Міністерство освіти і науки України

Луцький національний технічний університет

(повне найменування закладу вищої освіти)

Факультет комп'ютерних та інформаційних технологій

(повне найменування факультету)

Кафедра комп'ютерної інженерії та охоронних систем

(повне найменування кафедри)

КВАЛІФІКАЦІЙНА РОБОТА ЗА СТУПЕНЕМ ВИЩОЇ ОСВІТИ «БАКАЛАВР»

СИНХРОНІЗАЦІЯ ТА БАЛАНСУВАННЯ БАГАТОПОТОКОВОЇ СИСТЕМИ ОБРОБКИ ДАНИХ ДЛЯ ОНЛАЙН-КІНОТЕАТРУ

SYNCHRONIZATION AND BALANCING OF A MULTI-THREADED DATA PROCESSING SYSTEM FOR AN ONLINE CINEMA

спеціальність 123 Комп'ютерна інженерія

(шифр і назва спеціальності)

освітня програма Комп'ютерна інженерія

(назва освітньої програми)

Виконав: здобувач вищої освіти
групи КІ-41
Карасійчук Дмитро Ярославович

(підпис)

Керівник:
к.т.н., доцент
Мельник Катерина Вікторівна

(підпис)

Кваліфікаційну роботу
допущено до захисту
«_____» червня 2026 р.
Гарант освітньої програми:
к.т.н., доцент
Лавренчук Світлана Василівна

(підпис)

Луцьк – 2026 року

ЛУЦЬКИЙ НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ

Факультет *комп'ютерних та інформаційних технологій*
Кафедра *комп'ютерної інженерії та безпеки*
Ступінь вищої освіти: *бакалавр*
Галузь знань: *12 Інформаційні технології*
Спеціальність: *123 Комп'ютерна інженерія*
Освітня програма: *«Комп'ютерна інженерія»*

ЗАТВЕРДЖУЮ
Завідувач кафедри
_____ доц. Т. Терлецький
«__23__» _____12_____ 2025 р.

ЗАВДАННЯ
НА КВАЛІФІКАЦІЙНУ РОБОТУ ЗДОБУВАЧУ ВИЩОЇ ОСВІТИ

Карасійчуку Дмитру Ярославовичу
(прізвище, ім'я, по батькові)

1. Тема кваліфікаційної роботи *Синхронізаця та балансування багатопотокової системи обробки даних для онлайн-кінотеатру*

Керівник роботи *к.т.н., доцент Мельник Катерина Вікторівна*

затверджені наказом закладу вищої освіти від «20» грудня 2025 року № 536/01-02
2. Строк подання здобувачем вищої освіти кваліфікаційної роботи 28.05.2026 р.

3. Вихідні дані до роботи *Джерелом розробки є науково-технічна література та публікації в періодичних виданнях з даного питання, опубліковані зарубіжні та вітчизняні роботи в даній області, різні інтернет-ресурси технічного спрямування*

4. Зміст пояснювальної записки (перелік питань, які потрібно розробити):
Вступ
Аналіз проблеми синхронізації та балансування багатопотокових систем
Проєктування та реалізація системи синхронізації та балансування навантаження для online-кінотеатру сіпетаарр
Дослідження роботи системи синхронізації та аналіз ефективності балансування навантаження
Висновки

5. Перелік графічного (ілюстративного) матеріалу:
Аналіз архітектурних підходів лідерів індустрії
Трирівнева архітектура з інтегрованою підсистемою NeuralBalancer
Життєвий цикл запиту: 8 кроків синхронізації
Процес моніторингу
Аналіз ризиків паралелізму
Висновки

6. Консультанти розділів роботи

Розділ	Прізвище, ініціали та посада консультанта	Підпис	
		завдання видав	завдання прийняв
Аналіз проблеми синхронізації та балансування багатопотокових систем	*Мельник К. В., доцент*		
Проєктування та реалізація системи синхронізації та балансування навантаження для online-кінотеатру cinemaapp	*Мельник К. В., доцент*		
Дослідження роботи системи синхронізації та аналіз ефективності балансування навантаження	*Мельник К .В., доцент*		
Нормоконтроль	*Багнюк Н. В., доцент*		
Гарант ОП	*Лавренчук С. В., доцент*		
Показник запозичень тексту	***___%***		
Академічна доброчесність	*Міскевич О. І., ст. викладач*		

7. Дата видачі завдання 23.12.2025 р.

