Міністерство освіти і науки України
Харківський національний університет радіоелектроніки

Факультет комп'ютерних наук
(повна назва)

Кафедра програмної інженерії
(повна назва)

КВАЛІФІКАЦІЙНА РОБОТА
Пояснювальна записка

рівень вищої освіти другий (магістерський)

Дослідження алгоритмів балансування навантаження для підвищення продуктивності програмних систем на .NET Core
(тема)

Виконав:
здобувач 2 року навчання
групи ІПЗм-23-1

Сергій МИРОШНИЧЕНКО
(Власне ім'я, ПРІЗВИЩЕ)

Спеціальність 121 – Інженерія програмного забезпечення
(код і повна назва спеціальності)
Тип програми освітньо-наукова

Керівник доц. Наталія ГОЛЯН
(посада, Власне ім'я, ПРІЗВИЩЕ)

Допускається до захисту
Зав. кафедри _____ Кирило СМЕЛЯКОВ
(підпис) (Власне ім'я, ПРІЗВИЩЕ)

2025 р.

Харківський національний університет радіоелектроніки

Факультет комп’ютерних наук
Кафедра програмної інженерії
Рівень вищої освіти другий (магістерський)
Спеціальність 121 – Інженерія програмного забезпечення
Тип програми освітньо-наукова програма
Освітня програма Інженерія програмного забезпечення
(шифр і назва)

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

ЗАВДАННЯ
НА КВАЛІФІКАЦІЙНУ РОБОТУ

здобувачу Мирошниченко Сергію Анатолійовичу
(прізвище, ім’я, по батькові)

1. Тема роботи «Дослідження алгоритмів балансування навантаження для підвищення продуктивності програмних систем на .NET Core»
Затверджена наказом по університету від 15.04. 2025р. № 290 Ст
2. Термін подання студентом роботи до екзаменаційної комісії 19.06.2025
3. Вихідні дані до роботи проведення аналізу сучасних підходів до балансування навантаження, огляд наукових джерел, дослідження ефективності адаптивних алгоритмів, а також оцінка можливостей їх інтеграції в розподілені системи на платформі .NET
4. Перелік питань, що потрібно опрацювати в роботі
аналіз існуючих алгоритмів балансування навантаження, обґрунтування вибору адаптивного підходу до розподілу HTTP-запитів, проєктування архітектури системи з урахуванням модульності та масштабованості, реалізація програмного компонента балансування у середовищі .NET

КАЛЕНДАРНИЙ ПЛАН

№	Назва етапів роботи	Термін виконання етапів роботи	Примітка
1	Отримання завдання	15.04.2025	*виконано*
2	Аналіз предметної галузі і постановка задачі	20.04.2025	*виконано*
3	Аналіз існуючих підходів та методів	01.05.2025	*виконано*
4	Проектування системи та розробка алгоритму	10.05.2025	*виконано*
5	Проведення дослідження	15.05.2025	*виконано*
6	Підготовка до апробації результатів дослідження. Публікація матеріалів	03.06.2025	*виконано*
7	Підготовка пояснювальної записки	04.06.2025	*виконано*
8	Підготовка презентації та доповіді	05.06.2025	*виконано*
9	Перевірка на плагіат	11.06.2025	*виконано*
10	Нормоконтроль	11.06.2025	*виконано*
11	Рецензування	12.06.2025	*виконано*
12	Попередній захист	16.06.2025	*виконано*
13	Занесення диплома в електронний архів	17.06.2025	*виконано*
14	Допуск до захисту у зав. кафедри	18.06.2025	*виконано*

Дата видачі завдання 15 квітня 2025р.

Студент (ка) _____ Сергій МИРОШНИЧЕНКО
(підпис)

Керівник роботи _____ доц. Наталія ГОЛЯН
(підпис) (посада, Власне ім’я, ПРІЗВИЩЕ)

РЕФЕРАТ

Пояснювальна записка: 75 с., 34 рис., 13 джерел, 4 додатків.

АДАПТИВНІ АЛГОРИТМИ, АЛГОРИТМИ ROUND ROBIN, ДИНАМІЧНИЙ ТРАФІК, МОДЕЛЮВАННЯ НАВАНТАЖЕННЯ, РОЗПОДІЛ ЗАПИТІВ, СИСТЕМА-ЕМУЛЯТОР, СЕРВЕРНІ СИСТЕМИ, БАЛАНСУВАННЯ НАВАНТАЖЕННЯ, LEAST CONNECTIONS, WEIGHTED ROUND ROBIN.

Об’єктом дослідження є система розподілу запитів та балансування навантаження у серверних комплексах.

Предметом дослідження є методи та алгоритми балансування навантаження з урахуванням динаміки трафіку і різнорідності серверних вузлів.

Мета роботи - розробка та дослідження ефективного адаптивного алгоритму балансування навантаження, що забезпечує стійку роботу серверів в умовах високої динаміки трафіку.

Методи дослідження - аналіз існуючих класичних і адаптивних алгоритмів балансування (Round Robin, Least Connections, Weighted Round Robin тощо). Розробка програмної системи-емулятора з можливістю автоматичного перемикання між алгоритмами і збором метрик у реальному часі. Впровадження і тестування адаптивного алгоритму, який враховує поточні й історичні параметри серверів. Моделювання навантаження за допомогою інструменту k6 та аналіз результатів роботи системи.

В результаті було розроблено систему для моделювання балансування навантаження, у якій адаптивний алгоритм продемонстрував вищу ефективність і стабільність порівняно з класичними методами, адаптуючись до змін стану серверів та запобігаючи їх перевантаженню.

ABSTRACT

ADAPTIVE ALGORITHMS, ROUND ROBIN ALGORITHMS, DYNAMIC TRAFFIC, LOAD MODELING, REQUEST DISTRIBUTION, EMULATOR SYSTEM, SERVER SYSTEMS, LOAD BALANCING, LEAST CONNECTIONS, WEIGHTED ROUND ROBIN.

The object of research is the system of request distribution and load balancing in server complexes.

The subject of the study is methods and algorithms for load balancing taking into account traffic dynamics and heterogeneity of server nodes.

Purpose - to develop and study an effective adaptive load balancing algorithm that ensures stable operation of servers in conditions of high traffic dynamics.

Research methods - analysis of existing classical and adaptive balancing algorithms (Round Robin, Least Connections, Weighted Round Robin, etc.). Development of a software emulator system with the ability to automatically switch between algorithms and collect metrics in real time. Implementation and testing of an adaptive algorithm that takes into account current and historical server parameters. Modeling the load using the k6 tool and analyzing the system's performance.

As a result, a system for modeling load balancing was developed, in which the adaptive algorithm demonstrated higher efficiency and stability compared to classical methods, adapting to changes in the state of servers and preventing them from overloading.

Завідувачу кафедри ПІ
проф. Кирилу СМЕЛЯКОВУ

ЗАЯВА

щодо самостійності виконання кваліфікаційної роботи та можливості її публікації (та/або публікації анотації кваліфікаційної роботи) в електронному архіві відкритого доступу ElAr KhNURE

Я, Мирошниченко Сергій Анатолійович, здобувач вищої освіти на другому (магістерському) рівні вищої освіти академічної групи ІПЗм-23-1 кафедра програмної інженерії, заявляю: моя кваліфікаційна робота на тему «Дослідження алгоритмів балансування навантаження для підвищення продуктивності програмних систем на .NET Core», що буде представлена в екзаменаційну комісію для публічного захисту, виконана самостійно, в ній не містяться елементи плагіату і вона може бути опублікована в репозиторії "ElArKhNURE". Погоджуюся з авторським договором, відповідно до Положення про репозиторій ХНУРЕ "ElArKhNURE". Всі запозичення з друкованих та електронних джерел мають відповідні посилання.

Я ознайомлений з вимогами академічної доброчесності, згідно з якими виявлення плагіату є підставою для відмови в допуску кваліфікаційної роботи до захисту та застосування дисциплінарних заходів.

11.06.2025  Сергій МИРОШНИЧЕНКО

ЗМІСТ

ВСТУП

Інтенсивне зростання кількості запитів до веб-систем та сервісів створює критичну потребу в ефективних механізмах розподілу навантаження. Питання балансування трафіку між серверними екземплярами сьогодні є актуальним для забезпечення високої доступності, надійності та масштабованості веб-застосунків, особливо в умовах обмежених ресурсів серверної інфраструктури. В умовах динамічно мінливого навантаження традиційні методи, які базуються виключно на статичних алгоритмах, часто виявляються неефективними, що зумовлює необхідність пошуку і вдосконалення більш адаптивних рішень.

Актуальність теми визначається необхідністю створення та застосування легковагових компактних систем балансування навантаження, особливо для проектів, де неможливе або недоцільне використання складних та дорогих апаратних або програмних балансувальників. Вибір та порівняння оптимальних алгоритмів балансування, здатних ефективно адаптуватися до реального стану серверів, дозволить значно підвищити якість обслуговування запитів у розподілених системах.

Зв'язок роботи з програмами наукових досліджень кафедри програмної інженерії полягає у її інтеграції в загальний напрямок досліджень кафедри щодо розробки та вдосконалення програмних рішень для підвищення надійності та продуктивності розподілених обчислювальних систем. Робота відповідає стратегічним напрямкам кафедри, спрямованим на розробку алгоритмів та програмних комплексів, що забезпечують адаптивність і стійкість до змін навантаження.

Метою даної роботи є розробка компактної системи розподілу HTTP-запитів із реалізацією та порівняльним аналізом ефективності базових алгоритмів балансування в умовах симульованого навантаження.

Для досягнення цієї мети поставлені такі задачі:

– розробити архітектуру компактної системи розподілу запитів на базі платформи .net;

– реалізувати базові алгоритми балансування навантаження: round robin, weighted round robin, least connections та adaptive;
– налаштувати середовище симуляції навантаження за допомогою інструменту k6 та створити сценарії імітації різних типів навантаження;
– провести експериментальне тестування системи з використанням різних алгоритмів та виконати порівняльний аналіз отриманих результатів.

У дослідженні використано методи порівняльного аналізу (для оцінки ефективності різних алгоритмів балансування), емпіричного тестування (для проведення практичних експериментів та збору метрик), а також статистичні методи обробки результатів експериментів для забезпечення їх достовірності та інформативності.

Елементом наукової новизни роботи є запропонована компактна архітектура системи балансування, що дозволяє динамічно перемикатися між різними алгоритмами балансування залежно від поточного стану серверів та сценарію навантаження. Одержані результати щодо порівняння ефективності адаптивних та статичних алгоритмів вперше узагальнено для застосування у компактних .NET-рішеннях, що є новим внеском у розробку програмних систем розподіленого типу.

Практичне значення одержаних результатів полягає у можливості застосування розроблених рекомендацій щодо вибору алгоритму балансування в реальних проектах, де важливо забезпечити ефективність розподілу навантаження без значних витрат на інфраструктуру. Запропоновані рішення можуть бути впроваджені у малих і середніх підприємствах, що використовують .NET-технології для побудови своїх інформаційних систем, про що свідчать відповідні практичні рекомендації у даній роботі.

1 АНАЛІЗ ПРЕДМЕТНОЇ ГАЛУЗІ

1.1 Аналіз проблематики серверного розподілення

Невід'ємною частиною будь-якої сучасної інформаційної системи є сервери, що забезпечують обробку клієнтських запитів. Сервери являють собою програмно-апаратні вузли, призначені для надання різноманітних послуг користувачам через інтернет або внутрішні мережі. Вони здатні обробляти HTTP-запити, здійснювати операції з базами даних, виконувати бізнес-логіку, а також повертати результати виконання запитів користувачам. У системах, де кількість користувачів і навантаження постійно змінюється, важливим завданням є рівномірний і ефективний розподіл цих запитів між доступними серверами.

Запити надходять на сервери через так званий балансувальник навантаження (load balancer). Він діє як проміжний шар між користувачем та сервером, визначаючи, який саме сервер має обробити конкретний запит. Це дозволяє уникати ситуацій, коли один сервер перевантажений, а інші залишаються недовантаженими. Правильно організований розподіл навантаження дозволяє забезпечити стабільність і безперервність роботи сервісів навіть за умови високих навантажень або часткових відмов окремих вузлів.

Типовими метриками, що визначають стан сервера, є завантаження центрального процесора (CPU), кількість активних з'єднань, загальна кількість оброблених запитів, час відповіді (latency) та кількість помилок (error rate). Ці метрики важливі для оцінки поточного стану вузла, що в подальшому використовується алгоритмами балансування для прийняття рішень щодо маршрутизації запитів. Якщо, наприклад, один сервер демонструє високе завантаження процесора або значну кількість активних з'єднань, балансувальник навантаження може спрямувати нові запити до інших серверів, які наразі мають менше навантаження.

При організації балансування запитів найчастіше використовують кілька основних підходів. Простий алгоритм Round Robin рівномірно розподіляє запити по серверах у порядку черги. Більш складні алгоритми, такі як Weighted Round Robin або Least Connections, враховують вагу серверів чи їх поточний стан,

спрямовуючи запити на найменш завантажені вузли. Адаптивні алгоритми (Adaptive Balancing) додатково аналізують поточний стан серверів за різними метриками (наприклад, CPU-завантаження, кількість активних з'єднань) і динамічно регулюють маршрутизацію запитів для досягнення максимальної ефективності та уникнення перевантажень [1].

У реальних сценаріях трафік на сервери може бути нерівномірним і змінюватися в залежності від часу доби, дня тижня, або навіть у зв'язку із зовнішніми подіями, такими як проведення маркетингових кампаній чи виходу нових релізів продуктів. Для імітації подібних умов у дослідженнях використовуються спеціалізовані засоби генерації навантаження (наприклад, K6), що дозволяють створювати контрольовані сценарії трафіку та оцінювати реакцію серверів і алгоритмів балансування на різні умови роботи.

Це дає змогу детально дослідити особливості роботи кожного алгоритму, визначити найкращі стратегії балансування та підготувати рекомендації щодо їх практичного застосування.

Проблематика серверного розподілення запитів виникає через постійно зростаючу кількість клієнтських звернень до веб-систем, що створює значні навантаження на серверну інфраструктуру. Збільшення кількості запитів у пікові моменти призводить до погіршення якості обслуговування, збільшення часу відповіді і навіть часткових відмов роботи сервісів. Крім того, нерівномірне навантаження між серверами може спричиняти їх перевантаження, що негативно впливає на стабільність і відмовостійкість усієї системи.

Основною причиною цих проблем є обмеженість ресурсів серверів, таких як потужність процесорів, оперативна пам'ять, пропускна здатність мережі, а також наявність великої кількості одночасних з'єднань, які потребують швидкої та ефективної обробки. Особливо критичною ця ситуація стає у випадку різких, непередбачуваних коливань навантаження, які виникають у моменти запуску нових продуктів, рекламних акцій або сезонних пікових періодів.

Крім того, складність управління розподіленими системами зростає із необхідністю забезпечення високої доступності і безперервності роботи. В умовах

використання класичних статичних методів балансування, які не враховують динамічний стан серверів, виникають проблеми неефективного використання ресурсів, що веде до надмірних витрат або недосягнення очікуваної продуктивності.

Зазначені фактори зумовлюють необхідність розробки та впровадження більш адаптивних та інтелектуальних алгоритмів балансування навантаження, здатних оперативно аналізувати поточний стан системи, прогнозувати можливі ускладнення та ефективно розподіляти запити між серверами. Таким чином, сучасні дослідження у цій галузі спрямовані на пошук оптимальних підходів до реалізації адаптивного балансування, яке дозволить забезпечити стабільну і надійну роботу серверних комплексів в умовах високих навантажень та мінливої активності користувачів.

