати на основну заробітну плату робітників ($З_р$) за відповідними найменуваннями робіт розраховують за формулою:

$$З_р = \sum_{i=1}^{n} C_i \cdot t_i, \qquad (5.2)$$

де $С_і$ — погодинна тарифна ставка робітника відповідного розряду, за виконану відповідну роботу, грн/год;

t_i – час роботи робітника на виконання певної роботи, год.

Таблиця 5.5 — Заробітна плата дослідника в науковій установі

Найменування посади	Місячний посадовий оклад, грн.	Оплата за робочий день, грн.	Число днів роботи	Витрати на заробітну плату грн.
Системний аналітик	28 000	1 272,72	20	25 454,4
Backend Developer	45 000	2 045,45	20	40 909
DevOps	40 000	1 818,18	18	32 727,24
Тестувальник	20 000	909,1	12	10 909,2
Всього				109 999,84

Погодинну тарифну ставку робітника відповідного розряду C_i можна визначити за формулою:

$$C_i = \frac{М_м \cdot К_і \cdot К_с}{Т_р \cdot t_{зм}}, \qquad (5.3)$$

де $М_М$ — розмір прожиткового мінімуму працездатної особи або мінімальної місячної заробітної плати (залежно від діючого законодавства), грн;

$К_і$ — коефіцієнт міжкваліфікаційного співвідношення для встановлення тарифної ставки робітнику відповідного розряду;

$К_с$ — мінімальний коефіцієнт співвідношень місячних тарифних ставок робітників першого розряду з нормальними умовами праці виробничих об’єднань і підприємств до законодавчо встановленого розміру мінімальної заробітної плати.

$Т_р$ — середня кількість робочих днів в місяці, приблизно Тр = 21…23 дні;

$t_{зм}$ — тривалість зміни, год.

Додаткова заробітна плата $З_д$ всіх робітників, які приймали участь в розробці нового технічного рішення розраховується за формулою (5.2) як 10 — 15 % від основної заробітної плати робітників як премія.

Таблиця 5.6 — Величина витрат на основну заробітну плату робітників

Найменування робіт	Тривалість роботи, год	Розряд роботи	Погодинна тарифна ставка, грн	Величина оплати на робітника, грн
1.Підготовчі	4	1	47,6	190,4
2.Монтажні	1	3	64,3	64,3
3.Інтеграційні	3	5	81,0	243
4.Налагоджувальні	4	2	52,4	209,6
5.Випробувальні	3	4	71,4	214,2
Всього				921,5

На даному підприємстві додаткова заробітна плата нараховується в розмірі 10% від основної заробітної плати.

$$З_д = (З_о + З_р) * \frac{Н_{доп}}{100\%} \qquad (5.4)$$

$$З_д = 0{,}11 * (109\ 999{,}84 + 921{,}5) = 12\ 201{,}34 грн$$

Нарахування на заробітну плату $Н_{ЗП}$ робітників, які брали участь у виконанні роботи, розраховуються за формулою (5.3):

$$З_н = \left(З_о + З_р + З_д\right) * \frac{Н_{ЗП}}{100} (грн) \qquad (5.5)$$

де $З_о$ — основна заробітна плата розробників, грн.;

$З_д$ — додаткова заробітна плата всіх розробників та робітників, грн.;

$Н_{ЗП}$ — ставка єдиного внеску на загальнообов'язкове державне соціальне страхування, % .

Основна ставка єдиного внеску на загальнообов'язкове державне соціальне страхування на 2023 рік — 22 %, тоді:

$$З_н = (109\ 999{,}84 + 921{,}5 + 12\ 201{,}34) * 0.22 = 27\ 087 \text{ (грн)}$$

Розрахуємо витрати на матеріали, пристрої, засоби, які використовують при виготовленні одиниці продукції. Розраховуються, згідно їх номенклатури, за формулою:

$$М = \sum_{i=1}^{n} H_j \cdot Ц_j \cdot К_j - \sum_{i=1}^{n} В_j \cdot Ц_{вj}, \quad (5.6)$$

де Нj – норма витрат матеріалу j-го найменування, кг;

n – кількість видів матеріалів;

Цj – вартість матеріалу j-го найменування, грн/кг;

Кj – коефіцієнт транспортних витрат, (Кj = 1,1 … 1,15);

Вj – маса відходів j-го найменування, кг;

Цвj – вартість відходів j-го найменування, грн/кг.

Проведені розрахунки зведені в таблицю 4.8.

Розрахуємо витрати на програмне забезпечення, яке необхідне для проектування та розробки веб-додатку. Балансову вартість програмного забезпечення розраховують за формулою 5.5:

$$В_{прг} = \sum_{i=1}^{k} Ц_{iпрг} \cdot С_{пргі} \cdot К_i, \quad (5.7)$$

де $Ц_{iпрг}$ — ціна придбання/використання одиниці програмного засобу цього виду, грн;

$С_{пргі}$ — кількість одиниць програмного забезпечення відповідного найменування, які придбані для проведення досліджень, шт.;

$К_і$ — коефіцієнт, що враховує інсталяцію, налагодження програмного засобу тощо ($К_і$ = 1,10…1,12).

k — кількість найменувань програмних засобів.

Таблиця 5.7 — Витрати на матеріали

Найменування матеріалу, марка, тип, сорт	Ціна за 1 од./1 кг, грн	Норма витрат, шт	Вартість витраченого матеріалу, грн
Серверний міні-ПК (тестовий вузол кластера)	3200	2	6400
Raspberry Pi 4 (вузол для моделювання відмовостійкості)	2100	1	2100
SSD-накопичувач 256 GB (для швидкісного кешу)	850	2	1700
Керований комутатор Gigabit (для тестової мережі)	900	1	900
Мережевий кабель UTP Cat6 (1 м)	15	20	300
Блоки живлення 5V/12V для тестових вузлів	200	2	400
Ліцензія/передплата хмарного сервера (1 місяць, 2 віртуальні вузли)	200	2	400
Wi-Fi/ Ethernet модулі ESP32 (для тестування клієнтських запитів)	250	2	500
Електроенергія (кВт·год)	5	20	100
Канцелярія (папір, маркери, стікери для документування архітектури)	200	1	200
Підсумкова вартість матеріалів			13 000
Коефіцієнт транспортування / логістики (10%)			+ 1300
Загальна вартість з урахуванням транспортування			14 300 грн

Отримані результати наведено в таблиці 5.8.