КАЛЕНДАРНИЙ ПЛАН

№ з/п	Назва етапів кваліфікаційної роботи	Строк виконання етапів роботи	Примітка
1.	*Огляд літератури із досліджуваної проблеми, аналіз предметної області та наявних рішень*	до 10.02.2026 р.	
2.	*Аналіз проблеми синхронізації та балансування багатопотокових систем*	до 02.03.2026 р.	
3.	*Проєктування та реалізація системи синхронізації та балансування навантаження для online-кінотеатру cinemaapp*	до 02.04.2026 р.	
4.	*Дослідження роботи системи синхронізації та аналіз ефективності балансування навантаження*	до 10.04.2026 р.	
5.	*Представлення остаточного варіанту кваліфікаційної роботи керівникові*	до 01.05.2026 р.	
6.	*Нормоконтроль*	до 23.05.2026 р.	
7.	*Інструментальна перевірка на академічний плагіат*	до 20.05.2026 р.	
8.	*Здача кваліфікаційної роботи та всіх супровідних документів на кафедру*	до 28.05.2026 р.	

Здобувач вищої освіти _____ Дмитро КАРАСІЙЧУК
(підпис) (прізвище, ініціали)

Керівник кваліфікаційної роботи _____ Катерина МЕЛЬНИК
(підпис) (прізвище, ініціали)

АНОТАЦІЯ

Карасійчук Д. Синхронізація та балансування багатопотокової системи обробки даних для онлайн-кінотеатру. Рукопис.

Кваліфікаційна робота бакалавра ОП «Комп'ютерна інженерія» спеціальності 123 Комп'ютерна інженерія. Луцький національний технічний університет. Луцьк, 2026.

Кваліфікаційна робота присвячена дослідженню та практичній реалізації механізмів синхронізації потоків і динамічного балансування навантаження у багатопотоковій системі обробки даних для онлайн-кінотеатру. Робота складається зі вступу, трьох розділів, висновків, переліку використаних джерел.

Перший розділ містить теоретичний аналіз проблеми синхронізації та балансування навантаження в багатопотокових системах: розглянуто моделі багатопотоковості, основні примітиви синхронізації: м'ютекси, монітори, черги повідомлень, Read-Write Locks, атомарні операції та алгоритми балансування навантаження: Round Robin, Least Connections, Work Stealing. Проведено порівняльний аналіз архітектурних підходів стримінгових платформ Netflix, Megogo та Sweet.tv.

Другий розділ присвячено проєктуванню та реалізації системи CinemaApp на базі ASP.NET Core 9 і C#. Описано архітектуру підсистеми NeuralBalancer, розробку двох багатошарових нейронних мереж: LoaderNeuralNetwork та BalancerNeuralNetwork без використання зовнішніх ML-бібліотек.

Третій розділ містить результати дослідження поведінки системи в умовах динамічного навантаження, аналіз ефективності балансування, вхідні та вихідні дані підсистеми, а також ідентифікацію та усунення потенційних ризиків паралелізму.

Ключові слова: багатопотоковість, синхронізація потоків, балансування навантаження, нейронна мережа, ASP.NET Core, SemaphoreSlim, онлайн-кінотеатр, паралелізм.

ANNOTATION

Karasiichuk D. Synchronization and balancing of a multi-threaded data processing system for an online cinema. Manuscript.

Qualifying work of a bachelor of EP «Computer Engineering» specialty 123 Computer Engineering. Lutsk National Technical University. Lutsk, 2026.

The qualifying work focuses on the research and practical implementation of thread synchronization mechanisms and dynamic load balancing in a multi-threaded data processing system for an online cinema. The work consists of an introduction, three chapters, conclusions, a list of references.

The first chapter provides a theoretical analysis of synchronization and load balancing in multi-threaded systems: thread models, key synchronization primitives: mutexes, monitors, message queues, Read-Write Locks, atomic operations, and load balancing algorithms: Round Robin, Least Connections, Work Stealing are examined. A comparative analysis of the architectural approaches of Netflix, Megogo, and Sweet.tv streaming platforms is carried out.

The second chapter describes the design and implementation of the CinemaApp system built on ASP.NET Core 9 and C#. The architecture of the NeuralBalancer subsystem is presented, including two custom multilayer neural networks: LoaderNeuralNetwork and BalancerNeuralNetwork implemented without external ML libraries.

The third chapter presents the results of studying system behavior under dynamic load, an analysis of balancing efficiency, input and output data flows of the subsystem, and identification and mitigation of potential concurrency risks.

Keywords: multithreading, thread synchronization, load balancing, neural network, ASP.NET Core, SemaphoreSlim, online cinema, parallelism.