1.2 Загальні відомості про системи розподілення запитів

Системи розподілення запитів призначені для рівномірного розподілу вхідних запитів між кількома серверами або вузлами мережі. Основною метою цих систем є оптимізація використання ресурсів серверів та забезпечення стабільної роботи веб-додатків і сервісів. Сучасні системи балансування запитів класифікуються залежно від рівня мережевої моделі, на якому вони оперують: L4 (транспортний рівень) і L7 (прикладний рівень).

Балансувальники рівня L4 працюють з мережевими протоколами (TCP/UDP) і використовують базові характеристики мережевих з'єднань, такі як IP-адреси та порти, для маршрутизації запитів. Вони прості у використанні та швидкі в роботі, але не можуть аналізувати вміст запитів, що обмежує можливості динамічного управління навантаженням.

Балансувальники рівня L7, у свою чергу, аналізують HTTP-запити та можуть здійснювати більш складні рішення щодо маршрутизації, такі як визначення типу запиту, перевірку заголовків і вмісту запиту, що дозволяє більш тонко налаштовувати стратегії балансування [2].

Серед популярних алгоритмів, що застосовуються у системах розподілення запитів, можна виділити Round Robin, Weighted Round Robin, Least Connections, Weighted Least Connections та адаптивні алгоритми, які враховують поточний стан серверів, такі як завантаження CPU або кількість активних підключень.

Важливою складовою сучасних систем балансування є механізми моніторингу, які дозволяють в реальному часі оцінювати продуктивність серверів та оперативно реагувати на зміни навантаження. Використання таких систем суттєво підвищує ефективність використання серверних ресурсів і забезпечує стабільну роботу інформаційних сервісів навіть у випадку несподіваних змін навантаження.

Сучасні складні системи розподілу запитів, такі як апаратні балансувальники, спеціалізовані програмні рішення або інтегровані платформи на кшталт Kubernetes, забезпечують широкі можливості тонкого налаштування і високий рівень автоматизації. Проте разом із широкими можливостями зростає їх складність, витрати на розгортання, налаштування та супровід. Саме тому у певних випадках, особливо для малих і середніх проєктів, де інфраструктурні витрати та ресурсна база обмежені, актуальними залишаються більш прості і компактні системи розподілу запитів.

Такі спрощені варіанти можуть базуватися на класичних алгоритмах, як-от Round Robin або Least Connections, що легко реалізуються навіть на стандартних платформах без необхідності залучення додаткових ресурсів чи сторонніх рішень. Простота реалізації, низькі витрати на підтримку та можливість швидкої інтеграції в існуючі системи роблять ці алгоритми популярними та доцільними для широкого спектру веб-додатків і невеликих розподілених інформаційних систем. Таким чином, прості системи розподілення запитів зберігають свою актуальність, ефективно закриваючи потребу в надійному та доступному балансуванні навантаження [3].

1.3 Актуальність проблеми

Сучасні веб-додатки та інформаційні системи часто стикаються з істотним зростанням кількості користувацьких запитів. Це спричиняє високі вимоги до серверної інфраструктури, яка повинна витримувати постійні навантаження, пікові сплески та забезпечувати стабільність роботи сервісів.

Основним способом вирішення цієї проблеми є застосування систем балансування навантаження, які дозволяють розподіляти вхідні запити між кількома серверами, оптимізуючи використання наявних ресурсів та покращуючи загальну продуктивність системи.

Однак аналіз сучасних підходів та рішень показує, що більшість доступних систем розподілу запитів характеризуються складністю розгортання, налаштування та інтеграції. Багато з таких рішень потребують спеціалізованого обладнання або програмних платформ, таких як апаратні балансувальники або комплексні системи оркестрації на зразок Kubernetes. Це створює значні перешкоди для малих і середніх проєктів, які працюють з обмеженими ресурсами і для яких використання складних інструментів є економічно необґрунтованим.

Особливо гостро ця проблема проявляється на платформі .NET, яка активно використовується у корпоративному сегменті, але не завжди має широкий спектр доступних і зручних рішень для балансування навантаження.

В результаті, розробники змушені реалізовувати власні, часто спрощені та неефективні рішення, або ж адаптувати сторонні інструменти, що значно ускладнює розробку і підтримку таких систем. Додатковою перешкодою є відсутність доступних інструментів для тестування та порівняння ефективності різних алгоритмів балансування без складних налаштувань та значних витрат ресурсів і часу. Це обмежує можливість швидко оцінювати, обирати і адаптувати найкращі рішення для конкретних умов проєкту.

Ще однією суттєвою проблемою, що ускладнює вибір оптимального рішення, є відсутність універсального алгоритму балансування запитів, який міг би забезпечити найкращі результати у будь-яких сценаріях навантаження. Кожен з наявних алгоритмів (Round Robin, Weighted Round Robin, Least Connections,

Adaptive та інші) має свої сильні та слабкі сторони. Наприклад, прості алгоритми, такі як Round Robin, легко реалізуються і добре працюють в умовах стабільного, передбачуваного навантаження, але стають неефективними при різких коливаннях або при істотних відмінностях у потужності серверів.

Адаптивні алгоритми, що враховують метрики завантаження серверів, можуть ефективніше розподіляти навантаження в складних умовах, але потребують додаткових ресурсів на моніторинг, збір та обробку даних про стан системи.

Таким чином, вибір оптимального алгоритму для конкретної ситуації залишається непростим завданням, що потребує врахування специфіки конкретного середовища, особливостей роботи додатку та його навантажувальних сценаріїв. Відсутність єдиного універсального рішення зумовлює актуальність розробки доступних, зручних і ефективних інструментів та систем для балансування навантаження на платформі .NET, які дозволили б швидко тестувати і порівнювати різні алгоритми у різноманітних умовах і вибирати найкращий варіант для конкретного застосунку.

1.4 Постановка задачі

У контексті викладених вище проблем і обмежень, основною задачею дослідження є створення компактної та доступної системи розподілу запитів, яка б дозволяла здійснювати тестування і порівняння ефективності базових алгоритмів балансування у реальних або наближених до них умовах навантаження. Така система повинна бути орієнтована на платформу .NET, що забезпечить її сумісність і зручність використання у проєктах, які активно застосовують цю технологію.

Для досягнення цієї загальної мети визначено такі конкретні завдання:

– спроектувати архітектуру компактної системи розподілу запитів, яка включатиме:
– сервери, що імітують роботу справжніх вузлів і передають свої метрики (cpu, активні запити, загальна кількість оброблених запитів);

- центральний модуль балансування запитів, який здійснює вибір оптимального сервера на основі поточних метрик і обраного алгоритму;
- реалізувати базові алгоритми балансування – round robin, weighted round robin, least connections та adaptive, а також надати можливість легко перемикати їх у процесі тестування для проведення порівняльного аналізу;
- інтегрувати механізм збору метрик з серверів, що дозволить отримувати актуальну інформацію про стан кожного вузла кожні 5 секунд і використовувати її для прийняття рішень;
- організувати модуль генерації навантаження з використанням інструменту k6 для імітації різних сценаріїв роботи системи: рівномірне навантаження, пікові сплески, хаотичне та непередбачуване навантаження;
- провести експериментальні дослідження, у ході яких оцінити: загальний час відповіді (latency), розподіл запитів між серверами, кількість помилок та відмов, адаптивність системи до змін навантаження;
- проаналізувати отримані результати для кожного алгоритму балансування, порівняти їх ефективність і сформулювати рекомендації щодо використання у різних сценаріях навантаження та для різних типів застосунків.

Таким чином, виконання цих завдань дозволить не лише розробити практичний інструмент для тестування алгоритмів балансування у .NET-середовищі, а й отримати нові знання про переваги та обмеження кожного підходу, що стане підґрунтям для подальшої розробки більш універсальних та ефективних рішень у сфері розподілу навантаження.

2 АНАЛІЗ ІСНУЮЧИХ ПІДХОДІВ ТА МЕТОДІВ

2.1 Методи та алгоритми балансування

У контексті забезпечення стабільної та ефективної роботи веб-застосунків особливе значення мають алгоритми та методи балансування навантаження. Вони визначають, як саме розподіляються запити користувачів між доступними серверами і наскільки оптимально використовуються ресурси кожного вузла системи. Основна мета цих методів — уникнути перевантаження окремих серверів, забезпечити швидкий відгук системи і знизити ймовірність відмов при зростанні обсягу трафіку.

У рамках дослідження планується зосередитися на базових, класичних методах балансування, які застосовуються у більшості сучасних рішень та складають основу для розробки більш складних адаптивних стратегій. До таких методів належать:

- Round Robin – рівномірний розподіл запитів між серверами у циклічному порядку, незалежно від їхнього поточного стану;
- Weighted Round Robin – модифікація попереднього методу, яка враховує вагові коефіцієнти серверів, дозволяючи розподіляти більше запитів на більш потужні вузли;
- Least Connections – підхід, який орієнтується на кількість активних з’єднань, спрямовуючи нові запити до того сервера, який має найменше активних підключень у даний момент;
- Adaptive (Resource-Based) – алгоритм, що використовує актуальні дані про стан серверів (CPU, пам’ять, кількість активних з’єднань) для прийняття рішень про маршрутизацію запитів, забезпечуючи динамічне та більш гнучке балансування.

Також будуть розглянуті підходи до балансування які будуть задіяні в алгоритмах.

Надалі кожен з перелічених методів буде детально розглянутий окремо, зокрема його архітектура, принципи роботи, переваги та недоліки. Такий підхід дозволить сформувати цілісне уявлення про роль кожного методу у балансуванні

навантаження та визначити перспективи їх застосування або вдосконалення у рамках компактної системи розподілу запитів, що розробляється.

2.1.1 Round Robin

Алгоритм Round Robin є одним із найпростіших і найпоширеніших методів розподілу запитів у системах балансування навантаження. Його основна ідея полягає у послідовному, циклічному маршрутизації запитів на кожен з доступних серверів незалежно від їх поточного стану. Після досягнення останнього вузла цикл повторюється знову, починаючи з першого сервера. Такий підхід забезпечує рівномірний розподіл запитів у середовищах, де навантаження є відносно стабільним, а сервери мають однакові ресурси та продуктивність.

Серед переваг алгоритму Round Robin можна виділити простоту реалізації та відсутність потреби в зборі складних метрик або моніторингу стану серверів. Цей алгоритм працює за принципом «чорного ящика», де кожен сервер вважається рівноправним та готовим до обробки запиту. Завдяки цьому він може використовуватися у багатьох випадках, де відсутні значні коливання навантаження або різкі відмінності у можливостях серверів.

Проте основним недоліком Round Robin є його обмежена здатність враховувати реальні умови роботи кожного вузла. Якщо сервери мають різну потужність або піддаються нерівномірному навантаженню, алгоритм може призвести до ситуацій, коли один із серверів буде перевантажений, а інші — недовантажені. У таких випадках застосування більш складних підходів, наприклад, з урахуванням активних підключень чи метрик навантаження, стає необхідним для забезпечення стабільної роботи всієї системи [4].

Загалом алгоритм Round Robin є базовим відправним пунктом для розробки систем балансування навантаження та служить основою для більш складних і адаптивних стратегій, які намагаються вирішити його обмеження та підвищити ефективність розподілу запитів у реальних умовах експлуатації. Далі розглянуто схему роботи (див. рис. 2.1).

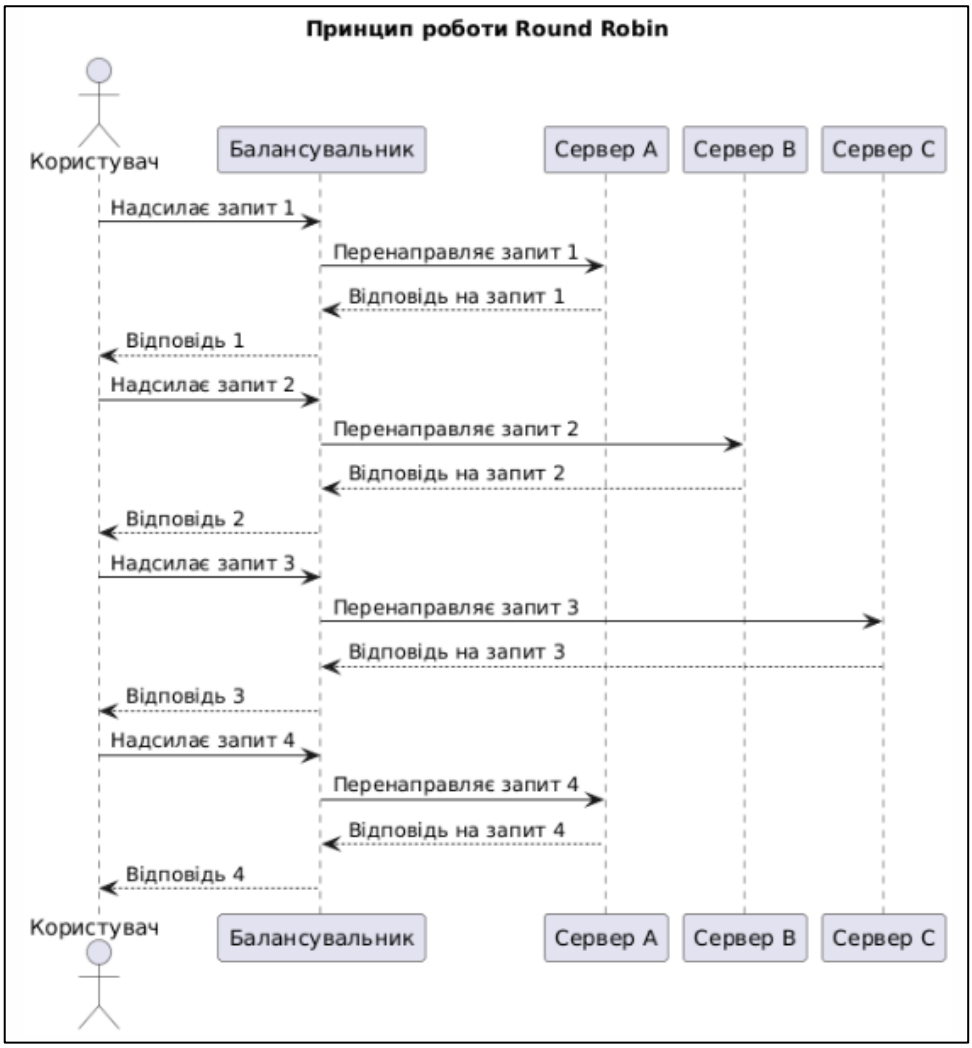

Рисунок 2.1 – Схема роботи (рисунок виконано самостійно)

Дана діаграма демонструє принцип роботи алгоритму балансування навантаження Round Robin. Вона показує послідовність обміну повідомленнями між користувачем, балансувальником і трьома серверами.

Користувач надсилає HTTP-запити, які спочатку потрапляють до балансувальника. Балансувальник виконує рівномірний розподіл, послідовно перенаправляючи запити на кожен сервер у визначеному циклічному порядку. Перший запит направляється на сервер A, другий — на сервер B, третій — на сервер C, після чого цикл повторюється, і четвертий запит знову спрямовується на сервер A.