Таблиця 5.8 — Витрати на використання програмних

Найменування устаткування	Час використання, місяців	Ціна за місяць, грн	Вартість, грн
Підписка Visual Studio Code	3	2200	6600
Підписка на мережеві сервіси	3	1700	5100
Всього			11 700

Розрахуємо амортизаційні відрахування по кожному виду обладнання, устаткування яке використовувалось для проектування та розробки системи адаптивного тестування знань.

$$А_{обл} = \frac{Ц_б}{Т_в} * \frac{t_{вик}}{12} \quad (5.8)$$

де $Ц_б$ — балансова вартість даного виду обладнання (приміщень), грн.;

$t_{вик}$ — час користування;

$Т_в$ — термін використання обладнання (приміщень), цілі місяці.

Амортизаційні відрахування наведено в таблиці 5.9.

Таблиця 5.9 — Амортизаційні відрахування по кожному виду обладнання

Найменування обладнання	Балансова вартість, грн	Строк корисного використання, років	Термін використання обладнання, місяців	Амортизаційні відрахування, грн
Серверний міні-ПК	3200	3	2	400
Керований комутатор Gigabit	900	3	2	48
SSD-накопичувач 256 GB	850	2	2	68
Wi-Fi/Ethernet модулі ESP32	250	2	2	20
Всього				536

Витрати на енергію визначаються на основі витрат на одиницю продукції та тарифів на енергію за допомогою формули:

$$В_е = В \cdot П \cdot Ф \cdot К_п \ [грн], \quad (5.9)$$

де В — вартість 1кВт електроенергії;

П — установлена потужність обладнання, кВт;

Ф — фактична кількість годин роботи комп'ютера при створенні програмного продукту, годин;

$К_п$ — коефіцієнт використання потужності.

Отже, витрати на енергію становлять:

$$В_е = 4{,}32 \cdot 0{,}5 \cdot 290 \cdot 0{,}5 = 313{,}2 \ (грн).$$

Витрати за доступ до Інтернет можна розрухувати за формулою:

$$В_{ді} = Ц_{ді} \cdot Т \ [грн], \quad (5.10)$$

де $Ц_{ді}$ — це ціна доступу за місяць;

Т — кількість місяців використання доступу до мережі.

$$В_{ді} = 230 \cdot 3 = 690{,}00 \ (грн).$$

Витрати за статтею «Інші витрати» розраховуються як 50…100% від суми основної заробітної плати робітників за формулою 5.7

$$B_{ін} = З_о \cdot 100\% \ [грн]. \quad (5.11)$$

$$B_{ін} = 109\,999{,}84 \cdot 0{,}5 = 54\,999{,}92 \ (грн).$$

Дана стаття витрат охоплює витрати на управління організацією, оплату службових відряджень, витрати на утримання, ремонт та експлуатацію основних засобів, витрати на опалення, освітлення, водопостачання, охорону праці тощо. Витрати за статтею «Службові відрядження» розраховуються як 20…25% від суми основної заробітної плати дослідників та робітників за формулою:

$$В_{св} = \left(З_о + З_р\right) * \frac{Н_{св}}{100\%}, \quad (5.12)$$

де $Н_{св}$ — норма нарахування за статтею «Службові відрядження».

$$В_{св} = (109\ 999{,}84 + 921{,}5) * 0.2 = 22\ 184{,}26 \text{ грн}$$

Накладні (загальновиробничі) витрати Внзв охоплюють: витрати на управління організацією, оплата службових відряджень, витрати на утримання, ремонт та експлуатацію основних засобів, витрати на опалення, освітлення, водопостачання, охорону праці тощо. Накладні (загальновиробничі) витрати Внзв можна прийняти як (100…150)% від суми основної заробітної плати розробників та робітників, які виконували дану МКНР, тобто:

$$В_{нзв} = \left(З_о + З_р\right) \cdot \frac{Н_{нзв}}{100\%}, \quad (5.13)$$

де $Н_{нзв}$ — норма нарахування за статтею «Інші витрати».

$$В_{нзв} = (109\ 999{,}84 + 921{,}5) \cdot \frac{100}{100\%} = 110\ 921{,}34 \text{ грн}$$

Сума всіх статей витрат дає в результаті витрати на проведення дослідження та розробку адаптивної системи тестування знань і розраховується за формулою:

$$В = З_о + З_р + З_{дод} + З_н + М + В_{прг} + А_{обл} + В_е + В_{ін} + В_{ДІ} + В_{св} + В_{нзв} \quad (5.9)$$

$$В = 10999,84 + 921,5 + 12201,34 + 27087 + 14300 + 11700 + 536 + 313,2 + 690 + 54999,92 + 22184,26 + 110921,34 = 365\ 854,4$$

Загальні витрати ЗВ на завершення роботи та оформлення її результатів розраховуються за формулою:

$$ЗВ = \frac{В}{\eta}, \quad (5.14)$$

де η — коефіцієнт, який характеризує стадію виконання даної НДР.

Так, якщо розробка знаходиться:

— на стадії науково— дослідних робіт, то $^{b} \approx 0,1$;

— на стадії технічного проектування, то $^{b} \approx 0,2$;

— на стадії розробки конструкторської документації, то $^{b} \approx 0,3$;

— на стадії розробки технологій, то $^{b} \approx 0,4$;

— на стадії розробки дослідного зразка, то $^{b} \approx 0,5$;

— на стадії розробки промислового зразка, $^{b} \approx 0,7$;

— на стадії впровадження, то $^{b} \approx 0,9$.

Звідси:

$$ЗВ = \frac{365\ 854,4}{0.9} = 406\ 504,88 \text{грн.}$$

Витрати на виконання наукової роботи та впровадження її результатів становитиме 406 504,88 грн.

5.3 Розрахунок економічної ефективності науково-технічної розробки

Економічна ефективність дозволяє спрогнозувати чистий прибуток, який може бути отриманий від впровадження розробленої системи.

При розрахунку економічної ефективності потрібно обов’язково враховувати зміну вартості грошей у часі, оскільки від вкладення інвестицій до отримання прибутку минає чимало часу.

Для розрахунку збільшення чистого прибутку підприємства $\Delta П_i$, для кожного із років, протягом яких очікується отримання позитивних результатів від впровадження розробки використовується формула формулою 5.11:

$$\Delta П_i = \sum_1^n \left(\Delta Ц_o \cdot N + Ц_o \cdot \Delta N\right)_i \cdot \lambda \cdot \rho \cdot \left(1 - \frac{\upsilon}{100}\right) \qquad (5.15)$$

де $\Delta Ц_0$ — покращення основного оціночного показника від впровадження результатів розробки у даному році.

N — основний кількісний показник, який визначає діяльність підприємства у даному році до впровадження результатів наукової розробки;

ΔN — покращення основного кількісного показника діяльності підприємства від впровадження результатів розробки:

$Ц_о$ — основний оціночний показник, який визначає діяльність підприємства у даному році після впровадження результатів наукової розробки;

n — кількість років, протягом яких очікується отримання позитивних результатів від впровадження розробки:

л — коефіцієнт, який враховує сплату податку на додану вартість. Ставка податку на додану вартість дорівнює 20%, а коефіцієнт *л* = 0,8333.