ЗМІСТ

ВСТУП

Актуальність теми. Сучасні онлайн-кінотеатри обслуговують мільйони паралельних підключень, де кожна секунда простою або деградації сервісу безпосередньо впливає на якість досвіду користувача. В умовах стрімкого зростання обсягів споживання відеоконтенту та підвищених вимог до часу відгуку серверних застосунків проблема ефективного управління паралельним виконанням задач набуває критичного значення. Некоректна синхронізація потоків призводить до гонок даних, взаємних блокувань і втрати консистентності стану, тоді як нераціональний розподіл навантаження спричиняє перевантаження окремих компонентів і деградацію продуктивності всієї системи. Попри широке застосування стандартних примітивів синхронізації, питання їхнього поєднання з адаптивними механізмами балансування навантаження, здатними динамічно реагувати на зміни інтенсивності вхідного потоку запитів, залишається недостатньо дослідженим у контексті реальних серверних застосунків.

Метою роботи є дослідження та практична реалізація механізмів синхронізації потоків і динамічного балансування навантаження у багатопотоковій системі обробки даних для онлайн-кінотеатру.

Об’єкт дослідження – процеси синхронізації та балансування навантаження в багатопотокових серверних системах.

Предмет дослідження – механізми синхронізації потоків і алгоритм динамічного балансування навантаження на основі нейронної мережі у серверному застосунку онлайн-кінотеатру.

Для досягнення мети необхідно виконати такі завдання:

– дослідити теоретичні засади багатопотоковості, існуючі примітиви синхронізації та алгоритми балансування навантаження, а також проаналізувати архітектурні підходи провідних стримінгових платформ;

– спроєктувати та реалізувати серверний застосунок CinemaApp на базі ASP.NET Core 9 з трирівневою архітектурою та підтримкою багатопотокового виконання;

– розробити підсистему NeuralBalancer, що включає дві нейронні мережі прямого поширення (мережу-навантажувач та мережу-балансувальник), реалізовані без використання зовнішніх ML-фреймворків;

– реалізувати комплекс механізмів синхронізації потоків (lock/Monitor, SemaphoreSlim, AutoResetEvent, Interlocked, ConcurrentQueue) та забезпечити потокобезпечність критичних операцій;

– дослідити поведінку системи в умовах динамічного навантаження, оцінити ефективність балансування та ідентифікувати потенційні ризики паралелізму.

Практична цінність роботи визначається тим, що розроблена система може слугувати як прототип підсистеми адаптивного управління навантаженням для реальних серверних застосунків онлайн-кінотеатрів та інших високонавантажених веб-сервісів.

РОЗДІЛ 1
АНАЛІЗ ПРОБЛЕМИ СИНХРОНІЗАЦІЇ ТА БАЛАНСУВАННЯ БАГАТОПОТОКОВИХ СИСТЕМ

1.1 Сутність і принципи багатопотокової обробки даних

У сучасному світі інформаційних технологій розрізняють два фундаментальні поняття: процес і потік виконання [10, 18]. Процес являє собою незалежну програму, яка виконується в ізольованому адресному просторі операційної системи. Кожен процес має власні ресурси, такі як: пам’ять, файлові дескриптори, мережеві з’єднання. Потік або ж іншими словами thread представляє деяку одиницю виконання всередині процесу [16]. Всі потоки одного процесу поділяють єдиний адресний простір, але мають власні стеки, програмні лічильники та стани регістрів. Чжаогуо Сюй, Кріс Стюарт та Роберт Гріффіт у своїй статті «Concurrency Control in Modern Distributed Systems» [18] наводять детальну класифікацію моделей потоків у контексті сучасних розподілених систем.

Така організація має принципове значення для серверних застосунків [19]. Онлайн-кінотеатр, отримуючи HTTP-запит від користувача, може обробляти його в окремому потоці, не блокуючи обробку інших запитів. Потоки дешевші за процеси у створенні та перемиканні контексту: у Linux створення потоку займає близько 10 мікросекунд проти 100 мікросекунд для процесу [18]. Крім того, спільний адресний простір дозволяє потокам швидко обмінюватися даними через загальну пам’ять [17].

Разом із тим спільний адресний простір є джерелом небезпеки [2, 18]. Якщо два потоки одночасно змінюють одну ділянку пам’яті без синхронізації, виникає невизначена поведінка – так звана гонка даних або іншими словами data race. Для її запобігання необхідні механізми синхронізації, розгляду яких присвячено підрозділ 1.2.

Моделі багатопотоковості.