Кожен сервер обробляє отриманий запит і повертає відповідь балансувальнику, який, у свою чергу, передає її користувачу. Таким чином, на діаграмі чітко простежується чергова зміна серверів при обробці кожного

наступного запиту, що є основною характеристикою алгоритму Round Robin. Такий підхід дозволяє уникати перевантаження одного окремого сервера і забезпечує рівномірний розподіл навантаження між усіма доступними вузлами.

Алгоритм Round Robin, попри свою простоту та широке застосування, має кілька суттєвих недоліків, які обмежують його ефективність у реальних умовах.

Головна проблема Round Robin полягає в тому, що він не враховує фактичне завантаження чи продуктивність кожного сервера. Це означає, що навіть якщо один сервер перевантажений (наприклад, обробляє складні, «важкі» запити) або має обмежені ресурси (наприклад, слабший процесор чи менше пам’яті), алгоритм все одно продовжує надсилати на нього запити у фіксованому порядку.

У випадках, коли сервери мають різну потужність або конфігурацію, Round Robin не дозволяє розподілити запити відповідно до їхніх можливостей. Потужні сервери можуть обробляти більше запитів, а менш потужні — менше, але алгоритм цього не враховує.

У сценаріях з різкими сплесками навантаження Round Robin може призводити до того, що сервери, які вже завантажені, отримуватимуть додаткові запити, не маючи можливості швидко їх обробити. Це призводить до збільшення часу відповіді або навіть відмови в обслуговуванні.

Алгоритм не передбачає автоматичного виведення з ладу серверів, які перестають відповідати або видають помилки. Якщо сервер не працює, балансувальник продовжує надсилати на нього запити, втрачаючи продуктивність і час.

У випадках, коли існують «важкі» та «легкі» запити (наприклад, генерація великих звітів проти запитів на отримання статичної інформації), Round Robin не здатен враховувати характер навантаження. Це може призводити до перевантаження серверів, що обробляють важчі запити, і простою інших [5].

У підсумку, хоча Round Robin добре працює у гомогенних середовищах зі схожими за потужністю серверами та стабільним навантаженням, у більшості реальних сценаріїв цей алгоритм поступається складнішим підходам, які враховують поточний стан кожного вузла або додаткові параметри навантаження.

2.1.2 Least Connections та Weighted Round Robin

Least Connections є більш гнучким та «інтелектуальним» підходом до балансування навантаження порівняно з базовим Round Robin. Його ключова ідея полягає у тому, що кожен новий запит надсилається на сервер із найменшою кількістю активних з'єднань у цей момент часу. Такий підхід дозволяє рівномірніше розподілити навантаження в умовах, коли час обробки запитів різний або запити мають різну «вагу».

Основні переваги цього алгоритму – здатність швидко адаптуватися до змінного навантаження та уникати перевантаження окремих вузлів, оскільки він не просто рахує запити, а фактично оцінює поточну «зайнятість» кожного сервера. Проте Least Connections потребує моніторингу поточного числа з'єднань на кожному сервері, що вимагає додаткового механізму збору метрик і може збільшувати накладні витрати в системі [6].

Алгоритм Weighted Round Robin. Weighted Round Robin – це модифікація базового Round Robin, яка додає концепцію «ваги» для кожного сервера. Кожному вузлу призначається певна вага, яка відображає його відносну потужність або продуктивність. Сервер з більшою вагою отримує пропорційно більше запитів у циклічному розподілі.

Наприклад, якщо один сервер має вдвічі більшу потужність, ніж інший, йому може бути призначена вдвічі більша вага, і він отримає відповідно більше запитів. Це дозволяє враховувати різні характеристики серверів та уникати простою більш потужних вузлів. Утім, так само як і базовий Round Robin, Weighted Round Robin не враховує реальний стан (активні з'єднання, CPU тощо) серверів у конкретний момент часу, тому при різких змінах навантаження він також може демонструвати обмеження [7]. Внаслідок цього система може приймати нераціональні рішення, перенавантажуючи вже зайняті вузли, тоді як інші залишаються менш задіяними. Це особливо критично у випадках непередбачуваного зростання кількості запитів або короткочасних піків активності, які не були враховані при початковому розподілі ваг. Далі розглянуто схему роботи (див. рис. 2.2).

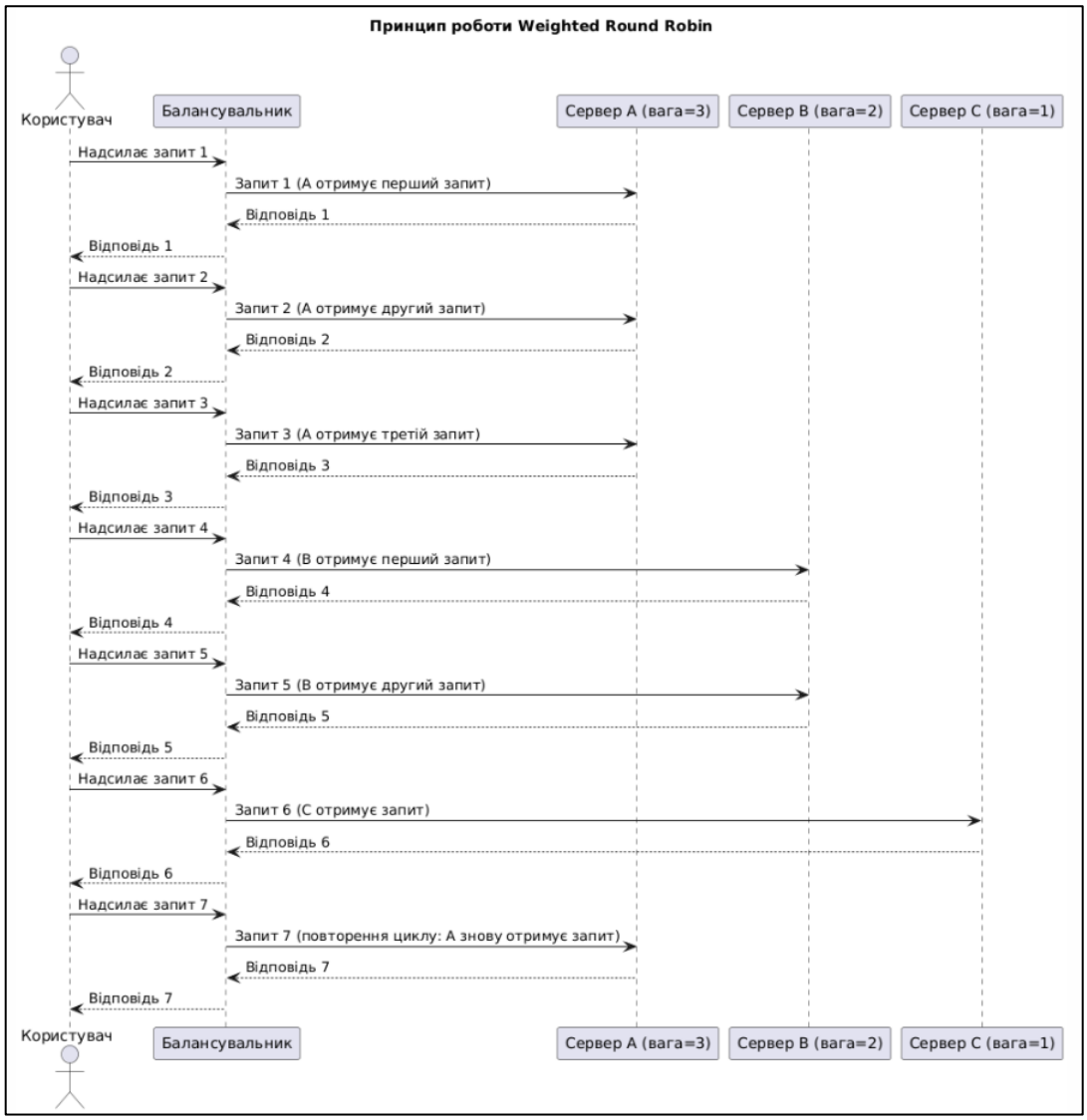

Рисунок 2.2 – Схема роботи Weighted Round Robin (рисунок виконано самостійно)

Ця діаграма ілюструє принцип роботи алгоритму Weighted Round Robin, у якому кожному серверу призначається певна вага, що відображає його відносну потужність або продуктивність. Користувач надсилає серію запитів до балансувальника, який розподіляє їх, орієнтуючись на ваги серверів.

Сервер A, що має вагу 3, отримує три запити підряд у кожному циклі. Сервер B з вагою 2 отримує два запити, а сервер C з вагою 1 — один запит. Після завершення циклу схема повторюється, знову починаючи з сервера A.

Такий підхід дозволяє враховувати різницю у потужності серверів: потужніші вузли обробляють більшу частину запитів, що забезпечує більш ефективне використання ресурсів і зменшує ризик перевантаження слабших

серверів. Однак, як і базовий алгоритм Round Robin, Weighted Round Robin не враховує реального стану серверів у поточний момент часу (наприклад, активних підключень чи завантаженості CPU), що може стати обмеженням у сценаріях з динамічними змінами навантаження.

Нижче розглянемо їх недоліки спільні та окремі.

Недоліки алгоритму Least Connections:

– не враховує продуктивність серверів: хоч цей алгоритм і обирає сервер із найменшою кількістю активних з'єднань, він не зважає на те, наскільки потужний чи швидкий кожен сервер. це може призвести до ситуацій, коли слабший сервер отримує запит, який він не в змозі ефективно обробити;
– додаткове навантаження на збір метрик: необхідно постійно моніторити активні підключення на всіх серверах, що додає накладні витрати на мережевий трафік і обчислення;
– вразливість до «важких» запитів: якщо сервер обробляє складний запит, він залишається зайнятим довше, але балансувальник цього не враховує (він бачить тільки кількість підключень, а не їх «вагу» чи тривалість обробки).

Недоліки алгоритму Weighted Round Robin:

– ігнорує поточний стан серверів: хоч вага і дає змогу краще розподіляти запити між серверами різної потужності, цей підхід не адаптується до реального завантаження. якщо сервер тимчасово перевантажений, алгоритм все одно продовжить надсилати йому запити згідно з вагою;
– фіксовані ваги: ваги задаються наперед і не змінюються під час роботи, що не дозволяє враховувати зміну навантаження чи продуктивності серверів у реальному часі;
– не враховує активні з'єднання: алгоритм працює циклічно, за схемою «згідно з вагами», але кількість підключень чи тривалість запитів залишаються поза увагою.

Спільні обмеження обох алгоритмів:

– обидва підходи демонструють обмежену здатність до адаптації в умовах, коли запити мають різну складність або сервери тимчасово втрачають працездатність;
– у сценаріях з високою динамікою трафіку або коли важливо забезпечити гнучке управління навантаженням у режимі реального часу, ці алгоритми поступаються більш адаптивним стратегіям, що враховують широкий спектр метрик (cpu, пам’ять, час відповіді, кількість помилок тощо).

2.1.3 Adaptive (Resource-Based) концептуально

Алгоритм Adaptive (Resource-Based) являє собою сучасний і більш гнучкий підхід до балансування навантаження, який враховує поточний стан ресурсів кожного серверного вузла. Його концепція базується на ідеї, що оптимальний розподіл запитів має залежати не лише від кількості вже активних з’єднань чи фіксованих ваг, а й від фактичної «здатності» кожного сервера до обробки нових запитів у конкретний момент часу.

Головною відмінністю цього методу є те, що він динамічно адаптується до реальних умов навантаження, використовуючи інформацію про такі ключові параметри, як завантаження центрального процесора (CPU), кількість активних з’єднань, споживання оперативної пам’яті, коефіцієнт помилок та інші метрики продуктивності. Універсальна ідея полягає в тому, що сервер із більшою кількістю вільних ресурсів (меншим завантаженням CPU чи меншою кількістю з’єднань) має отримати новий запит, оскільки саме він найкраще справиться з ним і забезпечить мінімальний час обробки.

Концептуально, Adaptive підхід побудований на циклічному процесі збору метрик, їхньої обробки та прийняття рішення про те, на який сервер відправити черговий запит. Для цього потрібен спеціальний механізм моніторингу — серверні агенти чи REST-ендпоїнти, які регулярно надають балансувальнику актуальні дані про стан системи. На основі цих даних балансувальник розраховує певний «індекс навантаження» або «коефіцієнт вільних ресурсів» для кожного вузла й обирає найкращий варіант для обробки нового запиту [8].

Перевагою такого підходу є його здатність швидко реагувати на коливання навантаження і мінімізувати ризики перевантаження окремих серверів, що особливо важливо у випадках різнорідного або динамічного навантаження (наприклад, коли запити відрізняються складністю або обсягом обробки). Адаптивний алгоритм дозволяє уникнути ситуацій, коли потужний сервер простоює, а слабший — перевантажений, оскільки вибір вузла заснований на реальних даних, а не лише на фіксованих налаштуваннях.

Разом із тим, Adaptive (Resource-Based) вимагає додаткових зусиль для реалізації і налаштування. Необхідно забезпечити точний і швидкий збір метрик, уникнути надмірного навантаження на систему моніторингу, а також налаштувати правила оцінки «завантаженості», що можуть бути специфічними для кожного застосунку чи сценарію використання. Важливо також збалансувати частоту опитування метрик і продуктивність балансувальника, щоб він сам не став вузьким місцем у системі.

Таким чином, концепція Adaptive (Resource-Based) втілює ідею інтелектуального балансування навантаження, що підлаштовується під реальні умови роботи. Вона відкриває шлях до більш стійкої та гнучкої архітектури, здатної ефективно працювати навіть у середовищах із високою динамікою запитів і різнорідними ресурсами серверів.

Приклади подібних алгоритмів адаптивного балансування демонструють, як різні компанії розв’язують проблему динамічного розподілу навантаження залежно від стану серверів. Наприклад:

– dynamic ratio load balancing (f5 networks) – цей алгоритм базується на оцінці продуктивності кожного серверного вузла в режимі реального часу. він враховує такі параметри, як час відповіді, використання ресурсів та кількість оброблених запитів, щоб визначити оптимальний «ваговий коефіцієнт» для кожного сервера. запити надсилаються пропорційно до цих динамічних коефіцієнтів, що дозволяє врахувати поточну продуктивність і забезпечити максимальну ефективність;

– least response time (nginx plus) – у цьому підході балансувальник обирає сервер, який продемонстрував найменший середній час відповіді за певний період. такий принцип дає змогу забезпечити максимально швидке обслуговування запитів і уникати перевантаження серверів, що тимчасово «просідають» у продуктивності;
– predictive load balancing (citrix netscaler) – цей алгоритм використовує історичні дані про навантаження та тренди в активності, щоб прогнозувати майбутнє завантаження серверів. на основі таких прогнозів він динамічно змінює правила розподілу трафіку, намагаючись мінімізувати затримки та підвищити відмовостійкість.

Зазвичай такі алгоритми інтегровані у комерційні рішення компаній-розробників і реалізовані як «чорні ящики». Це означає, що детальні принципи їхньої роботи, точні формули та методи обробки метрик залишаються закритими для публіки. Таким чином, хоча загальні ідеї (наприклад, урахування часу відповіді або динамічного стану ресурсів) добре відомі, конкретні реалізації та оптимізації, що дають конкурентні переваги, зазвичай приховуються компаніями, які їх розробляють. Нижче на рисунку 2.3 розглянуто класичну схему роботи адаптивного алгоритму.