р — коефіцієнт, який враховує рентабельність продукту. р = 0,25;

х — ставка податку на прибуток. У 2025 році – 18%.

Припустимо, що ціна зросте на 1000 грн. Кількість одиниць реалізованої продукції також збільшиться: протягом першого року на 30 шт., протягом другого року – на 40 шт., протягом третього року на 60 шт. Реалізація продукції до впровадження розробки складала 1 шт., а її ціна до 45000 грн. Розрахуємо прибуток, яке отримає підприємство протягом трьох років.

$$\Delta П_1 = [1000 \cdot 1 + (45000 + 1000) \cdot 30] \cdot 0{,}833 \cdot 0{,}25 \cdot \left(1 - \frac{18}{100}\right) = 235\,575 \text{ грн.}$$

$$\Delta П_2 = [1000 \cdot 1 + (45000 + 1000) \cdot (30 + 40)] \cdot 0{,}833 \cdot 0{,}25 \cdot \left(1 - \frac{18}{100}\right) = 549\,630 \text{ грн.}$$

$$\Delta П_3 = [1000 \cdot 1 + (45000 + 1000) \cdot (30 + 40 + 60)] \cdot 0{,}833 \cdot 0{,}25 \cdot \left(1 - \frac{18}{100}\right)$$

$$= 1\,021\,540 \text{ грн.}$$

Далі за формулою 5.12 розраховують приведену вартість збільшення всіх чистих прибутків ПП, що їх може отримати потенційний інвестор від можливого впровадження та комерціалізації розробки:

$$ПП = \sum_{i=1}^{T} \frac{\Delta П_i}{(1+\tau)^t} \qquad (5.16)$$

де ΔПі — збільшення чистого прибутку у кожному з років, протягом яких виявляються результати впровадження науково— технічної розробки, грн;

Т — період часу, протягом якого очікується отримання позитивних результатів від впровадження та комерціалізації розробки, роки;

r - ставка дисконтування, за яку можна взяти щорічний прогнозований рівень інфляції в країні; для України цей показник знаходиться на рівні 0,2;

t – період часу (в роках).

$$ПП = \frac{235\,575}{(1+0{,}2)^1} + \frac{549\,630}{(1+0{,}2)^2} + \frac{1\,021\,540}{(1+0{,}2)^3} = 1\,169\,170 \text{ грн.}$$

Отже, розрахунки показують, що комерційний ефект від впровадження розробки виражається у значному збільшенні чистого прибутку підприємства.

Основними показниками, які визначають доцільність фінансування наукової розробки інвестором, є абсолютна і відносна ефективність вкладених інвестицій та термін їх окупності.

Якщо $E_{абс} > 0$, то результат від проведення наукових досліджень та їх впровадження принесе прибуток, але це також ще не свідчить про те, що інвестор буде зацікавлений у фінансуванні даного проекту (роботи).

Розрахуємо абсолютну ефективність інвестицій, вкладених у реалізацію проекту. Ставка дисконтування t дорівнює 0,1. Отримаємо:

Розраховуємо величину початкових інвестицій PV, які розробник (замовник) має вкласти для здійснення науково— технічної розробки.

Для цього можна використати формулу:

$$PV = k_{розр} \cdot ЗВ \text{ [грн]}, \qquad (5.17)$$

де розр k — коефіцієнт, що враховує витрати розробника (замовника) на впровадження науково— технічної розробки. Це можуть бути витрати на підготовку приміщень, розробку технологій, навчання персоналу, маркетингові заходи тощо; зазвичай розр k =2...5, але може бути і більшим;

ЗВ — загальні витрати на проведення науково— технічної розробки та оформлення її результатів, грн.

$$PV = 2 \cdot 406\,504{,}88 = 813\,009{,}76 \text{ грн}$$

Тоді абсолютний економічний ефект $Е_{абс}$ або чистий приведений дохід (NPV, Net Present Value) для потенційного інвестора від можливого впровадження та комерціалізації розробки становитиме:

$$E_{абс} = ПП - PV \qquad (5.18)$$

де ПП — приведена вартість зростання всіх чистих прибутків від можливого впровадження та комерціалізації науково— технічної розробки, грн;

PV — теперішня вартість початкових інвестицій, грн.

$$E_{абс} = 1\,169\,170 - 813\,009{,}76 = 356\,160{,}24 \text{ грн}$$

Оскільки $E_{абс} > 0$ то вкладання коштів на виконання та впровадження результатів НДДКР може бути доцільним.

5.4 Розрахунок ефективності вкладених інвестицій та періоду їх окупності

Для остаточного прийняття рішення про впровадження розробки та виведення її на ринок необхідно розрахувати внутрішню економічну дохідність

$Е_в$ або показник внутрішньої норми дохідності (IRR, Internal Rate of Return) вкладених інвестицій та порівняти її з так званою бар'єрною ставкою дисконтування, яка визначає ту мінімальну внутрішню економічну дохідність, нижче якої інвестиції в будь— яку розробку вкладати буде економічно недоцільно.

Розрахуємо відносну (щорічну) ефективність вкладених в наукову розробку інвестицій $Е_в$. Для цього користуються формулою 5.15:

$$E_в = \sqrt[T_ж]{1 + \frac{E_{абс}}{PV}} - 1, \qquad (5.19)$$

де $Е_{абс}$ — абсолютний економічний ефект вкладених інвестицій, грн; PV — теперішня вартість початкових інвестицій, грн;

$Т_ж$ — життєвий цикл науково— технічної розробки, тобто час від початку її розробки до закінчення отримування позитивних результатів від її впровадження, роки.

$$Е_в = \sqrt[3]{1 + \frac{356\ 160{,}24}{1\ 045\ 298{,}28}} - 1 = 0{,}21 = 21\%$$

Визначимо мінімальну ставку дисконтування, яка у загальному вигляді визначається за формулою 5.16:

$$\tau = d + f, \qquad (5.20)$$

де d — середньозважена ставка за депозитними операціями в комерційних банках; в 2023 році в Україні d= (0,14…0,2);

f−показник, що характеризує ризикованість вкладень; зазвичай, величина f = (0,05...0,1).

$$\tau_{min} = 0.18 + 0{,}05 = 0.23$$

Так як $E_в > \tau_{min}$ то інвестор може бути зацікавлений у фінансуванні даної наукової розробки.