Існує кілька концептуальних моделей організації паралельного виконання [10, 18]. Перша, яку запропонували автори Чжаогуо Сюй, Кріс Стюарт та Роберт Гріффіт це модель спільної пам’яті(shared memory model), яка передбачає

те, що потоки мають прямий доступ до єдиного адресного простору [18]. Вони стверджують, що це найефективніша модель з точки зору продуктивності, але найскладніша з точки зору коректності. Саме цю модель реалізують операційні системи через POSIX-потоки та відповідні API багатьох мов програмування.

Друга модель запропонована Адамом Балльмаром представляє передачу повідомлень, які ізолюють потоки або процеси від спільних даних [1]. Вони спілкуються лише через явне надсилання повідомлень через канали або черги. Ця модель,є безпечнішою, але може мати накладні витрати на копіювання даних. У контексті мікросервісної архітектури вона реалізується через Apache Kafka [14], RabbitMQ або Redis Pub/Sub [13].

Третя модель, яку запропонували Камал Індрасірі та Прабат Сірівардена у своїй книзі «Design Patterns for Cloud Native Applications» – акторна моделью, де розглядається кожен об’єкт як незалежного актора, який обробляє повідомлення з власної поштової скриньки [5]. Фреймворки Akka та Actix реалізують цей підхід. Для онлайн-кінотеатру акторна модель особливо підходить для реалізації чергування плейлистів, де кожен сеанс перегляду є незалежним актором зі своїм станом.

Також досить важливо розрізняти і такі широкі поняття як паралелізм (parallelism) і конкурентність (concurrency) [2, 18]. Простими словами конкурентність – це по суті здатність системи мати кілька завдань у процесі виконання одночасно, навіть якщо в конкретний момент часу виконується лише одне. А паралелізм це вже фактично одночасне або ж навіть псевдо-одночасне виконання на кількох процесорних ядрах [19]. Конкурентність є логічною властивістю системи, паралелізм – фізичною.

Шварц Барон та Ткаченко Василь [13], а також Менасце Деніел, Алмейда Вірджіліо та Ріфф Лоуренс [11] розглядають практичне значення обох понять для стримінгових платформ. Автори пояснюють, що конкурентність забезпечує обслуговування тисяч одночасних HTTP-з’єднань через механізм асинхронного вводу-виводу навіть на однопроцесорній машині, тоді як паралелізм дозволяє використовувати всі ядра сервера для обробки відеопотоків, транскодування та аналітики. Компанія NGINX Inc. [9] описує, як сучасні веб-сервери комбінують

обидва підходи: асинхронний event loop для обробки з’єднань і worker-процеси для паралелізму.

Бонер Джонас та співавтори [2] та Урма Рауль-Габріель, Майкрофт Алан і Ворбертон Річард [16] визначають поняття потокобезпечного коду (thread-safe code) як коду, який коректно функціонує при одночасному виконанні кількома потоками. Автори виокремлюють три ключові принципи забезпечення потокобезпеки : принципи атомарності, видимості та впорядкованості.

Принцип атомарності вимагає, щоб критичні операції виконувалися неподільно або повністю, або не виконувалися взагалі [18]. Сучасні процесори надають атомарні інструкції (compare-and-swap, fetch-and-add), які дозволяють реалізовувати lock-free алгоритми [6].

Принцип видимості вимагає, щоб зміни, внесені одним потоком, були своєчасно видимі іншим [2]. Сучасні процесори мають складні кеші та буфери запису, що можуть затримати поширення змін. Для забезпечення видимості використовуються бар’єри пам’яті (memory barriers) та ключові слова volatile у різних мовах програмування [17]. У Java модель пам’яті (Java Memory Model) формально визначає правила видимості між потоками [2].

Принцип впорядкованості стосується порядку операцій [15, 18]. Компілятори та процесори можуть змінювати порядок виконання інструкцій для оптимізації, що іноді порушує коректність багатопотокових програм. Механізми синхронізації не лише забезпечують взаємовиключення, але й встановлюють чіткі відношення «відбулось-раніше» (happens-before) [2], що гарантують правильний порядок спостережуваних операцій.

Індрасірі Камал та Сірівардена Прабат [5], а також Менасце Деніел та співавтори [7] розглядають реалізацію багатопотоковості в стримінгових платформах на кількох рівнях архітектури. Урма Рауль-Габріель та співавтори [16] пояснюють, що на рівні веб-сервера кожен HTTP-запит обробляється в окремому потоці або корутині в рамках пулу потоків. Беллемар Адам [1] досліджує рівень бізнес-логіки і стверджує, що фонові завдання – нотифікації, нарахування статистики, оновлення рекомендацій – виконуються асинхронно через черги повідомлень. Менасце Деніел та співавтори [7] описують рівень бази даних, де пул

з’єднань забезпечує ефективне використання обмеженої кількості дорогих з’єднань між великою кількістю потоків.