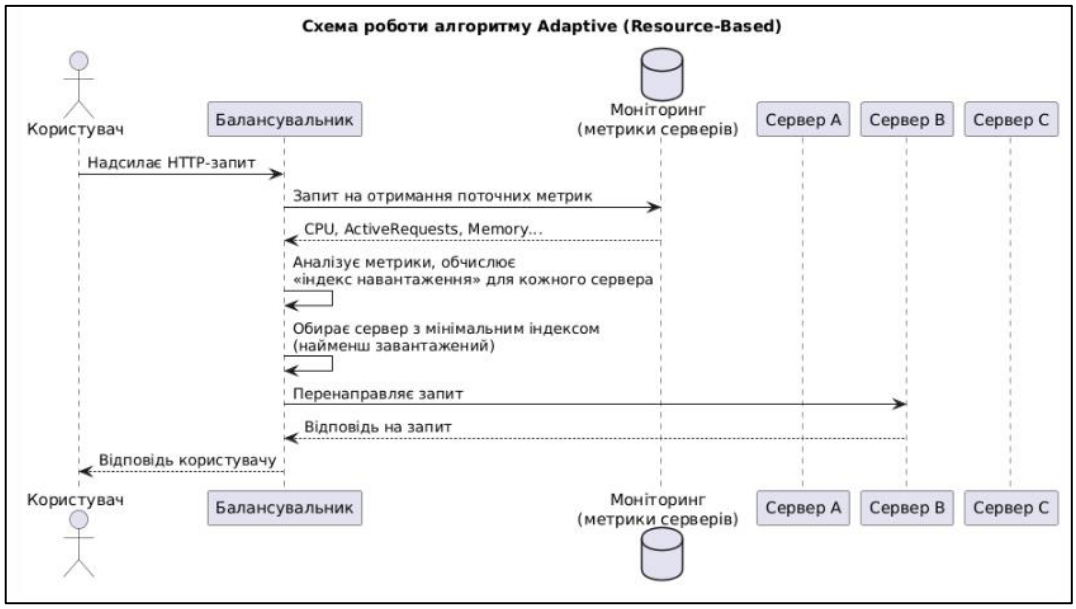

Рисунок 2.3 – Схема роботи Adaptive (Resource-Based) (рисунок виконано самостійно)

Ця діаграма ілюструє концептуальну схему роботи адаптивного (Resource-Based) алгоритму балансування навантаження. Вона демонструє ключові взаємодії між користувачем, балансувальником, компонентом моніторингу метрик та серверами.

Процес починається, коли користувач надсилає HTTP-запит до балансувальника. Щоб прийняти обґрунтоване рішення, балансувальник спочатку звертається до модуля моніторингу, запитуючи актуальні метрики стану всіх серверів, зокрема CPU-завантаження, кількість активних підключень, використання пам’яті тощо.

Отримавши ці дані, балансувальник виконує локальний аналіз та розраховує умовний «індекс навантаження» для кожного серверного вузла. Це дозволяє йому визначити, який сервер має найбільший резерв для обробки нового запиту та найменш завантажений у цей момент.

Після вибору оптимального сервера (наприклад, сервер B), балансувальник перенаправляє запит на нього. Сервер обробляє запит і повертає результат балансувальнику, який потім відправляє відповідь користувачу.

Таким чином, діаграма чітко відображає послідовність дій та ключові компоненти алгоритму Adaptive (Resource-Based), підкреслюючи його здатність до динамічного реагування на поточне навантаження та оптимального розподілу запитів.

2.2 Аналіз існуючих опен-сорс бібліотек для балансування навантаження на платформі .Net

Далі розглянемо основні відкриті (open-source) бібліотеки та проєкти на базі платформи .NET, призначені для балансування навантаження, із метою оцінити їхню придатність для тестування різних алгоритмів балансування, а також визначити їхні ключові обмеження. Важливо підкреслити, що жодна з розглянутих бібліотек у своєму поточному стані не створена як універсальний інструмент для порівняльного аналізу чи гнучкої розробки нових алгоритмів балансування: основний акцент у цих рішеннях робиться на проксінгу HTTP-трафіку та

оркестрації запитів у готових продакшен-сценаріях, а не на експериментальному тестуванні алгоритмів.

Проєкт YARP (Yet Another Reverse Proxy) – це відкритий і активно підтримуваний Microsoft (або радше спільнотою під егідою Microsoft) модуль для ASP.NET Core, який виконує роль гнучкого зворотного проксі (reverse proxy). YARP надає можливість проксувати HTTP- та HTTPS-запити від клієнта до одного або кількох backend-серверів, забезпечуючи базові механізми балансування навантаження, маршрутизації, перезапису заголовків, конфігурації політик кешування та інтеграції з middleware ASP.NET Core.

Конфігурація у YARP побудована навколо поняття “Route” і “Cluster”: кожен Route визначає набір критеріїв (шлях URI, HTTP-метод, хости, заголовки тощо), за яким запити перенаправляються у певний Cluster. Cluster, у свою чергу, містить перелік backend-endpoint’ів (URI серверів) та політики балансування. Політики балансування [9]. У коробковій конфігурації YARP підтримує прості методи:

– round robin (за замовчуванням);
– no load balancing (проксірування на перший доступний endpoint);
– least requests (аналог least connections, але реалізований через підрахунок активних запитів);
– power of two random choices (випадковий вибір між двома серверними endpoint, мінімізуючи активні підключення).

Всі політики задаються у форматі YAML/JSON (appsettings.json), що дає змогу розробникам визначати набори правил, змінювати їх під час роботи програми без перекомпіляції, а також плагінувати власні реалізації інтерфейсів (наприклад, ILoadBalancingPolicy, IClusterManager).

Взаємодія з pipeline ASP.NET Core дозволяє легко під’єднувати аутентифікацію, авторизацію, логування, метрики Prometheus/Grafana тощо.

Фокус на продакшен-проксі. YARP спроектований першочергово як високопродуктивний зворотний проксі для робочих середовищ (production), а не як бібліотека для дослідження різних стратегій розподілу. Основні модулі й

інтерфейси все ж орієнтовані на інтеграцію в ASP.NET Core-проєкт, що ускладнює ізольоване тестування алгоритму поза контекстом HTTP pipeline.

Обмежений набір вбудованих політик. Хоча YARP допускає розширення та створення власних ILoadBalancingPolicy, у коробковій версії доступні лише кілька базових стратегій (Round Robin, Least Requests, Power of Two Random Choices та No Load Balancing). Для дослідження інших алгоритмів необхідно реалізувати власні політики, а при цьому доводиться глибоко занурюватися в архітектуру YARP.

Використання актуальних метрик у реальному часі. Бібліотека підтримує підрахунок активних запитів (для політики Least Requests), але не надає готових компонентів для збирання детальніших даних «з нуля» (CPU, пам’ять, network I/O). Щоби застосувати підхід Resource-Based (Adaptive), потрібно інтегрувати власний модуль моніторингу метрик і реалізувати поліпшену політику вручну.

Відсутність зручного механізму емуляції. YARP не включає утиліт або шаблонів для генерації імітованого навантаження (наприклад, з K6) всередині проєкту — ця частина завжди реалізується окремо, що підвищує складність налаштування дослідного середовища.

З огляду на ці обмеження, хоча YARP і є динамічним та розширюваним рішенням для продакшен-проксі на ASP.NET Core, його використання для швидкого «прототипування» та порівняльного аналізу різних алгоритмів балансування є досить громіздким та вимагає значного обсягу власного коду.

2.2.1 Ocelot

Ocelot – це відкрите рішення у вигляді API-Gateway для .NET-екосистеми. Основна мета Ocelot – забезпечити єдину точку входу (Single Entry Point) для клієнтів та розподілити запити між низкою мікросервісів або бекенд-серверів. Ocelot здебільшого використовується у архітектурі мікросервісів як API-шел, що об’єднує маршрутизацію, авторизацію, аутентифікацію, агрегацію результатів та базове балансування.

До основних можливостей можна виділити роутинг та перезапис URL. Ocelot дозволяє детально налаштовувати правила маршрутизації, включаючи переписування шляхів, заголовків, параметрів запиту тощо.

Підтримка кількох схем аутентифікації/авторизації. Є вбудовані сервіси для JWT, OAuth2, IdentityServer4 та інші middleware, що спрощують захист API-шлюзу [10].

Кешування відповідей (Response Caching). Можливість налаштувати кешування за таймаутами, ключами та умовами, що зменшує навантаження на бекенди.

Rate Limiting (обмеження кількості запитів). Забезпечує механізми обмеження частоти запитів від одного IP або за ключем API.

Ocelot підтримує два основних методи балансування:

– round robin (рівномірний розподіл у циклі);
– least connection (вибір сервера з найменшою кількістю активних підключень).

Обмеження Ocelot як інструмента для досліджень – архітектурна прив’язка до API-Gateway. Ocelot розроблено як готовий рішення для ролі API-Gateway, що поєднує кілька функцій: маршрутизацію, автентифікацію, кешування, моніторинг та балансування. Якщо мета полягає виключно у тестуванні алгоритмів балансування, проєкт доведеться розділяти на окремі компоненти, або розгортати повноцінний Ocelot-Gateway із зайвими функціями. Це ускладнює дослідні середовища та підвищує поріг входу.

Обмежений набір політик. У коробковому пакеті Ocelot доступні лише Round Robin та Least Connection. Немає базової підтримки для Weighted Round Robin, Adaptive-стратегій або будь-яких кастомних «вагових» налаштувань. Хоча можна реалізувати свою логіку через розширення та middleware, це вимагає суттєвих зусиль.

Відсутність «пульту» для тонкої конфігурації алгоритмів. Основні налаштування задаються у файлі ocelot.json, і це зручно для базового продакшен-конфігурування. Проте для наукових експериментів потрібна можливість

програмного контролю алгоритмів у часі виконання, що вимагає змінити код ядра Ocelot або створювати власні обгортки над його класами.

Немає вбудованих засобів для емуляції навантаження та збирання детальних метрик. Як і у випадку з YARP, Ocelot не надає готової платформи для одночасного генерування трафіку і збору метрик про навантаження – такі механізми потрібно додавати окремо.

Таким чином, хоча Ocelot є надійним API-Gateway із базовими можливостями балансування, для цілей дослідження алгоритмів балансування запитів він вимагає значної доопрацювання або створення окремого шару, що ускладнює оцінювання різних стратегій.

2.2.2 Результати аналізу бібліотек

Проведений огляд показує, що на сьогодні в екосистемі .NET існує кілька якісних open-source рішень для балансування HTTP-навантаження, серед яких YARP та Ocelot є найпоширенішими та підтримуваними спільнотою. Водночас кожне з цих рішень має суттєві обмеження, що знижують їхню придатність для використання в дослідницьких цілях, а саме:

Більшість бібліотек спроектовано як готові промислові (production) рішення. Основний акцент зроблено на надійності проксінгу, безпеці (authN/authZ), кешуванні та швидкості інтегруватись у сервіс-mesh. Проте немає «легкого» способу швидко ітеративно змінювати алгоритми балансування, експортувати результати у зручному вигляді чи порівнювати продуктивність різних стратегій.

У коробкових версіях здебільшого доступні тільки базові стратегії (Round Robin, Least Connections, інколи Weighted Round Robin у примітивному вигляді). Алгоритми, що враховують ресурсне навантаження (Adaptive, Resource-Based), як правило, відсутні або вимагають власної реалізації через спеціальні інтерфейси. Це ускладнює експерименти з більш гнучкими чи власними стратегіями балансування.

Жодна з розглянутих бібліотек не містить готових модулів для емуляції навантаження, збирання статистики (наприклад, розподіл запитів за часом, середній latency для кожної стратегії, відсоток помилок тощо) чи для інтеграції з

K6/Locust/Gatling. Для побудови середовища тестування доводиться або адаптувати сам код бібліотек, або розгортати окремий набір сервісів (моніторинг, логування, генератор трафіку), що значно підвищує складність і ресурсоємність дослідного процесу.

Деякі бібліотеки (наприклад, менш відомі Stormgate чи DotNetLoadBalancer) не оновлюються тривалий час і мають фрагментарні README або застарілі приклади. Інтерфейси цих рішень часто не відповідають актуальним версіям .NET, що унеможливлює їх безболісне використання у нових проєктах.

Навіть у найбільш розвинених продуктах (YARP, Ocelot) немає готових компонентів, що б у реальному часі зчитували CPU, пам’ять або мережеві метрики з серверів і використовували їх у ядрах алгоритмів. Щоби реалізувати справді адаптивний Resource-Based підхід, доведеться додавати власні агенти моніторингу та реалізовувати відповідні інтерфейси.

У підсумку, існуючі open-source рішення для систем балансування на платформі .NET мають значний функціонал і здатні задовольнити більшість завдань у продакшен-середовищах, але зазвичай не розраховані на задачі дослідження та тестування алгоритмів балансування. Усі ці бібліотеки призначені першочергово для стабільного розгортання проксі/маршрутизуючих шлюзів, коли алгоритми балансування є швидко конфігуруваними, але рідко доступними для глибокої модифікації. Отже, виникає потреба у спеціалізованому опен-сорс рішенні, яке б поєднувало те, що наразі представлене лише частково: можливість гнучко зазначати вагові коефіцієнти, екстремуми, порогові значення, а також зручні засоби для аналізу результатів різних стратегій у єдиному уніфікованому середовищі.

2.3 Методика проведення дослідження

Після детального аналізу класичних та адаптивних алгоритмів балансування стає очевидним, що жоден із них не є універсальним і повністю придатним для всіх можливих сценаріїв навантаження та архітектурних рішень. Кожен підхід має свої переваги, але також демонструє суттєві обмеження:

Round Robin і Weighted Round Robin мають просту реалізацію та гарно працюють за умови однорідних серверів, однак не враховують реальний стан вузлів.

Least Connections підвищує чутливість до «зайнятості» серверів, але ігнорує інші фактори продуктивності, такі як CPU чи пам’ять.

Adaptive алгоритми гнучкіші, проте зазвичай «зашиті» у пропрієтарні рішення та не дають змоги легко їх налаштувати під конкретні потреби.

Це підкреслює потребу у створенні відкритого (опенсорсного) універсального алгоритму балансування, який би об’єднував переваги кожного з перелічених підходів та надавав можливість гнучко налаштовувати ключові параметри. Такий універсальний алгоритм повинен:

Підтримувати вагові коефіцієнти – для урахування різниці у потужності серверів та специфіки виконуваних завдань.

Враховувати екстремуми – уміти відстежувати ситуації, коли один із серверів наближається до критичного стану, щоб вчасно виводити його з пулу або зменшувати потік запитів на нього.

Оперувати пороговими значеннями – дозволяти задавати гнучкі правила (наприклад, CPU > 90% – сервер «непридатний» для прийняття нових запитів; або ActiveRequests > 50 – перенаправлення трафіку до менш завантажених вузлів).

Важливо, щоб така система була відкритою для спільноти розробників: це дозволить уникнути залежності від пропрієтарних рішень великих компаній і дасть змогу адаптувати алгоритм під конкретні умови експлуатації. Окрім того, прозорість коду полегшить аудит безпеки, налаштування та інтеграцію з існуючими архітектурами. У підсумку, універсальний опенсорс-алгоритм із можливістю враховувати вагові коефіцієнти, екстремуми й порогові значення забезпечить ефективність, гнучкість і відмовостійкість розподілу запитів у найрізноманітніших IT-ландшафтах.