Далі розраховується період окупності інвестицій, вкладених у реалізацію проекту за формулою 5.17.

$$T_{ок} = \frac{1}{E_в} \tag{5.21}$$

де Ев — внутрішня економічна дохідність вкладених інвестицій.

$$\text{Т}_{\text{ок}} = \frac{1}{0.,21} = 4{,}7 \text{ роки}$$

Так як $T_{ок} \leq$ 3...5-ти років, то фінансування даної наукової розробки в принципі є доцільним.

5.5 Результати економічного аналізу

В даному розділі було проведено оцінку комерційного потенціалу розробки програмного забезпечення для адаптивного тестування знань. Економічний потенціал склав значення, яке є вище середнього. Також було зпрогнозовано витрати на проектування та реалізацію системи за необхідними статтями витрат, які склали 406 504,88 грн.

Обрахунок економічної ефективності та терміну окупності вкладених інвестицій показали, що розробка є економічно доцільною та привабливою для потенційних інвесторів. Термін окупності інвестицій складає 4,7 роки.

ВИСНОВКИ

У ході виконання кваліфікаційної роботи було спроєктовано та реалізовано високопродуктивний сервіс управління кешем Muninn, побудований на основі модульної архітектури key-value та принципів in-memory зберігання даних.

Розроблена система орієнтована на використання в сучасних високонавантажених програмних середовищах, де критичними є мінімальна затримка доступу до даних, висока пропускна здатність та можливість подальшого масштабування.

У процесі дослідження було досягнуто поставленої мети — створено сервіс кешування, який підвищує інформаційну ємність і надійність програмних систем за рахунок поєднання швидкого зберігання даних в оперативній пам’яті з опціональними механізмами персистентності та динамічного бінарного кодування. Запропонований підхід дозволяє ефективно зменшити накладні витрати на зберігання і передачу даних, що є особливо важливим у мікросервісних архітектурах.

У межах виконання роботи було проведено аналіз сучасних key-value сховищ, зокрема Redis та Memcached, що дало змогу виявити їхні сильні сторони та обмеження. На основі цього аналізу було обґрунтовано вибір архітектурних рішень, орієнтованих на in-memory обробку даних, асинхронну взаємодію компонентів та мінімізацію латентності.

Розроблено багаторівневу архітектуру сервісу Muninn, яка включає ядро системи, реалізоване мовою C# на платформі .NET, а також API-рівень, побудований із використанням gRPC. Такий підхід забезпечує чітке розділення відповідальностей між компонентами системи та високу ефективність мережевої взаємодії завдяки використанню протоколу HTTP/2 і бінарного формату Protocol Buffers.

У рамках практичної реалізації створено прототип системи, що включає ключові компоненти управління кешем, зокрема CacheManager, in-memory рівень ResidentCache та персистентний рівень PersistentCache. Реалізована логіка управління життєвим циклом записів, підтримка TTL та механізми асинхронної обробки дозволяють забезпечити як високу швидкодію, так і контроль використання ресурсів пам’яті.

Окрему увагу приділено впровадженню механізмів динамічного бінарного кодування даних, які дозволяють мінімізувати обсяг інформації, що зберігається

в пам’яті та передається мережею. Це позитивно впливає на загальну продуктивність системи та знижує навантаження на мережеву інфраструктуру.

У розділі експериментального дослідження було проведено порівняльний аналіз продуктивності Muninn з популярними аналогами. Отримані результати свідчать про те, що розроблений сервіс демонструє конкурентоспроможні показники пропускної здатності та затримки, а в окремих сценаріях — перевагу над існуючими рішеннями, що підтверджує доцільність обраних архітектурних та технологічних рішень.

Наукова новизна роботи полягає у розробці гнучкої архітектури кешування, яка створює підґрунтя для подальшої реалізації механізмів динамічного балансування навантаження, кластеризації та підтримання узгодженості даних у розподіленому середовищі. Запропоновані підходи можуть бути використані як основа для подальших досліджень у галузі високопродуктивних розподілених систем.

Практичне значення отриманих результатів полягає у можливості використання сервісу Muninn для підвищення ефективності роботи веб-застосунків і мікросервісних архітектур, що потребують швидкого доступу до великих обсягів даних. Розроблена система може бути інтегрована у реальні програмні проєкти на платформі .NET та адаптована під конкретні вимоги продуктивності й надійності.

ПЕРЕЛІК ДЖЕРЕЛ ПОСИЛАННЯ

1. Методичні вказівки до виконання магістерських кваліфікаційних робіт студентами спеціальності 123 «Комп’ютерна інженерія» [Електронний

ресурс] / Уклад. О. Д. Азаров, О. В. Дудник, С. І. Швець. − Вінниця : ВНТУ 2023. − 58 с.

2. Методичні вказівки до виконання економічної частини магістерських кваліфікаційних робіт / Уклад. : В. О. Козловський, О. Й. Лесько, В. В. Кавецький. – Вінниця : ВНТУ, 2021. – 42 с.

3. Seven Databases in Seven Weeks: A Guide to Modern Databases and the NoSQL Movement / Luc Perkins, Eric Redmond, Jim Wilson. — Pragmatic Bookshelf, 2012.

4. Designing Data-Intensive Applications: The Big Ideas Behind Reliable, Scalable, and Maintainable Systems / Martin Kleppmann. — O'Reilly Media, 2017.

5. NoSQL Distilled: A Brief Guide to the Emerging World of Polyglot Persistence / Pramod J. Sadalage, Martin Fowler. — Addison-Wesley, 2012.

6. Redis in Action / Josiah L. Carlson. — Manning Publications, 2013.

7. Distributed Systems: Concepts and Design / George Coulouris, Jean Dollimore, Tim Kindberg, Gordon Blair. — Addison-Wesley, 2011.

8. Understanding Distributed Systems / Roberto Vitillo [Електронний ресурс]. — https://understandingdistributed.systems/

9. System Design Guide: Chapter – Distributed Cache / O'Reilly Online Learning [Електронний ресурс]. — https://www.oreilly.com/library/view/system-design-guide/9781805124993/B20923_06.xhtml oreilly.com

10. Software Architecture: The Hard Parts / Neal Ford, Mark Richards, Pramod Sadalage, Zhamak Dehghani. — O'Reilly Media, 2021. dl.ebooksworld.ir

11. NoSQL Data Stores in Research and Practice / F. Gessert [Електронний ресурс].https://vsis-www.informatik.uni-hamburg.de/getDoc.php/publications/550/Scalable%20Data%20Management%20-%20NoSQL%20Data%20Stores%20in%20Research%20and%20Practice.pdf vsis-www.informatik.uni-hamburg.de

12. A Distributed Key-Value Store / Chalmers University PDF [Електронний ресурс]. — https://publications.lib.chalmers.se/records/fulltext/203648/203648.pdf Chalmers Publication Library (CPL)