1.2 Огляд основних механізмів синхронізації потоків

Синхронізація потоків є ключовим аспектом побудови багатопотокових систем і визначається як процес координації доступу до спільних ресурсів з метою забезпечення коректності, узгодженості та передбачуваності виконання програм [2, 17, 18]. Як зазначають О. Прокопенко та С. Білоус [11], а також В. Савченко і А. Коваленко [12], сучасні операційні системи надають широкий спектр механізмів синхронізації, кожен з яких має власну область застосування, переваги та обмеження.

М’ютекс є одним із базових примітивів синхронізації, що забезпечує взаємне виключення доступу до критичної секції [17, 18]. Відповідно до досліджень А. Вільямса, м’ютекс функціонує як двостановий механізм (захоплений/вільний), який дозволяє лише одному потоку виконувати критичну ділянку коду в конкретний момент часу [17].

Реалізація м’ютексів базується на використанні атомарних інструкцій процесора, таких як compare-and-swap або test-and-set, що гарантують неподільність операцій [6, 18]. У POSIX-сумісних системах м’ютекси підтримують різні режими роботи (NORMAL, RECURSIVE, ERRORCHECK), а також механізми успадкування пріоритету для запобігання інверсії пріоритетів [20].

Практичне застосування м’ютексів у високонавантажених системах, зокрема в онлайн-кінотеатрах, полягає у захисті спільних лічильників або обмежених ресурсів. Наприклад, відповідно до підходів, описаних Д. Менаше та ін. [7], м’ютекси можуть використовуватись для контролю кількості одночасних підключень до ресурсу, що має ліцензійні обмеження.

Водночас, як підкреслюють А. Ковальчук та Л. Тимченко [19], застосування м’ютексів пов’язане з ризиком виникнення взаємних блокувань (deadlock). Для їх уникнення використовуються стратегії впорядкованого захоплення ресурсів, таймаути або алгоритми виявлення циклів очікування.

Окремим різновидом м’ютексів є spinlock, який реалізує активне очікування. Такий підхід є ефективним лише для коротких критичних секцій у багатоядерних системах, де витрати на перемикання контексту перевищують час очікування [18].

Семафори є більш універсальним механізмом синхронізації порівняно з м’ютексами, оскільки дозволяють керувати доступом до ресурсу за допомогою лічильника [12, 18]. Як зазначають В. Савченко та А. Коваленко [12], семафори широко застосовуються у високонавантажених вебсервісах для регулювання рівня паралелізму.

Операції над семафором (wait/P та signal/V) виконуються атомарно, що забезпечує коректність роботи навіть у конкурентному середовищі [18]. Бінарні семафори функціонально подібні до м’ютексів, тоді як рахівні семафори дозволяють одночасний доступ до ресурсу кільком потокам.

Згідно з роботами Д. Менаше та ін. [7], семафори ефективно застосовуються для керування пулами ресурсів, зокрема з’єднаннями з базою даних. Крім того, вони широко використовуються для реалізації класичних задач синхронізації, таких як «виробник-споживач» [2].

Для онлайн-кінотеатру рахівний семафор ідеально підходить для управління пулом з’єднань до бази даних [9, 11]. Якщо максимальна кількість з’єднань дорівнює 100, семафор ініціалізується значенням 100. Кожен потік, що хоче отримати з’єднання, виконує P (зменшує лічильник). Якщо лічильник стає від’ємним (всі з’єднання зайняті), потік чекає. Після завершення роботи з’єднання повертається у пул. Це гарантує, що ніколи не буде більше 100 одночасних з’єднань, незалежно від кількості потоків.

Монітори та умовні змінні.

Монітори є високорівневим механізмом синхронізації, що інкапсулює спільні дані, методи їх обробки та механізми взаємного виключення в єдину абстракцію [16]. Як зазначають Р.-Г. Урма, А. Майкрофт та Р. Варбуртон [16], у сучасних мовах програмування (зокрема Java) монітори реалізуються за допомогою вбудованих конструкцій синхронізації.

Ключовим елементом монітора є умовні змінні, які дозволяють потокам очікувати на виконання певних умов. Операції wait, signal та broadcast забезпечують ефективну координацію потоків без активного опитування [2, 18].

Важливим аспектом використання умовних змінних є необхідність повторної перевірки умови після пробудження через можливість «помилкових пробуджень» (spurious wakeup), що детально описано у роботі А. Вільямса [17].