2.3.1 Загальна методика

Методика дослідження базується на розробці єдиної .NET-платформи, здатної одночасно виконувати роль як тестового середовища для алгоритмів балансування, так і реального балансувальника запитів у мережевому житті додатку. Уявимо собі, що наша система складається з кількох взаємопов’язаних компонентів, кожен із яких виконує свою задачу, але весь потік координується центральним модулем.

Початковим етапом є створення набору «імітаційних» серверів, що моделюють реальні вузли з різними навантажувальними профілями. Кожен із цих серверів періодично (раз на кілька секунд) передає власні метрики — наприклад, рівень завантаження процесора, кількість активних з’єднань та загальну кількість оброблених запитів. Ці дані надходять у центральний сервіс, де акумулюються у внутрішніх структурах. З одного боку, вони підживлюють алгоритми, що базуються на ресурсних показниках вузлів, а з іншого – дають можливість відстежити зміну стану мережі протягом кожного циклу тестування.

Коли користувач (або емулятор навантаження на кшталт K6) надсилає перший HTTP-запит на наш балансувальник, система опрацьовує його у такий спосіб: спочатку зчитуються найсвіжіші метрики з кожного імітованого сервера, далі відповідно до вибраної стратегії балансування обирається саме той вузол, який зможе обробити запит ефективніше. У випадку класичних алгоритмів (Round Robin чи Weighted Round Robin) вибір відбувається внаслідок фіксованого циклу або з урахуванням «вагових коефіцієнтів», які ми задаємо для кожного вузла заздалегідь. У разі Least Connections рішення ґрунтується на кількості активних з’єднань, тобто перевага віддається тому серверу, у якого найменше одночасних оброблюваних запитів. І нарешті, у випадку Adaptive (Resource-Based) алгоритму враховуються безпосередньо ресурси: початково обчислюється умовний «індекс вільних ресурсів» для кожного вузла на основі CPU-завантаження, числа активних з’єднань та інших показників. Саме цей індекс визначає, куди відправляти запит, щоб мінімізувати затримки й уникнути перевантаження.

Наша система працює у режимі безперервних ітерацій: кожен новий запит обробляється з урахуванням максимально актуального стану серверів. Це дає змогу спостерігати за тим, як змінюється розподіл навантаження під час пікових сплесків, різких стрибків у CPU чи зростання кількості одночасних з’єднань. Результати кожного циклу зберігаються й у підсумковому звіті можуть бути зведені до графіків, таблиць або логів, де видно, скільки запитів дісталося кожному серверу, якою була середня затримка, скільки запитів завершилися з помилкою тощо.

Поряд із класичними підходами одним із завдань є розробка нового, універсального адаптивного алгоритму. Цей алгоритм поєднує кілька ідей одночасно: він підтримує вагові коефіцієнти, що визначають базовий розподіл для гомогенних або умовно рівних вузлів, але водночас контролює порогові значення по кожному серверу (наприклад, якщо CPU перевищив 80 % або кількість активних з’єднань наближається до критичної межі, такий вузол тимчасово виводиться з пулу). У той самий час він фіксує екстремальні стани – наприклад, якщо сервер перебуває у стані «перевантаження» або демонструє підвищений коефіцієнт помилок, – і автоматично змінює вагові налаштування, перерозподіляючи трафік на інші вузли. Таким чином, наш універсальний алгоритм гнучко поєднує базове «вагове» правило, оцінювання ресурсних навантажень і відстеження порогових станів у режимі реального часу.

Ключовою відмінністю нашої розробки є те, що система не лише тестуватиме різні підходи, а й буде готова працювати у «бойовому» режимі, реально балансуючи запити в продакшин-сценаріях. Умовно кажучи, її можна інтегрувати в робочий кластер .NET-серверів та переконатися, що вибрана стратегія балансування дійсно забезпечує оптимальну відповідь користувачам. Такий підхід дозволяє одночасно виконувати два завдання: науково-аналітичне (порівняння алгоритмів у контрольованому середовищі) та практичне (забезпечення рівномірного розподілу трафіку в реальних умовах). Оскільки всі компоненти побудовані на базі ASP.NET Core і .NET 6+, переваги платформи – висока швидкодія, зручність у налаштуванні middleware, багата екосистема бібліотек моніторингу та логування – використовуються в повному обсязі. Система

орієнтована на широку гнучкість у застосуванні та здатна адаптуватися до різних потреб користувача. Її можна використовувати як у навчальних цілях, так і в межах реальних проєктів, де важливі стабільність та ефективність обробки запитів. Рішення легко масштабувати під різні сценарії та умови експлуатації. Простота налаштування та зрозумілий підхід до архітектури роблять цю систему зручною навіть для розробників без глибокої спеціалізації у темі балансування. Вона може стати основою для подальших досліджень, удосконалень або промислового використання. Такий універсальний характер платформи забезпечує її актуальність у різних сферах застосування. У результаті ми отримуємо універсальний інструмент, який дозволяє як швидко змінювати й перевіряти алгоритми балансування під різні навантаження, так і застосовувати обраний алгоритм безпосередньо для розподілу живого трафіку у виробничому середовищі.

3 ПРОЕКТУВАННЯ СИСТЕМИ ТА РОЗРОБКА АЛГОРИТМУ

3.1 Проектування архітектури системи

Наша система призначена для дослідження та порівняння різних алгоритмів балансування навантаження in situ, із використанням змодельованих показників трьох серверів. Вона складається з трьох основних частин. По-перше, це емуляція серверів (Server1, Server2, Server3). Кожен із цих “серверів” запускається як окремий ASP.NET-проєкт, що працює на власному порті (5001, 5002 та 5003). Усередині кожного проєкту є клас ServerMetricsService, який кожні п’ять секунд подає заздалегідь заданий набір значень метрик (CpuUsage, ActiveRequests, AvgResponseTime, FailureRate тощо). За допомогою таких імітованих патернів ми можемо відтворити сервери різної потужності та різної завантаженості: наприклад, один із них демонструє постійне низьке навантаження, інший—працює майже на межі своїх ресурсів, а третій періодично переживає піки та спади. Контролер StatusController повертає поточний стан сервера у вигляді JSON, який містить поля cpuUsage, activeRequests, avgResponseTime, failureRate, cpuCores, availableRAM. Таким чином, наша “бекенд-емуляція” постійно оновлює набір метрик, котрі потім зчитуються основною частиною системи.

По-друге, основний проєкт DistLoad (Load Balancer & Middleware) підтримує повний цикл роботи з цими метриками та реалізовує логіку вибору сервера для кожного запиту. Усередині DistLoad ми реєструємо колекцію ServerInstance — об’єктів, що містять Id, Address, CpuUsage, ActiveRequests, RequestCount, LastState та інші внутрішні поля. Також у Program.cs реєструється сервіс LoadBalancerManager, який утримує поточний алгоритм розподілу (round-robin, least-connections, adaptive, log-mix-adaptive тощо), а також сервіс MetricsLogger, який у фоновому циклі кожні п’ять секунд опитує кожний із трьох серверів через їхні ендпоїнти /api/status, оновлює поля відповідних ServerInstance, а також оновлює Prometheus-метрики. Завдяки цьому балансувальник завжди працює з максимально актуальною інформацією.

Коли надходить HTTP-запит (будь то від браузера або від інструмента навантаження), наш кастомний LoadBalancerMiddleware перехоплює його, якщо це

не запит до /swagger чи /metrics. У момент обробки запиту middleware викликає LoadBalancerManager.GetNextServerAsync(), який обирає найкращий сервер згідно з поточними метриками й активним алгоритмом. Після вибору ServerInstance з найменшими затримками чи найнижчим навантаженням відбувається інкремент лічильника в ServerCounters, аби зафіксувати, що цей сервер отримав ще один запит. Поки що ми не робимо фактичного HTTP-проксіювання до бекенд-серверів, а просто передаємо контекст далі (_next(context)), щоб можна було сфокусуватися виключно на логіці розподілу без ускладнень мережевого рівня.

По-третє, підсистема збору та відображення метрик реалізована через клас MetricsLogger. У конструкторі цей компонент запускає фоновий таск, який у методі Loop() здійснює HTTP GET до кожного з адрес http://localhost:5001/api/status, http://localhost:5002/api/status і http://localhost:5003/api/status, десеріалізує отриманий JSON у об’єкт ServerState і під час блокування (lock) оновлює відповідні Prometheus-метрики із міткою server_id, наприклад:

```
CpuUsageGauge.WithLabels(s.Id).Set(st.CpuUsage);
ActiveRequestsGauge.WithLabels(s.Id).Set(st.ActiveRequests);
TotalRequestsCounter.WithLabels(s.Id).Inc();
```

Ці самі значення записуються й у поля ServerInstance.CpuUsage і ServerInstance.ActiveRequests, щоб LoadBalancerManager бачив найбільш свіжі дані. Одночасно у консолі виводяться два кольорові блоки: зелений, де показані поточні метрики кожного сервера, і жовтий, де зафіксовано лічильники запитів, які надійшли на кожен з них. Варто зазначити, що ми відкриваємо Prometheus-endpoint через app.UseMetricServer(), за замовчуванням доступний за адресою http://localhost:5149/metrics, тож будь-який Prometheus-сервер може підключатися до цього ендпоїнту і забирати наші Gauge/Counter із відповідними лейблами server_id.

Для генерації навантаження у дослідному середовищі використовується інструмент k6. Ми запускаємо k6 із готовим JS-скриптом, який визначає фази ramp-up, plateau та ramp-down і надсилає сталий потік HTTP-GET запитів до адреси нашого балансувальника (наприклад, до http://localhost:5149/ або будь-якого іншого ендпоїнту, що перехоплюється LoadBalancerMiddleware). Кожен такий

запит фіксується у LoadBalancerMiddleware, лічильники відповідних серверів збільшуються, але сам запит пропускається далі, повертаючи 200 OK. Таким чином, ми імітуємо реальний трафік, не переводячи запити справді на бекенд, але водночас можемо чітко відстежувати, як змінюється розподіл у кожну п’ятисекундну порцію нової інформації про метрики.

Попри те, що система насамперед слугує дослідницьким стендом, вона також здатна виконувати роль реального балансувальника запитів. Якщо необхідно, можна дописати справжню проксінгову логіку замість await _next(context), і тоді кожен запит від клієнта буде перенаправлений на вибраний сервер, що дозволить використовувати DistLoad у продакшені.

У рамках цього проєкту ми ставимо такі завдання: по-перше, реалізувати і порівняти класичні алгоритми (Round-Robin, Weighted-Round-Robin, Least-Connections, Weighted-Least-Connections) та адаптивні підходи (Adaptive, Log-Mix-Adaptive); по-друге, створити власний «універсальний адаптивний» алгоритм, який поєднуватиме вагові коефіцієнти, порогові значення (наприклад, обмеження за CPU та ActiveRequests) і реагування на екстремальні стани; по-третє, забезпечити можливість запускати навантажувальні тести із зовнішнього інструменту (k6), фіксувати результати у вигляді Prometheus-метрик та виводити їх у консоль кожні п’ять секунд; нарешті, побудувати єдину панель керування (наприклад, через API POST /api/loadbalancer/algorithm/{algorithm}), яка дозволить динамічно перемикати алгоритми балансування під час тесту чи в реальному світі. Особливу увагу приділено гнучкості системи та її здатності адаптуватися до змін у навантаженні без перезапуску сервісу. Для забезпечення узгодженої взаємодії між модулями необхідно було чітко визначити порядок обміну даними, циклічність оновлення метрик та принципи обробки запитів [11]. Такий підхід дозволяє досягти високого ступеня контрольованості процесу та швидко реагувати на непередбачувані зміни в поведінці серверів. Реалізовані інструменти повинні не тільки забезпечувати функціональність, але й бути зручними для розширення в майбутньому[12]. На рисунку 3.1 зображено схему роботи системи.

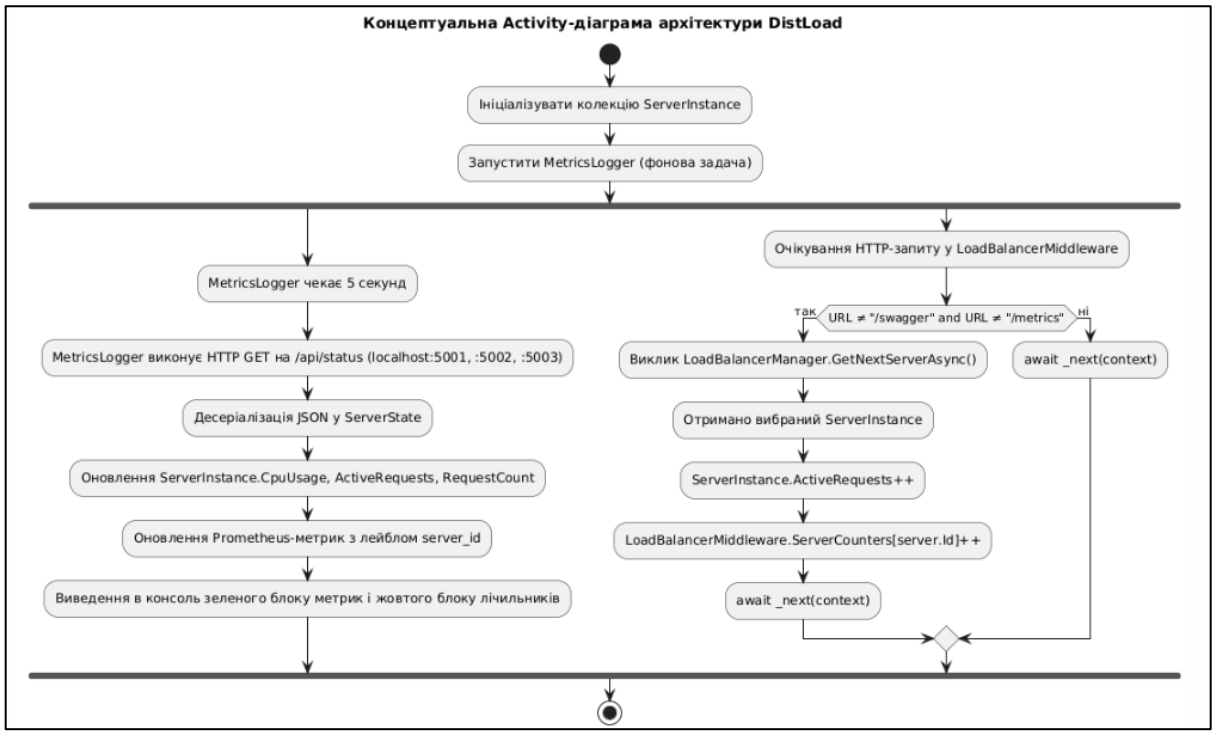

Рисунок 3.1 – Базовий пайплайн виконання потоку (рисунок виконано самостійно)

Отже, наша система є гнучким і розширюваним ASP .NET-рішенням, яке одночасно емулює сервери з фіксованими патернами метрик, збирає їхні дані через MetricsLogger, на основі цих даних приймає рішення через LoadBalancerManager (із обраним алгоритмом), веде лічильники запитів і відображає результати кожні 5 секунд. Зовнішній генератор навантаження k6 формує реальний трафік, а DistLoad “лічить”, як саме різні алгоритми розподіляють це навантаження. Така структура дає змогу досліджувати й калібрувати алгоритми в реальному часі, а також за необхідності застосовувати ці самі алгоритми для справжнього балансування живого трафіку.