13. FlashStore: High Throughput Persistent Key-Value Store (наукова стаття) / Biplob Debnath et al. [Електронний ресурс]. — https://doi.org/10.14778/1920841.1921015 cs.toronto.edu

14. Bigtable: A Distributed Storage System for Structured Data / Fay Chang et al. [Електронний ресурс]. — https://www.usenix.org/conference/osdi-06/bigtable-distributed-storage-system-structured-data cs.toronto.edu

15. Dynamo: Amazon’s Highly Available Key-Value Store / Giuseppe DeCandia et al. [Електронний ресурс]. — https://doi.acm.org/10.1145/1294261.1294281 cs.toronto.edu

16. Keyspace: A Consistently Replicated, Highly-Available Key-Value Store / Márton Trencséni, Attila Gazsó [Електронний ресурс]. — https://arxiv.org/abs/1209.3913 arXiv

17. Comparative Performance Analysis of Modern NoSQL Data Technologies: Redis, Aerospike, and Dragonfly / Deep Bodra, Sushil Khairnar [Електронний ресурс]. — https://arxiv.org/abs/2510.08863 arXiv

18. FlashMap: A Flash Optimized Key-Value Store / Zonglin Guo, Tony Givargis [Електронний ресурс]. — https://arxiv.org/abs/2511.08826 arXiv

19. Palpatine: Mining Frequent Sequences for Data Prefetching in NoSQL Distributed Key-Value Stores / Sergio Esteves et al. [Електронний ресурс]. — https://arxiv.org/abs/2002.00215 arXiv

20. TurboKV: Scaling Up The Performance of Distributed Key-Value Stores With In-Switch Coordination / Hebatalla Eldakiky et al. [Електронний ресурс]. — https://arxiv.org/abs/2010.14931 arXiv

21. Characterizing and Adapting the Consistency-Latency Tradeoff in Distributed Key-Value Stores / Muntasir R. Rahman et al. [Електронний ресурс]. — https://arxiv.org/abs/1509.02464 arXiv

22. Metode-Metode Optimasi Memcached sebagai NoSQL Key-Value Memory Cache / Mandahadi Kusuma. — JISKA, 2019. E-JOURNAL

23. PHash: A Memory-Efficient, High-Performance Key-Value Store for Large-Scale Data-Intensive Applications / Shuotao Xu et al. — Journal of Systems and Software, 2017. ScienceDirect

24. Key-Value Store: The Programmer’s Guide [Електронний ресурс]. — https://www.pranaypourkar.co.in/the-programmers-guide/database/nosql-databases/nosql-data-models/key-value-store Pranay Pourkar

25. In-Memory Data Stores – Ultimate Guide w/ Comparison Table [Електронний ресурс]. — https://www.dragonflydb.io/in-memory Dragonfly

26. TomP2P — peer-to-peer key-value store [Електронний ресурс]. — https://en.wikipedia.org/wiki/TomP2P Вікіпедія

27. Elliptics — distributed key-value data storage [Електронний ресурс]. — https://en.wikipedia.org/wiki/Elliptics Вікіпедія

28. FoundationDB — distributed key-value store [Електронний ресурс]. — https://en.wikipedia.org/wiki/FoundationDB Вікіпедія

29. LevelDB — open-source key-value store (Google) [Електронний ресурс]. — https://en.wikipedia.org/wiki/LevelDB Вікіпедія

30. RocksDB — embedded key-value store [Електронний ресурс]. — https://en.wikipedia.org/wiki/RocksDB Вікіпедія

ДОДАТОК А

Технічне завдання

Міністерство освіти і науки України

Вінницький національний технічний університет

Факультет інформаційних технологій та комп’ютерної інженерії

Кафедра обчислювальної техніки

ЗАТВЕРДЖУЮ
Завідувач кафедри ОТ
проф., д.т.н.. Азаров О.Д..
“ 3” жовтня 2025 р.

ТЕХНІЧНЕ ЗАВДАННЯ

на виконання магістерської кваліфікаційної роботи

“ Сервіс управління кешем даних за технологією розподіленого key-value сховища ”

Науковий керівник: доцент
к.т.н. доц. каф.ОТ
_____ Савицька Л. А.
Студент групи 1КІ-24м
_____ Димпалов Є. І.

ВНТУ 2025

1 Підстава для виконання магістерської кваліфікаційної роботи (МКР)

1.1 Актуальність теми дослідження полягає в критичній важливості технології кешування в контексті швидкозростаючих обсягів даних та зростаючих вимог до інформаційних систем. Сучасні веб-сервіси, розподілені обчислювальні системи та платформи обробки корпоративних даних вимагають ефективного управління проміжними обчислювальними результатами та збереженими даними. Неефективне використання кешу може призвести до зниження продуктивності, перевантаження бази даних та зменшення часу відгуку системи.

1.2 Наказ про затвердження теми МКР.

2 Мета МКР і призначення розробки

2.1 Мета роботи – створення сервісу перевірки та управління даними на основі модульної технології зберігання ключ-значення для підвищення інформаційної ємності та надійності інформаційних систем.

2.2 Призначення розробки — визначається необхідністю створення системи управління кешем даних, яка забезпечить ефективне зберігання, обробку та швидкий доступ до інформації у розподіленому середовищі. Система дозволяє адміністратору керувати кешованими даними, виконувати операції додавання, оновлення та видалення записів, а також отримувати статистичну інформацію про продуктивність та стан сховища у реальному часі за допомогою мережевих сервісів.

3 Вихідні дані для виконання МКР

3.1 Проведення аналізу існуючих методів та принципів.

3.2 Розробка алгоритму роботи веб- додатку.

3.4 Проведення верифікації та аналізу отриманих результатів.

3.5 Виконання розрахунків для доведення доцільності нової розробки з економічної точки зору.

4 Вимоги до виконання МКР

Головна вимога — щоб система забезпечувала ефективне управління кешованими даними та інформаційну підтримку адміністратора щодо стану сховища і доступності даних. Вона повинна забезпечувати швидкий доступ до записів у режимі реального часу, відображати стан кешу, статистику використання ресурсів, історію операцій додавання, оновлення та видалення даних, а також надавати аналітичну інформацію про продуктивність та ефективність роботи системи.

5 Етапи МКР та очікувані результати

Етапи роботи та очікувані результати приведено в Таблиці А.1.