Черги повідомлень.

Черги повідомлень є механізмом асинхронної взаємодії між потоками або процесами, який дозволяє уникнути прямого доступу до спільної пам’яті [1]. Як зазначає А. Беллмар [1], використання черг повідомлень є основою побудови подієво-орієнтованих систем.

Даний підхід забезпечує слабке зв’язування компонентів системи та дозволяє ефективно обробляти нерівномірні навантаження шляхом буферизації повідомлень [13]. У сучасних розподілених системах широко використовуються брокери повідомлень, такі як Apache Kafka, архітектуру якого детально описано Г. Шапірою та співавторами [14]. Ці автори [14] стверджують, що Kafka забезпечує горизонтальне масштабування завдяки розподілу повідомлень між розділами (partitions), що дозволяє досягати високої пропускної здатності при мінімальних затримках.

Read-Write Locks та атомарні операції.

Для сценаріїв, де операції читання значно переважають операції запису, доцільно використовувати блокування типу read-write lock, які дозволяють одночасний доступ багатьох потоків для читання та виключний доступ для запису [18].

Як зазначають Б. Шварц і В. Ткаченко [13], такі механізми є ефективними для систем кешування, де частота читання значно перевищує частоту оновлення даних.

Атомарні операції, у свою чергу, забезпечують виконання дій без використання блокувань, що суттєво підвищує продуктивність системи [6]. Використання примітивів, таких як compare-and-swap, дозволяє реалізовувати lock-free структури даних.

Дослідження Т. А. Ле та Х. Д. Нгуєна [6] демонструють, що lock-free алгоритми забезпечують значно кращу продуктивність у високонавантажених системах порівняно з традиційними підходами на основі блокувань.

1.3 Методи балансування навантаження в багатопотокових системах

Балансування навантаження є ключовим механізмом підвищення ефективності функціонування багатопотокових і розподілених систем. Воно визначається як процес розподілу запитів або обчислювальних задач між доступними ресурсами з метою оптимізації продуктивності, мінімізації часу відгуку та запобігання перевантаженню окремих компонентів системи [7].

Як зазначають К. Індрасірі та П. Сірівардена [5], у сучасних хмарних архітектурах балансування навантаження реалізується на різних рівнях: від внутрішнього планування потоків до глобального розподілу запитів між серверами в кластері.

Алгоритм Round Robin та його варіації.

Алгоритм Round Robin є одним із найпростіших і найпоширеніших методів балансування навантаження [7, 9]. Його суть полягає у послідовному розподілі запитів між доступними ресурсами у циклічному порядку. Такий підхід забезпечує рівномірний розподіл навантаження за умови однакової вартості обробки запитів.

Однак, як підкреслюється в документації NGINX [9], базовий алгоритм Round Robin не враховує гетерогенність ресурсів. Для вирішення цієї проблеми застосовується зважений варіант – Weighted Round Robin, який дозволяє враховувати продуктивність окремих серверів шляхом призначення їм ваг.

Ще однією модифікацією є IP Hash, який забезпечує прив’язку клієнта до конкретного сервера. Це дозволяє реалізувати так звану «липкість сесій» – session affinity, який є важливим для систем із тривалими з’єднаннями [9].

Алгоритми на основі стану навантаження.

На відміну від Round Robin, алгоритми на основі стану враховують поточні характеристики системи. Одним із найпоширеніших є алгоритм Least Connections, який направляє запити до ресурсу з найменшою кількістю активних з’єднань [4].

Як зазначають Д. Менаше та ін. [7], цей підхід є ефективним у середовищах із нерівномірною тривалістю запитів, оскільки дозволяє уникнути перевантаження окремих серверів.

Більш складним є алгоритм Least Response Time, який додатково враховує середній час відгуку сервера. Це дозволяє більш точно оцінювати реальне навантаження та підвищувати якість обслуговування користувачів [4].

Найбільш адаптивним підходом є балансування на основі ресурсів (resource-based load balancing), що враховує завантаження CPU, пам’яті та мережевих інтерфейсів [5, 7]. Такий підхід активно використовується у мікросервісних архітектурах.

Пули потоків (Thread Pools).

Пул потоків є ефективним механізмом управління паралельними обчисленнями, який дозволяє уникнути накладних витрат на створення та знищення потоків [2, 17]. Згідно з дослідженнями А. Вільямса [17], використання пулів потоків значно підвищує продуктивність системи за рахунок повторного використання потоків.