3.2 Файлова структура

У корені рішення «DistLoad» знаходяться чотири проєкти: основний проєкт DistLoad та три емуляційних сервери Server1, Server2 і Server3. У проєкті DistLoad є папка Controllers, у якій містяться два контролери: LoadBalancerController.cs, що обробляє вхідні HTTP-запити та делегує їх до алгоритму балансування, і MetricsController.cs, що повертає накопичені метрики в форматі JSON.

Нижче на рисунку 3.2 наведено файлову структуру системи.

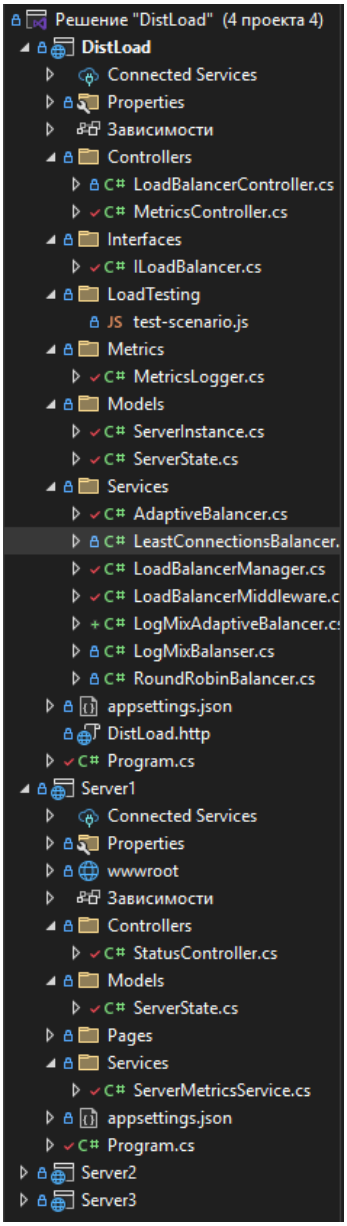

Рисунок 3.2 – Файлова структура системи (рисунок виконано самостійно)

У папці Interfaces розміщено інтерфейс ILoadBalancer.cs, який визначає контракт для реалізації алгоритмів балансування. Папка LoadTesting містить скрипт test-scenario.js, що використовується інструментом k6 для емуляції навантаження. У папці Metrics знаходиться MetricsLogger.cs, який у фоновому режимі опитує усі сервери та оновлює дані метрик і Prometheus-лічильників. Папка Models містить два класи: ServerInstance.cs, що описує стан кожного сервера в колекції, і ServerState.cs, який відображає структуру даних, що повертає

StatusController. Папка Services містить реалізації алгоритмів балансування та суміжних компонентів: AdaptiveBalancer.cs і LogMixAdaptiveBalancer.cs для адаптивних балансувальників, LeastConnectionsBalancer.cs для алгоритму «найменша кількість з’єднань», RoundRobinBalancer.cs і LogMixBalancer.cs для «кругової» та «логарифмічно-зваженої» реалізації, LoadBalancerManager.cs, який координує вибір алгоритму, та LoadBalancerMiddleware.cs, що перехоплює запити і викликає LoadBalancerManager.

У корені проєкту DistLoad також присутні файли appsettings.json для конфігурації, DistLoad.http із прикладами запитів і Program.cs, у якому налаштовується вся інфраструктура (реєстрація серверів, запуск MetricsLogger, налаштування middleware тощо). Кожен із проєктів Server1, Server2 та Server3 має подібну внутрішню структуру: папку Controllers із файлом StatusController.cs, який повертає поточний ServerState, папку Models із класом ServerState.cs для представлення метрик, папку Services із ServerMetricsService.cs, що кожні п’ять секунд генерує новий набір значень метрик, а також файл Program.cs та appsettings.json, що задають конфігурацію та порт запуску відповідного емуляційного сервера.

3.3 Розробка універсальних адаптивних алгоритмів

На основі аналізу попередніх алгоритмів та їх недоліків та особливостей – було розроблено власний адаптивний алгоритм LogMixAdaptiveBalancer.

У попередніх розділах ми розглянули класичні стратегії балансування («кругова робін», «зважений кругова робін», «найменша кількість з’єднань») та суто ресурсно-орієнтовані підходи, які враховують лише одну–дві змінні (наприклад, число активних підключень або завантаження CPU). Однак наш аналіз виявив, що будь-який із цих методів має слабкі сторони, коли йдеться про динамічні та неоднорідні середовища, в яких одночасно діють такі чинники:

– поточне завантаження cpu та кількість активних запитів (це прямі індикатори того, наскільки «зайнятий» сервер у кожен момент): але, якщо покладатися лише на ці дані, алгоритм може ігнорувати те, що сервер

учора чи за останню годину багато помилявся або мав дуже довгий «середній час відповіді»;

– історичні метрики (даталога): загальна кількість оброблених запитів (totalreq) як індикатор «стажу» та стабільності, середній час відповіді за останній інтервал (resptime), частота помилок (failurerate), велике «минуле навантаження» або, навпаки, постійна стабільність. жоден класичний метод (round robin, least connections) не уникає ситуацій, коли сервіс, що раніше «ламається», отримує жирну порцію нових запитів лише тому, що він «злегка менш зайнятий» тут і зараз;
– різні технічні характеристики серверів: кількість логічних ядер cpu, обсяг вільної пам’яті – ці показники визначають «максимально можливу» продуктивність вузла. weighted round robin дозволяє частково врахувати різницю потужностей, але фіксовані ваги (заданні на початку) не реагують на те, що сервер раптово «заростає» іншими процесами чи знижує своє свободне місце у пам’яті.

Урахувавши всі ці обмеження та бажаючи створити дійсно «універсальний» адаптивний алгоритм, ми сформулювали двокомпонентну модель, яка складається з:

– оцінки «ефективності» (score) кожного сервера у поточний момент, що базується на чотирьох ключових показниках;
– activerequests – кількість активних запитів у цей момент (поточне навантаження);
– cpuusage – відсоток завантаження cpu (0…100 %);
– totalreq – загальна кількість запитів за поточну сесію (історична метрика, що вказує на стабільність і «продакшен-стаж» сервера);
– resptime – середній час відповіді (ms) за останній інтервал.

Формула для обчислення Score була обрана так, щоб у чисельнику знаходилися чотири логарифмічні складові, а у знаменнику — ваговий коефіцієнт (Weight) плюс невелика константа ε. Сам вигляд функції (3.1):

$$Score(S) = \frac{Log(ActiveReq + 1) + Log(CpuUsa + 1) + Log(TotalReq + 1) + Log(RespType + 1)}{\text{Weight} + \text{e}} \quad (3.1)$$

де activerequests i — поточна кількість активних запитів на і-му сервері (поточний load);

cpuusage i - відсоток завантаженості cpu (0 ... 100%) у і-го сервера;

totalreq i — загальна кількість запитів, оброблених цим сервером за сеанс (історична метрика – що більше, тим стабільніше працює сервер);

resptime i — середній час відповіді (ms) сервера за останній інтервал;

ε — невелика постійна (наприклад, 0,001), щоб знаменнику ніколи не вийшло точне «0».

Чим більший Score, тим «гіршим» вважається сервер у поточний момент, тобто трафік на нього повинен надходити рідше (або взагалі припинятися), адже зростання будь-якої складової чисельника означає: або активних запитів багато, або CPU завантажений, або сервер був дуже «активний» в історичному сенсі (що іноді свідчить про високу задіяність), або ж сам сервер має великий середній час відповіді.

Класичні алгоритми «Round Robin» і «Weighted Round Robin» не враховують жодної інформації про поточну або історичну продуктивність серверів, а «Least Connections» реагує лише на кількість активних підключень, не звертаючи уваги на інші важливі параметри. Багато сучасних адаптивних (ресурсно-орієнтованих) рішень у комерційних продуктах часто залишаються «чорними ящиками» — їхня внутрішня логіка не розкривається, а врахування обмежується здебільшого тільки поточним завантаженням CPU.

Натомість наша формула дає змогу одночасно оцінювати як state variables (поточні метрики: ActiveRequests, CpuUsage, TotalReq, RespTime), так і history variables (історичні тренди: AvgResponseTime, FailureRate), застосовуючи логарифмічні перетворення, які пом’якшують вплив раптових «вибухів» значень без надмірної чутливості. Вона забезпечує динамічну корекцію ваги кожного вузла залежно від його реальної стабільності та здатності обробляти навантаження, і працює в режимі «усі показники в одному місці», що суттєво спрощує

конфігурацію: достатньо задати інтервали опитування, значення ε та джерела ресурсних даних, і система сама «зважує» всі важливі фактори.

У підсумку розробка універсального адаптивного алгоритму на основі цих формул поєднує переваги класичних і ресурсно-орієнтованих підходів, усуває їхні недоліки та досягає стабільного, збалансованого й гнучкого розподілу навантаження в середовищах із динамічною зміною параметрів серверів, що забезпечує оптимальне використання ресурсів, підвищує відмовостійкість та дає змогу адаптуватися до будь-яких нетипових або екстремальних сценаріїв у межах єдиного механізму.

Нижче наведено фрагмент коду що описує концептуально та реалізує нашу формулу.

```
using DistLoad.Interfaces;
using DistLoad.Models;
using System;
using System.Collections.Generic;
using System.Linq;
using System.Threading.Tasks;

namespace DistLoad.Services
{
    public class LogMixAdaptiveBalancer : ILoadBalancer
    {
        private readonly List<ServerInstance> _servers;
        private const double Epsilon = 0.001;

        public LogMixAdaptiveBalancer(List<ServerInstance> servers)
        {
            _servers = servers;
        }
        public Task<ServerInstance> GetNextServerAsync()
        {
            var candidates = _servers
                .Where(s => s.IsOnline && !IsOverloaded(s))
                .ToList();

            if (!candidates.Any())
                throw new Exception("No available servers");
            ServerInstance best = null;
            double bestScore = double.MaxValue;

            foreach (var s in candidates)
            {
                double activeReq = s.ActiveRequests;
                double cpuUsage = s.CpuUsage;
                double totalReq = s.RequestCount;
                double respTime = s.LastState?.ResponseTime ?? 0.0;
                double failureRate = s.LastState?.FailureRate ?? 0.0;
                double cores = s.CpuCores; /*CpuCores ?? 1.0;*/
```

```
                    double ram = s.AvailableRAM;/*AvailableRAM ?? 1.0;*/
                    double avgResp = s.LastState?.ResponseTime ?? 0.0;
                    double   denom   =   Math.Log(avgResp   +   1.0)   +
Math.Log(failureRate + 1.0);
                    if (denom <= 0) denom = Epsilon;

                    double weight = (cores * ram) / denom;
                    double logActive = Math.Log(activeReq + 1.0);
                    double logCpu = Math.Log(cpuUsage + 1.0);
                    double logTotal = Math.Log(totalReq + 1.0);
                    double logResp = Math.Log(respTime + 1.0);
                    double numeratorScore = logActive + logCpu + logTotal
+ logResp;
                    double score = numeratorScore / (weight + Epsilon);
                    if (score < bestScore)
                    {
                        bestScore = score;
                        best = s;
                    }
                }
                if (best == null)
                    throw new Exception("No available servers");

                best.ActiveRequests++;
                return Task.FromResult(best);
            }
            private bool IsOverloaded(ServerInstance s)
            {
                if (s.CpuUsage >= s.CpuCriticalThreshold) return true;
                if (s.ActiveRequests > s.MaxActiveRequests) return true;
                return false;
            }
            public List<ServerInstance> GetServers() => _servers;
        }
    }
```

У класі LogMixAdaptiveBalancer реалізовано універсальний адаптивний алгоритм вибору сервера на основі комбінованої оцінки поточного та історичного стану кожного вузла. Конструктор приймає список об’єктів ServerInstance, кожен із яких містить властивості: поточна кількість активних запитів ActiveRequests, відсоток завантаження CPU CpuUsage, загальна кількість оброблених запитів RequestCount, історичні дані в LastState (середній час відповіді ResponseTime та частота збоїв FailureRate), а також апаратні характеристики CpuCores та AvailableRAM. Метод GetNextServerAsync спочатку відфільтровує тих серверів, які онлайн (IsOnline) і не перевантажені (метод IsOverloaded перевіряє, чи не перевищує CPU критичний поріг або чи не зростає число активних запитів понад допустимий максимум). Далі для кожного кандидата розраховується «вага» weight за формулою (cores × ram) ÷ (log(AvgResponseTime + 1) + log(FailureRate + 1)), де,

якщо знаменник виходить нульовим або меншим, ставиться мале ε = 0.001, щоб уникнути ділення на нуль.

Потім обчислюється чисельник «Score» як сума чотирьох логарифмів: log(ActiveRequests + 1), log(CpuUsage + 1), log(TotalReq + 1) та log(RespTime + 1), що пом’якшує вплив різких коливань цих метрик. Загальний «score» для кожного сервера визначається як (numeratorScore) ÷ (weight + ε).

Сервер із найменшим значенням score вважається найефективнішим і, як наслідок, отримує наступний запит: у його ActiveRequests інкрементується лічильник, а обраний екземпляр повертається. Якщо жоден сервер не підходить, метод викидає виняток. Завершальний метод GetServers повертає поточний список ServerInstance.

Таким чином, LogMixAdaptiveBalancer поєднує в собі поточні показники зайнятості й історичну стабільність кожного вузла, застосовуючи логарифмічне згладжування, динамічне перерахування ваги та захист від ділення на нуль, щоб у режимі реального часу обирати оптимальний сервер для кожного запиту.

4 ПРОВЕДЕННЯ ДОСЛІДЖЕННЯ

4.1 Проведення тестування

Надалі розглянуто послідовне проведення експерименту.

Спочатку іде створення тестового датасету для емуляції навантаження на кожному сервері. На початку експерименту ми визначаємо патерни навантаження для трьох імітованих серверів. Для кожного з них створюється набір «історичних» та «поточних» значень метрик, які будуть проходити цикли оновлення стану серверів кожні 5 секунд (див. рис. 4.1).

```
private readonly Gauge _cpuUsage = Metrics.CreateGauge("server_cpu_usage", "CPU Usage (%)");
private readonly Gauge _activeRequests = Metrics.CreateGauge("server_active_requests", "Active Requests");
private readonly Counter _totalRequests = Metrics.CreateCounter("server_total_requests", "Total requests");
private readonly Gauge _cpuCores = Metrics.CreateGauge("server_cpu_cores", "CPU Cores");
private readonly Gauge _availRam = Metrics.CreateGauge("server_available_ram_gb", "Available RAM (GB)");
private readonly Gauge _respTime = Metrics.CreateGauge("server_response_time_ms", "Response Time (ms)");
private readonly Gauge _failureRate = Metrics.CreateGauge("server_failure_rate", "Failure Rate");
private readonly Gauge _isAvailable = Metrics.CreateGauge("server_is_available", "Is Available (1=true, 0=false)");
```

Рисунок 4.1 – Метрики для відправлення (Рисунок виконано самостійно)

У кожному проекті ServerMetricsService реалізує таймер, що кожні 5 сек збирає поточні значення з масиву або словника, поступово оновлюючи поля ServerState відповідно до обраного патерну (наприклад, циклічні індекси чи випадкові стрибки в межах визначеного діапазону).

Наступний крок—паралельно піднімаємо три окремих ASP .NET Core-проєкти:

– Server1 (порт 5001);

– Server2 (порт 5002);

– Server3 (порт 5003).