Таблиця А.1 — Етапи МКР

№ етапу	Назва етапу	Термін виконання		Очікувані результати
		початок	кінець	
1	Аналіз існуючих рішень			Розділ 1
2	Визначення архітектури системи			Розділ 2
3	Розробка алгоритму та функціоналу комп’ютерної системи			Розділ 3
4	Тестування системи			Розділ 4
4	Підготовка економічної частини			Розділ 5
6	Оформлення пояснювальної записки, графічного матеріалу і презентації			ПЗ, графічний матеріал і презентація
7	Підготовка і підпис супроводжуючих документів, нормоконтроль та тест на плагіат			Оформленні документи

6 Матеріали, що подаються до захисту МКР

До захисту подаються: пояснювальна записка МКР, графічні і ілюстративні матеріали, протокол попереднього захисту МКР на кафедрі, відгук наукового керівника, відгук опонента, протоколи складання державних екзаменів, анотації до МКР українською та іноземною мовами.

7 Порядок контролю виконання та захисту МКР

Виконання етапів графічної та розрахункової документації МКР контролюється науковим керівником згідно зі встановленими термінами. Захист МКР відбувається на засіданні Екзаменаційної комісії, затвердженої наказом ректора.

8 Вимоги до оформлювання та порядок виконання МКР

8.1 При оформлюванні МКР використовуються:

— ДСТУ 3008: 2015 «Звіти в сфері науки і техніки. Структура та правила оформлювання»;

— ДСТУ 8302: 2015 «Бібліографічні посилання. Загальні положення та правила складання»;

— ГОСТ 2.104–2006 «Єдина система конструкторської документації. Основні написи»;

— методичні вказівки до виконання магістерських кваліфікаційних робіт зі спеціальності 123 — «Комп’ютерна інженерія»;

— документи на які посилаються у вище вказаних.

8.2 Порядок виконання МКР викладено в «Положення про кваліфікаційні роботи на другому (магістерському) рівні вищої освіти СУЯ ВНТУ–03.02.02 П.001.01:21

ДОДАТОК Б

Протокол перевірки кваліфікаційної роботи

Назва роботи: ____ Сервіс управління кешем даних за технологією розподіленого key-value сховища._____

Тип роботи: _____ магістерська кваліфікаційна робота _____

Підрозділ: _____кафедра обчислювальної техніки _____

Коефіцієнт подібності текстових запозичень, виявлених у роботі системою StrikePlagiarism (КП1) 1 %

✓ Висновок щодо перевірки кваліфікаційної роботи (відмітити потрібне)
Запозичення, виявлені у роботі, оформлені коректно і не містять ознак академічного плагіату, фабрикації, фальсифікації. Роботу прийняти до захисту.

У роботі не виявлено ознак плагіату, фабрикації, фальсифікації, але надмірна кількість текстових запозичень та/або наявність типових розрахунків не дозволяють прийняти рішення про оригінальність та самостійність її виконання. Роботу направити на доопрацювання.

У роботі виявлено ознаки академічного плагіату та/або в ній містяться навмисні спотворення тексту, що вказують на спроби приховування недобросовісних запозичень. Робота до захисту не приймається.

Експертна комісія:

_____Азаров О. Д. д.т.н. зав. каф. ОТ_____ _____

_____Мартинюк Т. Б. _ д.т.н. зав. каф. ОТ __ _____

Особа, відповідальна за перевірку _____ ___Захарченко С. М.__

З висновком експертної комісії ознайомлений(-на)

Керівник _____ _____Савицька Л. А._____

Здобувач _____ _____Димпалов Є. І. _____

ДОДАТОК В

Схема архітектурного потоку даних при бінарному зберіганні та відновленні у системі Muninn

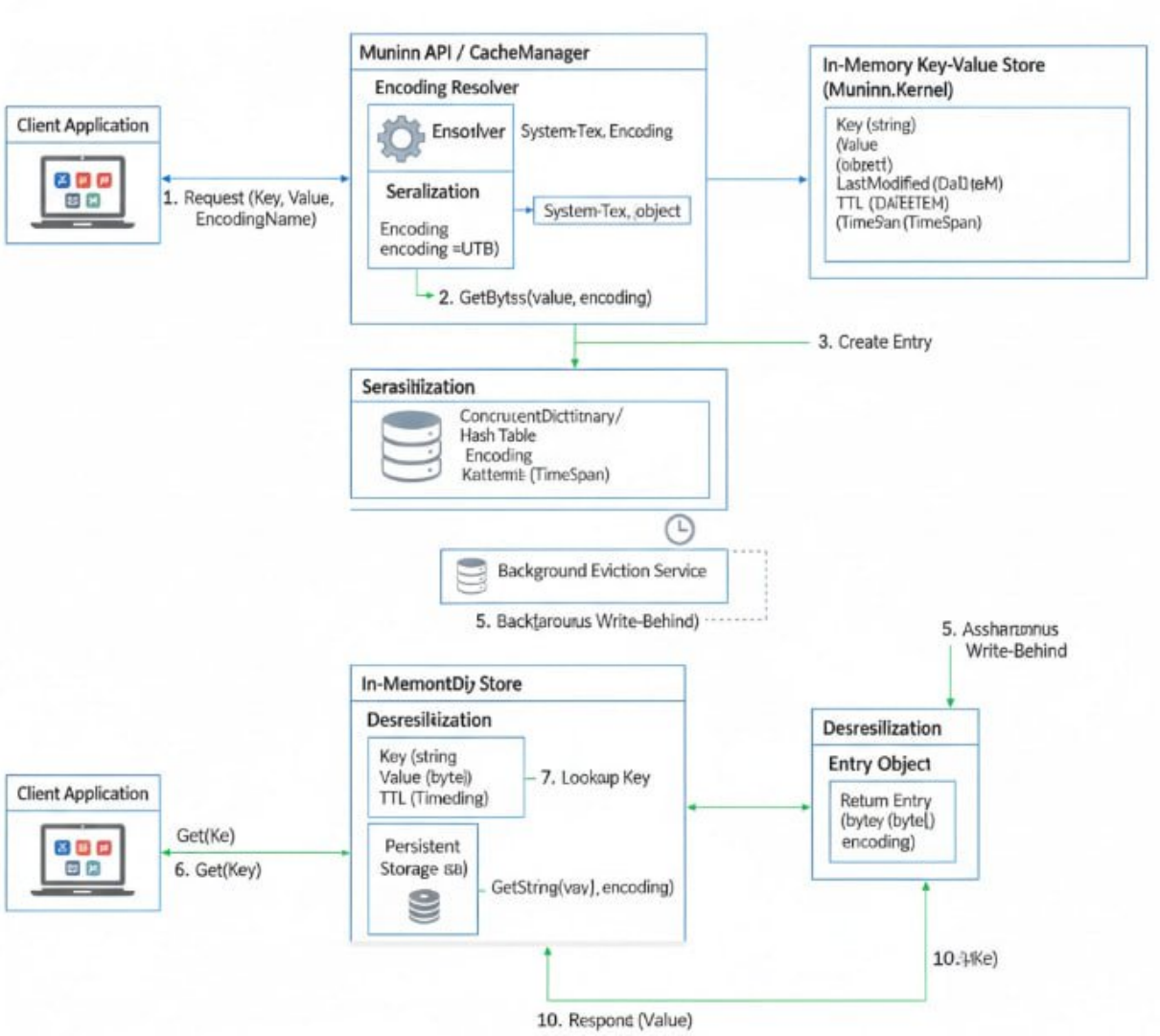

Рисунок В1. — Архітектурний потік даних при бінарному зберіганні та відновленні у системі Muninn