Ключові параметри пулу потоків включають мінімальний розмір, тобто кількість потоків, яка постійно підтримуються активними, максимальний розмір – верхня межа, час життя неактивного потоку, після якого надлишкові потоки знищуються та розмір черги завдань [2, 16].

Як зазначають Р.-Г. Урма та співавтори [16], правильне налаштування цих параметрів є критичним для досягнення оптимального балансу між продуктивністю та використанням ресурсів.

У серверних застосуваннях (зокрема вебсерверах) пули потоків дозволяють ефективно обробляти велику кількість одночасних запитів, обмежуючи при цьому загальну кількість активних потоків [13].

Work Stealing.

Алгоритм Work Stealing є сучасним підходом до балансування навантаження у багатопотокових системах [13, 18]. Його особливість полягає в тому, що кожен потік має власну чергу завдань, а у разі простою може «викрадати» задачі з черг інших потоків.

Як підкреслюють Р.-Г. Урма та співавтори [16], такий підхід мінімізує конкуренцію за спільні ресурси та зменшує потребу в синхронізації.

Реалізація цього алгоритму у Java представлена у вигляді ForkJoinPool, який ефективно використовується для рекурсивних паралельних обчислень [2].

Реактивне програмування та асинхронна обробка.

Реактивне програмування є сучасною парадигмою побудови масштабованих систем, яка базується на асинхронній обробці даних та принципі зворотного тиску (backpressure) [2].

Згідно з роботою В. Пасічника та Н. Юрченка [20], механізм backpressure дозволяє динамічно регулювати швидкість генерації даних відповідно до можливостей їх обробки, що запобігає перевантаженню системи.

Reactive Manifesto [2] визначає основні характеристики реактивних систем: чутливість, стійкість, еластичність та орієнтованість на повідомлення. Такі системи широко застосовуються у високонавантажених сервісах, зокрема стримінгових платформах.

1.4 Аналіз існуючих підходів до масштабування і розподілу навантаження

Сучасні стримінгові платформи використовують комплексні підходи до забезпечення масштабованості та ефективного розподілу навантаження. Як зазначають Б. Бернс та ін. [3], основою таких систем є принципи хмарних обчислень і мікросервісної архітектури.

Netflix: мікросервісна архітектура та технологічний стек.

Платформа Netflix є одним із найбільш репрезентативних прикладів реалізації високонавантажених розподілених систем, побудованих на основі мікросервісної архітектури та хмарної інфраструктури [5, 8]. Згідно з офіційними технічними матеріалами компанії, повний перехід до хмарного середовища дозволив забезпечити високу масштабованість, еластичність і відмовостійкість системи [8].

Архітектурна модель Netflix базується на декомпозиції функціональності на велику кількість незалежних мікросервісів, кожен з яких виконує обмежений набір завдань і може масштабуватися автономно [5]. Такий підхід відповідає сучасним принципам побудови cloud-native систем, що передбачають слабке зв’язування компонентів і використання стандартизованих протоколів взаємодії.

Особливістю системи є застосування клієнтського балансування навантаження, що реалізується за допомогою бібліотеки Ribbon [8]. На відміну від централізованих балансувальників, даний підхід передбачає прийняття рішення щодо маршрутизації запиту безпосередньо клієнтською стороною. Це дозволяє зменшити затримки та усунути єдину точку відмови. Актуальна інформація про доступні екземпляри сервісів надається через систему виявлення Eureka, що забезпечує динамічне оновлення конфігурації [8].

Для забезпечення відмовостійкості застосовується патерн Circuit Breaker, який передбачає тимчасове блокування викликів до сервісів у разі їх деградації або перевищення порогових значень часу відповіді [5]. Такий підхід дозволяє запобігати поширенню помилок у системі та забезпечує стабільність її функціонування.

Значна увага приділяється обробці потокових даних, що реалізується за допомогою платформи Apache Kafka [14]. Використання розподіленого журналу подій дозволяє забезпечити високу пропускну здатність та масштабованість при обробці великих обсягів інформації в режимі реального часу.

Крім того, для оптимізації доставки контенту використовується власна мережа доставки Open Connect, яка забезпечує кешування даних на периферійних вузлах і зменшує навантаження на центральну інфраструктуру [8].

Отже, підхід Netflix характеризується високим рівнем децентралізації, широким використанням асинхронних механізмів взаємодії та орієнтацією на горизонтальне масштабування.

Megogo: гібридна архітектура та адаптація до українського ринку.

Питання оптимізації інфраструктури розподілених систем розглядаються у працях Б. Шварца та В. Ткаченка [13], які зазначають, що поєднання хмарних і

локальних ресурсів дозволяє досягти балансу між продуктивністю та вартістю. Саме такий підхід реалізовано у платформі Megogo.