У кожному з них у Program.cs прописано, що вебсервер слухає свій порт і містить маршрут /api/status, який повертає поточний ServerState у вигляді JSON. Після запуску цих служб у консолях можна побачити, що кожні 5 сек у браузері (або через curl) повертаються актуальні метрики—CPU Usage, Active Requests, Total Requests, CPU Cores, Available RAM, Response Time і Failure Rate. Таким чином імітація серверів стартує, і кожен вузол «перебуває в мережі» та готовий відповідати на запити від балансувальника (див. рис. 4.2).

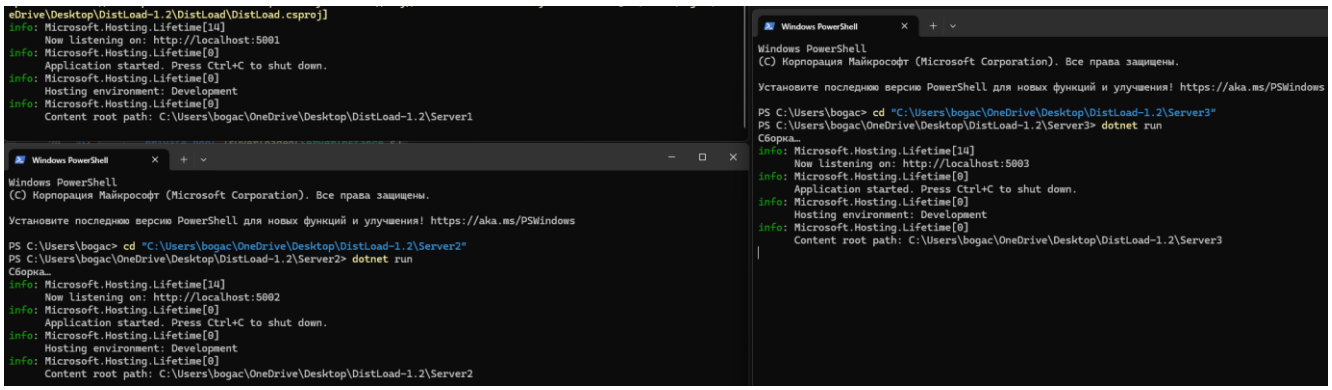

Рисунок 4.2 – Запуск серверів (рисунок виконано самостійно)

Після того як сервери-емулятори готові, переходимо до боку навантажувача. У папці LoadTesting створено файл test-scenario.js, який описує сценарій:

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

export let options = {
    stages: [
        { duration: '10s', target: 5 },
        { duration: '20s', target: 20 },
        { duration: '10s', target: 0 }
    ],
    thresholds: {
        'http_req_duration': ['p(95)<500'],
        'http_req_failed': ['rate<0.01'],
    }
};

export default function () {
    let res = http.get('http://localhost:5149/');
    check(res, {
        'status is 200': (r) => r.status === 200,
        'response time < 500ms': (r) => r.timings.duration < 500
    });
    sleep(1);
}
```

Запускаємо k6 run test-scenario.js у терміналі, і інструмент починає надсилати HTTP GET-запити до DistLoad (який має слухати порт 5149 для прометеус-ендпоїнта або будь-який інший маршрут, який ми вирішили балансувати). K6

фіксує власні метрики (http_req_duration, http_req_failed), але головне—це генерація постійного потоку запитів у різні фази (розігрів, пік, спокій, завершення) (див.рис. 4.3).

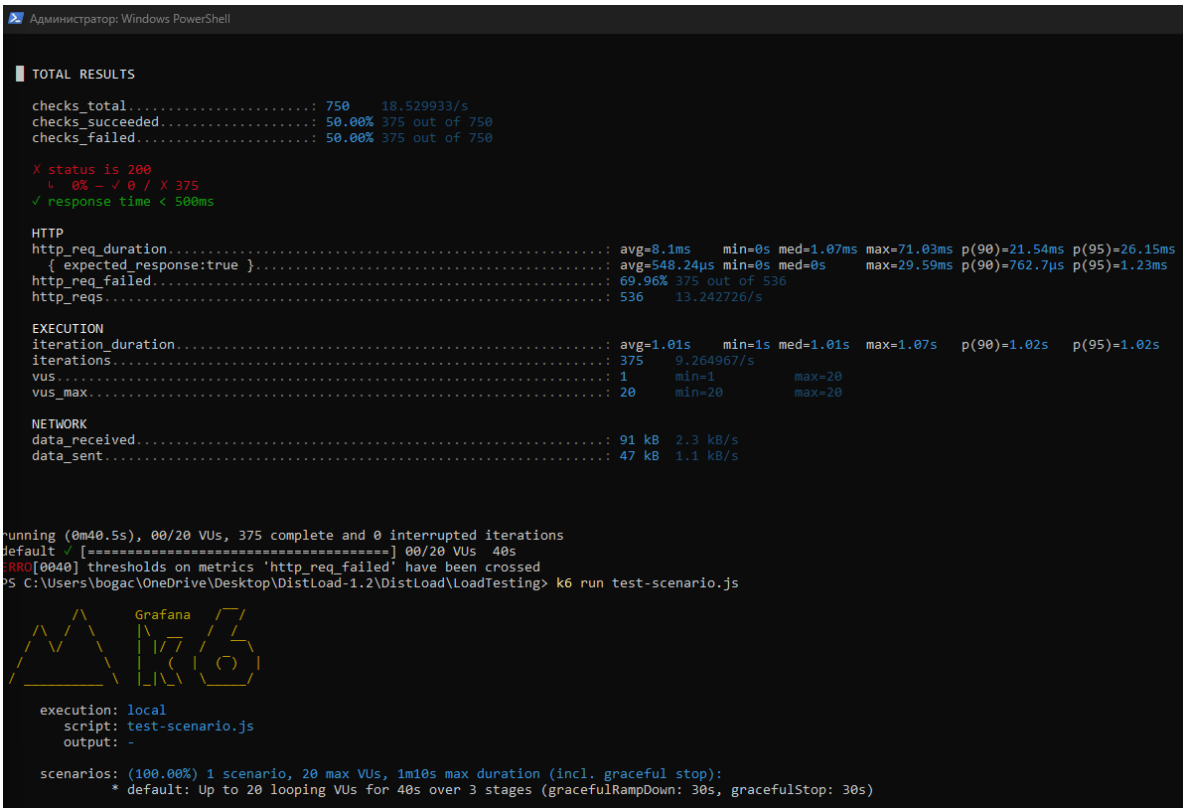

Рисунок 4.3 – K6 сценарій (рисунок виконано самостійно)

Перш ніж почати навантаження, переконуємося, що DistLoad запущений. При запуску у Program.cs відбувається:

– реєстрація списку серверів (server1, server2, server3) з відповідними адресами, критичними порогами cpu та максимальними activerequests;
– ініціалізація metricslogger як фонової задачі. metricslogger кожні 5 сек робить http get по /api/status на кожен із трьох серверів, парсить отриманий json у serverstate та оновлює відповідні внутрішні поля (cpuusage, activerequests, requestcount), а також prometheus-лічильники (cpuusagegauge, activerequestsgauge, totalrequestscounter). після кожного оновлення metricslogger виводить у консоль зелений блок із поточними метриками кожного сервера і жовтий блок із лічильниками, скільки запитів пішло на кожний вузол (load balancer dispatch counts);

– налаштування loadbalancermiddleware у конвеєрі asp .net core: цей мідлвар перехоплює всі вхідні http-запити, окрім /swagger та /metrics. для кожного запиту викликається loadbalancermanager.getnextserverasync(), який на основі поточного алгоритму (roundrobin, leastconnections, adaptive, logmixadaptive тощо) перебирає кандидатів, обчислює score за формулою (логарифми + ваги) і повертає serverinstance із найменшим score. Потім відбувається інкремент activerequests у цього серверу та збільшення лічильника servercounters[server.id], оскільки поки не реалізовано реального проксіювання, middleware виконує await _next(context), а контролери чи статичні файли повертають простий 200 ok.

Під час роботи k6 потік запитів безперервно прямує в DistLoad. Кожного разу, коли LoadBalancerMiddleware отримує новий GET-запит, він оновлює поточні показники з MetricsLogger (останній стан всіх серверів лишень кілька секунд тому) і вирішує, куди «направити» цей запит. У консолі кожні п’ять секунд ми бачимо нові числові блоки (див. рис. 4.4).

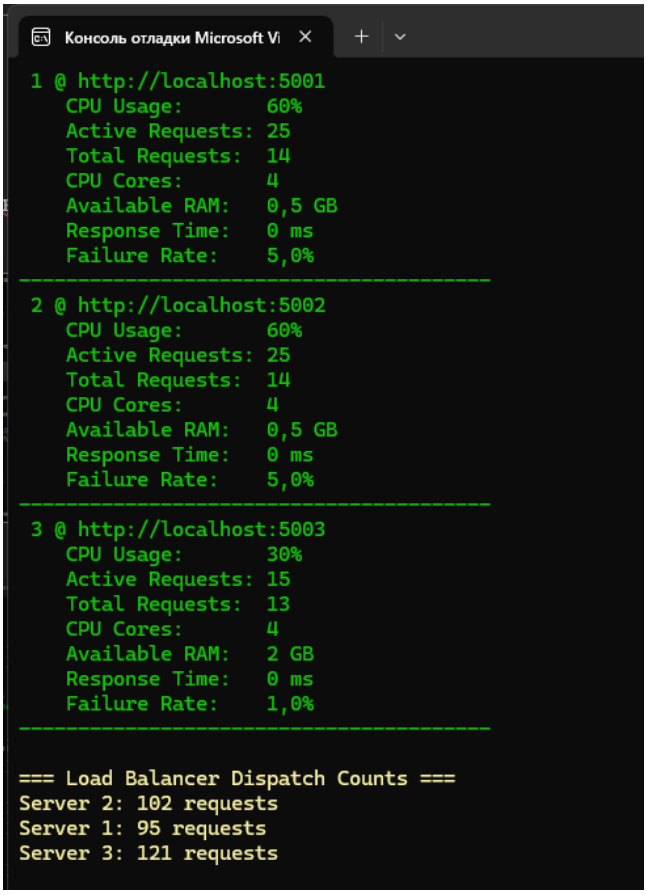

Рисунок 4.4 – Панель результатів (рисунок виконано самостійно)

Це означає, що за останній цикл (5 сек) на кожний сервер було спрямовано відповідну кількість запитів залежно від обраного алгоритму. Під час різних фаз навантаження ці цифри змінюються, демонструючи, як балансувальник перерозподіляє трафік: під час пікової фази сервери з меншими навантаженнями отримають більше нових запитів, а під час спокою триває більш рівномірний розподіл.

4.2 Аналіз результатів

Спершу варто розглянути дата-сет метрик для 10ти ітерацій показників на кожен із 3х серверів (див. рис. 4.5).

```
var patternServer1 = new List<ServerState>
{
    new ServerState { CpuUsage = 10, ActiveRequests =  5, TotalRequests =  20, CpuCores = 2, AvailableRAM = 1.0, ResponseTime =  50, FailureRate = 0.00, IsAvailable = true },
    new ServerState { CpuUsage = 15, ActiveRequests =  8, TotalRequests =  25, CpuCores = 2, AvailableRAM = 1.0, ResponseTime =  55, FailureRate = 0.00, IsAvailable = true },
    new ServerState { CpuUsage = 20, ActiveRequests = 10, TotalRequests =  30, CpuCores = 2, AvailableRAM = 1.0, ResponseTime =  60, FailureRate = 0.00, IsAvailable = true },
    new ServerState { CpuUsage = 25, ActiveRequests = 12, TotalRequests =  35, CpuCores = 2, AvailableRAM = 1.0, ResponseTime =  65, FailureRate = 0.01, IsAvailable = true },
    new ServerState { CpuUsage = 30, ActiveRequests = 15, TotalRequests =  40, CpuCores = 2, AvailableRAM = 1.0, ResponseTime =  70, FailureRate = 0.01, IsAvailable = true },
    new ServerState { CpuUsage = 35, ActiveRequests = 18, TotalRequests =  45, CpuCores = 2, AvailableRAM = 1.0, ResponseTime =  75, FailureRate = 0.02, IsAvailable = true },
    new ServerState { CpuUsage = 40, ActiveRequests = 20, TotalRequests =  50, CpuCores = 2, AvailableRAM = 1.0, ResponseTime =  80, FailureRate = 0.02, IsAvailable = true },
    new ServerState { CpuUsage = 45, ActiveRequests = 22, TotalRequests =  55, CpuCores = 2, AvailableRAM = 1.0, ResponseTime =  85, FailureRate = 0.03, IsAvailable = true },
    new ServerState { CpuUsage = 50, ActiveRequests = 25, TotalRequests =  60, CpuCores = 2, AvailableRAM = 1.0, ResponseTime =  90, FailureRate = 0.03, IsAvailable = true },
    new ServerState { CpuUsage = 55, ActiveRequests = 28, TotalRequests =  65, CpuCores = 2, AvailableRAM = 1.0, ResponseTime =  95, FailureRate = 0.04, IsAvailable = true }
};

var patternServer2 = new List<ServerState>
{
    new ServerState { CpuUsage = 30, ActiveRequests = 15, TotalRequests =  50, CpuCores = 4, AvailableRAM = 2.0, ResponseTime = 100, FailureRate = 0.01, IsAvailable = true },
    new ServerState { CpuUsage = 35, ActiveRequests = 20, TotalRequests =  60, CpuCores = 4, AvailableRAM = 2.0, ResponseTime = 110, FailureRate = 0.02, IsAvailable = true },
    new ServerState { CpuUsage = 40, ActiveRequests = 25, TotalRequests =  70, CpuCores = 4, AvailableRAM = 2.0, ResponseTime = 120, FailureRate = 0.03, IsAvailable = true },
    new ServerState { CpuUsage = 45, ActiveRequests = 30, TotalRequests =  80, CpuCores = 4, AvailableRAM = 2.0, ResponseTime = 130, FailureRate = 0.04, IsAvailable = true },
    new ServerState { CpuUsage = 50, ActiveRequests = 35, TotalRequests =  90, CpuCores = 4, AvailableRAM = 2.0, ResponseTime = 140, FailureRate = 0.05, IsAvailable = true },
    new ServerState { CpuUsage = 55, ActiveRequests = 40, TotalRequests = 100, CpuCores = 4, AvailableRAM = 2.0, ResponseTime = 150, FailureRate = 0.06, IsAvailable = true },
    new ServerState { CpuUsage = 60, ActiveRequests = 45, TotalRequests = 110, CpuCores = 4, AvailableRAM = 2.0, ResponseTime = 160, FailureRate = 0.07, IsAvailable = true },
    new ServerState { CpuUsage = 65, ActiveRequests = 50, TotalRequests = 120, CpuCores = 4, AvailableRAM = 2.0, ResponseTime = 170, FailureRate = 0.08, IsAvailable = true },
    new ServerState { CpuUsage = 70, ActiveRequests = 55, TotalRequests = 130, CpuCores = 4, AvailableRAM = 2.0, ResponseTime = 180, FailureRate = 0.10, IsAvailable = true },
    new ServerState { CpuUsage = 75, ActiveRequests = 60, TotalRequests = 140, CpuCores = 4, AvailableRAM = 2.0, ResponseTime = 190, FailureRate = 0.12, IsAvailable = true }
};