ДОДАТОК Г

Діаграма варіантів використання (Use Case Diagram) системи керування кешем Muninn

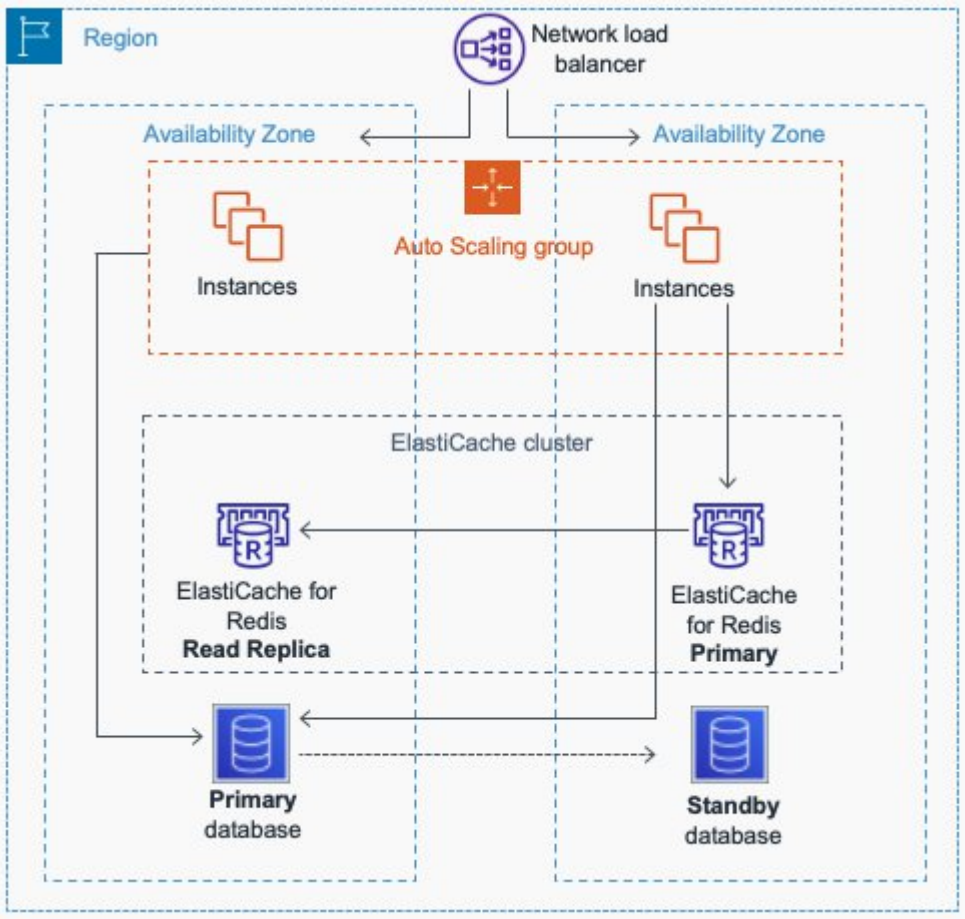

Рисунок Г1. — Діаграма варіантів використання системи керування кешем Muninn

ДОДАТОК Д

Архітектура кластерного Key-Value сховища з проксі-сервером та централізованим керуванням станом

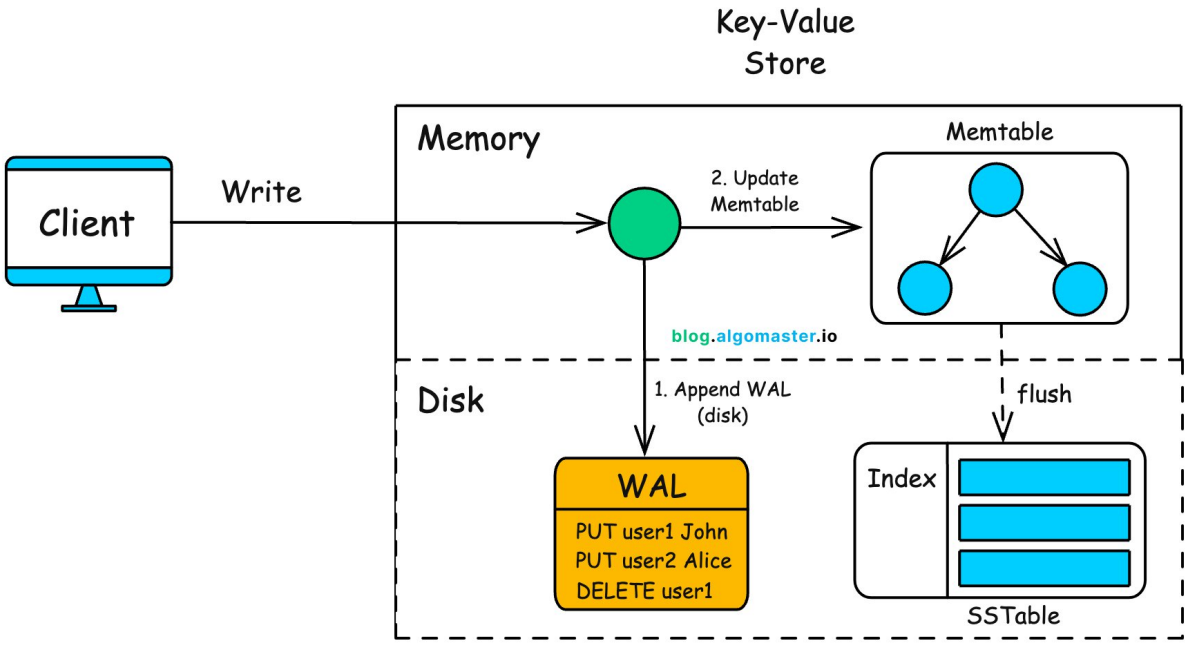

Рисунок Д1. — Схема архітектури кластерного Key-Value сховища з проксі-сервером та централізованим керуванням станом

ДОДАТОК Е

Лістинг програмного забезпечення

Лістинг Е1. — Реєстрація REST ендпоінтів (файл Register.cs)

```
public static IEndpointRouteBuilder MapEndpoints(this
IEndpointRouteBuilder app)
{
    var group = app.MapGroup("muninn");
    group.MapPost("{key}", PostAsync);
    group.MapPost("{key}/insert", InsertAsync);
    group.MapPut("{key}", PutAsync);
    group.MapDelete("{key}", DeleteAsync);
    group.MapDelete(string.Empty, DeleteAllAsync);
    group.MapGet("{key}", GetAsync);

    return group;
}
```

Лістинг Е2. — Реалізація REST ендпоінту PostAsync

```
private static async Task<IResult> PostAsync([FromBody] PostRequest
request,
    [FromServices] ICacheManager cacheManager, CancellationToken
cancellationToken)
{
    var encoding = Encoding.GetEncoding(request.Body.EncodingName);
    var entry = new Entry(request.Key,
encoding.GetBytes(request.Body.Value), encoding);
    var lifeTime = request.Body.LifeTime;

    if (lifeTime > TimeSpan.Zero)
    {
```

```
            entry.LifeTime = lifeTime;
        }

        var result = await cacheManager.AddAsync(entry, cancellationToken);

        return GetResponse(result, true);
    }
```

Лістинг Е3. – Реалізація операції додавання у CacheManager

```
    public async Task<MuninnResult> AddAsync(Entry entry,
CancellationToken cancellationToken)
    {
        var residentResult = await _residentCache.AddAsync(entry,
cancellationToken);

        if (residentResult.IsSuccessful)
        {
            // Асинхронний виклик опціональних хендлерів (наприклад, для
PersistentCache)
            _ = _handlers.Select(handler => handler.AddAsync(entry,
cancellationToken));
        }

        return residentResult;
    }
```

Лістинг Е4. — Реєстрація опціональних модулів у DI Container (AddOptionalCache)

```
    public static IServiceCollection AddOptionalCache(this IServiceCollection
services, string args)
```

```
        {
            if (args.Contains("--sort"))
            {
                services.AddSingleton<ISortedResidentCache, SortedResidentCache>();
                services.TryAddEnumerable(new
ServiceDescriptor(typeof(IBaseCache), typeof(SortedResidentCache),
                    ServiceLifetime.Singleton));
                services.TryAddEnumerable(new
ServiceDescriptor(typeof(IOptionalCacheHandler),
                    typeof(SortedResidentCacheHandler), ServiceLifetime.Singleton));
            }

            if (args.Contains("--persistent"))
            {
                services.AddSingleton<IPersistentCache, PersistentCache>();
                services.TryAddEnumerable(new
ServiceDescriptor(typeof(IBaseCache), typeof(PersistentCache),
                    ServiceLifetime.Singleton));
                services.TryAddEnumerable(new
ServiceDescriptor(typeof(IOptionalCacheHandler),
                    typeof(PersistentCacheHandler), ServiceLifetime.Singleton));
            }

            return services;
        }
```

Лістинг Е5. – Фонова служба очищення прострочених записів (CacheLifeTimeBackgroundService)

```
        public class CacheLifeTimeBackgroundService : BackgroundService
        {
```

```
        private readonly ILogger _logger;
        private readonly ICacheManager _cacheManager;
        private readonly TimeSpan _delayTime = TimeSpan.FromSeconds(10);

        public CacheLifeTimeBackgroundService(ILogger<BackgroundService>
logger, ICacheManager cacheManager)
        {
            _logger = logger;
            _cacheManager = cacheManager;
        }

        protected override async Task ExecuteAsync(CancellationToken
stoppingToken)
        {
            while (!stoppingToken.IsCancellationRequested)
            {
                var result = await _cacheManager.GetAllAsync(true, stoppingToken);

                // Фільтрація записів, термін дії яких закінчився
                var targets = result.Where(entry
=> !entry.LifeTime.Equals(TimeSpan.Zero) &&
entry.LastModificationTime.Add(entry.LifeTime) < DateTime.UtcNow).ToList();

                if (targets.Any())
                {
                    var maxThreadsCount = Math.Min(targets.Count, 100);
                    var parallelOptions = new ParallelOptions
{ MaxDegreeOfParallelism = maxThreadsCount, CancellationToken =
stoppingToken };
```

```
                    // Паралельне видалення прострочених записів
                    await Parallel.ForEachAsync(targets, parallelOptions, async (target,
cancellationToken) =>
                    {
                        await _cacheManager.RemoveAsync(target.Key,
cancellationToken);
                    });
                }
                await Task.Delay(_delayTime, stoppingToken);
            }
_logger.LogBackgroundServiceShutdown(nameof(CacheLifeTimeBackgroundServi
ce));
        }
    }
```

Лістинг Е6. – Допоміжні функції (BaseCache)

```
    public abstract class BaseCache<TSelf> (ILogger<TSelf> logger,
IFilterService filterService)
    {
        internal const int DefaultIncreaseValue = 1000;
        internal const int InitialArraySize = 10_000;
        protected readonly ILogger _logger = logger;
        protected readonly IFilterService _filterService = filterService;
        protected readonly SemaphoreSlim _semaphoreSlim = new(1);
        protected MuninnResult GetCancelledResult(string methodName, string
key, OperationCanceledException operationCanceledException)
        {
            _logger.LogCancelledRequest(methodName, key,
operationCanceledException);
            return GetFailedResult("Cancellation has been requested", true,
operationCanceledException);
```

```
        }
        protected MuninnResult GetCancelledResult(OperationCanceledException
operationCanceledException)
        {
            _logger.LogCancelledRequest(operationCanceledException);
            return GetFailedResult("Cancellation has been requested", true,
operationCanceledException);
        }
        protected static MuninnResult GetSuccessfulResult(Entry? entry = null) =>
new(true, entry);
        protected static MuninnResult GetFailedResult(string message, bool
isCanceled, Exception? exception = null) => new(false, null, message, isCanceled,
exception);
    }
```

Лістинг Е7. – Фрагмент коду для зміни розміру масиву (Resize)

```
    // Цей фрагмент, ймовірно, знаходиться всередині методу, захищеного
SemaphoreSlim
    /*
    await _semaphoreSlim.WaitAsync(cancellationToken);
    _isResizeCalled = true;
    var size = _count + DefaultIncreaseValue;
    var entries = new Entry?[size];
    var indexes = new ResidentCacheIndex[size];

    for (var i = 0; i < _entries.Length; i++)
    {
        entries[i] = _entries[i];
        indexes[i] = _indexes[i];
        indexes[size + i - _indexes.Length] = new();
    }
```

```
_entries = entries;
_indexes = indexes;
_logger.LogIncreasedSize(size);
return GetSuccessfulResult();
*/
```