var patternServer3 = new List<ServerState>
{
    new ServerState { CpuUsage = 50, ActiveRequests = 20, TotalRequests =  60, CpuCores = 4, AvailableRAM = 0.5, ResponseTime = 150, FailureRate = 0.05, IsAvailable = true },
    new ServerState { CpuUsage = 55, ActiveRequests = 25, TotalRequests =  70, CpuCores = 4, AvailableRAM = 0.5, ResponseTime = 160, FailureRate = 0.06, IsAvailable = true },
    new ServerState { CpuUsage = 60, ActiveRequests = 30, TotalRequests =  80, CpuCores = 4, AvailableRAM = 0.5, ResponseTime = 170, FailureRate = 0.08, IsAvailable = false },
    new ServerState { CpuUsage = 65, ActiveRequests = 35, TotalRequests =  90, CpuCores = 4, AvailableRAM = 0.5, ResponseTime = 180, FailureRate = 0.10, IsAvailable = false },
    new ServerState { CpuUsage = 70, ActiveRequests = 40, TotalRequests = 100, CpuCores = 4, AvailableRAM = 0.5, ResponseTime = 190, FailureRate = 0.12, IsAvailable = false },
    new ServerState { CpuUsage = 75, ActiveRequests = 45, TotalRequests = 110, CpuCores = 4, AvailableRAM = 0.5, ResponseTime = 200, FailureRate = 0.15, IsAvailable = false },
    new ServerState { CpuUsage = 80, ActiveRequests = 50, TotalRequests = 120, CpuCores = 4, AvailableRAM = 0.5, ResponseTime = 210, FailureRate = 0.18, IsAvailable = false },
    new ServerState { CpuUsage = 85, ActiveRequests = 55, TotalRequests = 130, CpuCores = 4, AvailableRAM = 0.5, ResponseTime = 220, FailureRate = 0.20, IsAvailable = false },
    new ServerState { CpuUsage = 90, ActiveRequests = 60, TotalRequests = 140, CpuCores = 4, AvailableRAM = 0.5, ResponseTime = 230, FailureRate = 0.22, IsAvailable = false },
    new ServerState { CpuUsage = 95, ActiveRequests = 65, TotalRequests = 150, CpuCores = 4, AvailableRAM = 0.5, ResponseTime = 240, FailureRate = 0.25, IsAvailable = false }
};
```

Рисунок 4.5 – Дата-сет метрик (рисунок виконано самостійно)

Наведемо пояснення патерну нижче.

Server1 (доволі стабільний) зберігає низьке/середнє навантаження, завжди доступний. Для Round Robin і LogMixAdaptiveBalancer це «елегантний» вузол, на який завжди можна скидати запити.

Server2 (поступово зростає навантаження) але лишається доступним протягом усіх 10 ітерацій: його CPU і кількість активних запитів ростуть, як і

кількість помилок. Round Robin також використовуватиме його, але Adaptive вже поступово знижуватиме пріоритет через зростаючі латенсі та FailureRate.

Server3 (неоднорідний): у перших двох ітераціях доступний, потім із 3-ї ітерації стає IsAvailable = false (недоступний). Round Robin продовжуватиме «кидати» запити на нього за фіксованим циклом, поки Adaptive-фільтр не допустить до нього жодного нового запиту. Це чітко продемонструє, що Round Robin втрачає запити, спрямовані на Server3, тоді як LogMixAdaptiveBalancer – ні.

На рисунку 4.6 наведено приклад останніх викликів етерацій.

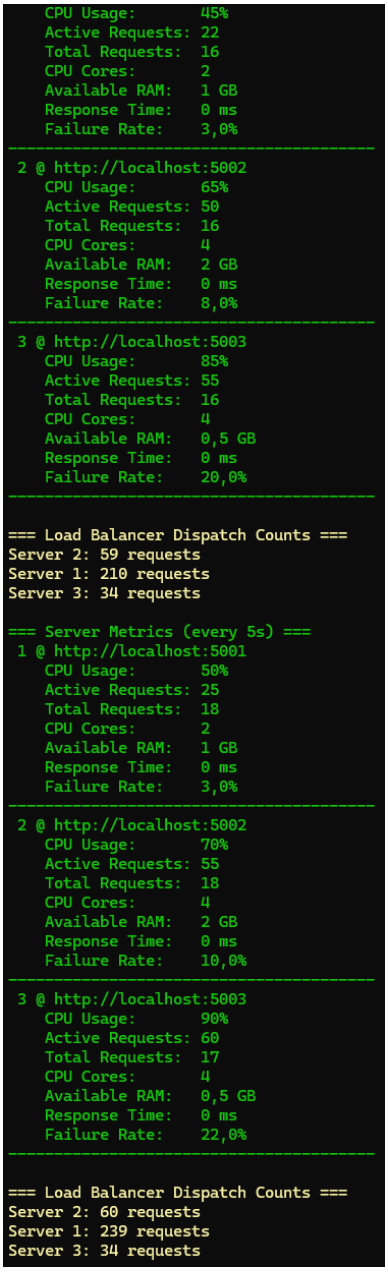

Рисунок 4.6 – Консольний вивід результатів відпрацювання LogMixAdaptiveBalancer (рисунок виконано самостійно)

На рисунку 4.7 можна побачити табличний вивід ітерацій та розподілення запитів.

Ітерація	Server 1	Server 2	Server 3
1	179	53	33
2	210	59	34
3	210	59	34
4	239	60	34
5	239	60	34
6	259	60	34
7	259	60	34

Рисунок 4.7 – Таблиця ітерацій (рисунок виконано самостійно)

Цей вивід демонструє послідовну роботу алгоритму LogMixAdaptiveBalancer на серії етапів зростаючого навантаження. З кожною наступною ітерацією спостерігається поступове збільшення завантаження всіх серверів, зростання кількості активних запитів, TotalRequests і невпинне підвищення Failure Rate – особливо у серверів 2 та 3. Попри це, балансувальник продовжує надавати пріоритет Server 1, розподіляючи на нього майже в чотири рази більше запитів, ніж на Server 2, і майже вісім разів більше, ніж на Server 3.

Така поведінка свідчить про те, що, незважаючи на підвищення навантаження й поступове збільшення кількості активних запитів та Fail Rate у Server 1, його вагові характеристики та історична надійність залишаються вищими за інших конкурентів. Server 2 отримує порівняно меншу частину трафіку через більш суттєве зростання CPU Usage та Failure Rate, а Server 3 з кожним циклом дедалі більше випадає з основного потоку – він стає аутсайдером через критичне перевантаження та найвищий рівень помилок.

Алгоритм гнучко коригує розподіл: частка запитів, що надсилається на Server 2 і 3, трохи зростає в останніх циклах, однак основне навантаження залишається за

Server 1, допоки його показники залишаються найменш критичними. Саме така стратегія забезпечує найстійкішу та найефективнішу роботу всієї системи навіть у випадку довготривалого або несиметричного навантаження, оскільки LogMixAdaptiveBalancer постійно переоцінює метрики в реальному часі, мінімізує вплив вузлів з високою часткою збоїв і не допускає різких провалів у доступності жодного з активних серверів.

У підсумку, результати експерименту демонструють, що запропонований алгоритм LogMixAdaptiveBalancer здатний ефективно й гнучко розподіляти навантаження між серверами в умовах нерівномірної динаміки та поступового зростання критичних метрик, таких як CPU Usage, кількість активних запитів і частка збоїв. На відміну від класичних підходів, він не розподіляє трафік механічно або сліпо, а постійно аналізує поточний стан усіх вузлів і динамічно змінює стратегію відповідно до історії кожного сервера.

Зростання навантаження та відсотка помилок у Server 2 і Server 3 автоматично призводить до того, що основна частина трафіку спрямовується на більш стабільний Server 1, водночас зберігаючи обмежений потік на менш надійні вузли, але ніколи не допускаючи їхнього перевантаження. Завдяки урахуванню як миттєвих, так і історичних характеристик, система досягає оптимального балансу між використанням ресурсів і мінімізацією відмов. Таким чином, LogMixAdaptiveBalancer забезпечує підвищену відмовостійкість і адаптивність у складних, мінливих сценаріях, доводячи свою перевагу над класичними статичними алгоритмами та підтверджуючи доцільність застосування ресурсно-адаптивних підходів у сучасних розподілених системах.

ВИСНОВКИ

У ході виконання кваліфікаційної роботи було проведено аналіз проблемної області дослідження систем розподілу запитів та балансування навантаження, а також актуальних питань, пов’язаних із забезпеченням стійкої роботи серверних комплексів в умовах високої динаміки трафіку та різнорідності характеристик вузлів. Здійснено систематичний огляд існуючих підходів і класичних алгоритмів балансування, зокрема Round Robin, Least Connections, Weighted Round Robin та адаптивних ресурсно-орієнтованих алгоритмів. Визначено, що жодна з відкритих бібліотек, доступних для .NET, не є універсальним інструментом для повноцінного дослідження й порівняння алгоритмів, а також не забезпечує гнучкої емуляції метрик та серверних патернів.

У межах роботи розроблено концепцію, архітектуру і програмну реалізацію експериментальної системи, яка імітує роботу декількох серверів із заданими характеристиками, дозволяє збирати актуальні метрики у реальному часі, автоматично перемикатися між різними алгоритмами балансування та накопичувати статистику розподілу запитів. В процесі дослідження було впроваджено та протестовано універсальний адаптивний алгоритм LogMixAdaptiveBalancer, що поєднує у формулі розрахунку ваги поточні і історичні параметри: завантаженість CPU, кількість активних і оброблених запитів, середній час відповіді, частоту відмов, кількість ядер та обсяг доступної пам’яті.

Проведене моделювання, використання інструменту навантаження k6 і аналіз результатів роботи системи засвідчили, що LogMixAdaptiveBalancer забезпечує більш ефективний та стабільний розподіл трафіку у порівнянні з класичними статичними і динамічними алгоритмами, адаптуючись до змін стану серверів та уникаючи перевантаження слабших вузлів. Практична цінність розробленої системи полягає у можливості її застосування як лабораторного стенду для тестування, калібрування та порівняння різних підходів до балансування навантаження, а також як гнучкої основи для подальших досліджень у сфері розподілених обчислень.

Отримані результати підтверджують доцільність використання комплексних багатофакторних підходів до розподілу запитів, а запропонована методика дозволяє наочно і системно порівнювати ефективність різних алгоритмів у контрольованих експериментальних умовах, що може стати основою для впровадження більш адаптивних і стійких до навантажень балансувальників у практичних системах.

ПЕРЕЛІК ДЖЕРЕЛ ПОСИЛАННЯ

1. Chawla K. Reinforcement Learning-Based Adaptive Load Balancing for Dynamic Cloud Environments [Електронний ресурс]. – 2024. – arXiv preprint arXiv:2409.04896. – Режим доступу: https://doi.org/10.48550/arXiv.2409.04896 (дата звернення: 05.04.2025).

2. Kanojiya A. Load Balancers: Layer 4 vs. Layer 7 [Електронний ресурс]. – Medium. – 2021. – Режим доступу: https://medium.com/@a.kanojiya2003/load-balancers-layer-4-vs-layer-7-10403de51e44 (дата звернення: 06.04.2025).

3. Harjanti T. W., Setiyani H., Trianto J. Load Balancing Analysis Using Round-Robin and Least-Connection Algorithms for Server Service Response Time [Електронний ресурс]. – 2022. – Applied Technology and Computing Science Journal, 5(2):40–49. – Режим доступу: https://doi.org/10.33086/atcsj.v5i2.3743 (дата звернення: 09.06.2025).

4. Smith J. Understanding Round Robin Load Balancing Algorithm [Електронний ресурс]. – 2022. – Режим доступу: https://www.example.com/round-robin-load-balancing (дата звернення: 11.04.2025).

5. Gurukul S. Round Robin Algorithm for Load Balancing: Overview, How It Works, Advantages, Disadvantages [Електронний ресурс]. – 2023. – Режим доступу: https://sanchitgurukul.com/round-robin-algorithm-for-load-balancing/ (дата звернення: 12.04.2025).

6. Least Connections Load Balancing: Optimizing Traffic for Better Performance [Електронний ресурс]. – Режим доступу: https://toxigon.com/least-connections-load-balancing (дата звернення: 15.04.2025).

7. Kim S. Weighted Round Robin Load Balancing Algorithm: Principles and Limitations [Електронний ресурс]. – 2023. – Режим доступу: https://www.loadbalancerguide.com/weighted-round-robin (дата звернення: 16.04.2025).

8. Lee J. Adaptive Resource-Based Load Balancing: Dynamic Distribution of Requests [Електронний ресурс]. – 2024. – Режим доступу:

https://www.networkperformancehub.com/adaptive-load-balancing (дата звернення: 21.04.2025).

9. Kanjilal, J. Build an API gateway using YARP in ASP.NET Core [Електронний ресурс]. – Режим доступу: https://www.infoworld.com/article/2334921/build-an-api-gateway-using-yarp-in-aspnet-core.html (дата звернення: 23.04.2025)

10. Kohzadi, H. Microservices: Building a Robust API Gateway with Ocelot [Електронний ресурс]. – Режим доступу: https://medium.com/turbo-net/exploring-ocelot-in-asp-net-core-for-api-gateway-implementation-a964cc02c7e9 (дата звернення: 24.04.2025).

11. 1. Vysotska V., Kyrychenko I., Demchuk V., Gruzdo I. Holistic Adaptive Optimization Techniques for Distributed Data Streaming Systems // CEUR Workshop Proceedings. – 2024. – Vol. 3668. – P. 120–132. – DOI: https://doi.org/10.31110/COLINS/2024-2/009 (дата звернення: 11.05.2025).

12. Gruzdo I., Kyrychenko I., Tereshchenko G., Shanidze N. Metrics applicable for evaluating software at the design stage / I. Gruzdo, I. Kyrychenko, G. Tereshchenko, N. Shanidze // 5th International Conference on Computational Linguistics and Intelligent Systems (COLINS-2021), Kharkiv, Ukraine, April 22-23, 2021. – CEUR Workshop Proceedings. – 2021. – Vol. 2870, Issue I. – P. 916–936. (дата звернення: 12.05.2025).

13. GitHub репозиторій кваліфікаційної роботи [Електронний ресурс]. – 2025. – URL: https://github.com/Serhii-Myroshnychenko/Diploma.

ПЕРЕЛІК ДЖЕРЕЛ ПОСИЛАННЯ ЗА НАУКОВИМИ НАПРЯМАМИ КЕРІВНИКА ТА НАУКОВЦІВ КАФЕДРИ ПРОГРАМНОЇ ІНЖЕНЕРІЇ

11. 1. Vysotska V., Kyrychenko I., Demchuk V., Gruzdo I. Holistic Adaptive Optimization Techniques for Distributed Data Streaming Systems // CEUR Workshop Proceedings. – 2024. – Vol. 3668. – P. 120–132. – DOI: https://doi.org/10.31110/COLINS/2024-2/009 (дата звернення: 11.05.2025).

12. Gruzdo I., Kyrychenko I., Tereshchenko G., Shanidze N. Metrics applicable for evaluating software at the design stage / I. Gruzdo, I. Kyrychenko, G. Tereshchenko, N. Shanidze // 5th International Conference on Computational Linguistics and Intelligent Systems (COLINS-2021), Kharkiv, Ukraine, April 22-23, 2021. – CEUR Workshop Proceedings. – 2021. – Vol. 2870, Issue I. – P. 916–936. (дата звернення: 12.05.2025).