Як зазначають дослідники [13], використання гібридної архітектури дозволяє зменшити затримки за рахунок локалізації обробки даних, а також забезпечити масштабування при пікових навантаженнях за допомогою хмарних ресурсів. У Megogo це реалізується через розподіл інфраструктури між власними серверами та зовнішніми дата-центрами.

Балансування навантаження в таких системах, за даними офіційної документації NGINX та HAProxy [4], здійснюється на декількох рівнях. Зокрема, GeoDNS дозволяє направляти запити до найближчих серверів, що, як підкреслюється у [9], суттєво зменшує затримки доступу.

На рівні прикладних сервісів застосовуються алгоритми балансування, такі як Round Robin та Least Connections, які, за визначенням Д. Менаше та співавторів [7], забезпечують ефективний розподіл запитів залежно від поточного стану системи.

Важливим аспектом є також організація обробки відеоданих. А. Беллмар [1] у своїй роботі підкреслює, що подієво-орієнтовані архітектури дозволяють ефективно масштабувати системи за рахунок асинхронної обробки задач. У Megogo це реалізується через використання черг повідомлень для організації процесів транскодування.

Для оптимізації доступу до даних застосовуються кеш-системи. Б. Шварц та В. Ткаченко [13] зазначають, що використання Redis у поєднанні зі стратегією cache-aside дозволяє значно зменшити навантаження на базу даних і підвищити швидкість обробки запитів.

Таким чином, Megogo демонструє приклад збалансованого підходу до побудови інфраструктури, який поєднує централізовані та розподілені механізми управління навантаженням.

Sweet.tv: спеціалізований підхід до IPTV та OTT.

Особливості потокових сервісів, орієнтованих на IPTV та OTT, розглядаються у працях Д. Менаше та співавторів [7], які підкреслюють відмінності

між індивідуальною та широкомовною передачею даних. Платформа Sweet.tv реалізує поєднання цих підходів, що визначає специфіку її архітектури.

Як зазначають В. Оліфер та Н. Оліфер [10], використання технології мультикасту дозволяє значно зменшити навантаження на мережу за рахунок передачі одного потоку даних декільком користувачам одночасно. Це є характерною рисою IPTV-сервісів.

Для OTT-компонента системи застосовуються CDN-мережі, які, за словами Б. Шварца та В. Ткаченка [13], забезпечують кешування контенту та його доставку з мінімальними затримками. Такий підхід дозволяє ефективно масштабувати систему при зростанні кількості користувачів.

Проблема пікових навантажень, як зазначається у [13], вирішується шляхом застосування механізмів rate limiting та jitter, що дозволяє розподіляти запити у часі та уникати перевантаження серверів.

Крім того, В. Пасічник та Н. Юрченко [20] досліджують використання потокових технологій у відеосервісах і зазначають, що сегментоване зберігання даних є ефективним підходом для реалізації функцій відкладеного перегляду. У Sweet.tv це реалізується через збереження відеосегментів у розподілених сховищах із використанням швидких кеш-систем.

Порівняльний аналіз підходів.

Порівнюючи архітектурні підходи трьох платформ, можна виділити кілька спільних принципів та відмінностей [5, 8, 13] в таблиці 1.1.

Таблиця 1.1 – Порівняльний аналіз підходів

Критерій	Netflix	Megogo	Sweet.tv
1	2	3	4
Тип архітектури	Повністю мікросервісна, хмарна	Гібридна (хмара + власні сервери)	IPTV/OTT-орієнтована
Рівень децентралізації	Клієнтський (Ribbon), service discovery (Eureka)	Середній	Середній
Метод балансування навантаження	Клієнтський (Ribbon), service discovery (Eureka)	DNS (GeoDNS), NGINX, HAProxy	CDN, HTTP-балансування, rate limiting
Основні алгоритми балансування	Least Connections, zone-aware	Round Robin, Weighted RR, Least Connections	Rate limiting, розподіл у часі (jitter)

Продовження таблиці 1.1

1	2	3	4
Механізми масштабування	Горизонтальне (auto-scaling у AWS)	Горизонтальне + частково вертикальне	Горизонтальне через CDN
Обробка подій та даних	Apache Kafka (масштабована потокова обробка)	Черги повідомлень (RabbitMQ або аналоги)	Подієво-орієнтована обробка
Синхронізація даних	Eventual consistency, розподілені кеші (EVCache)	Cache-aside (Redis + БД)	Redis + сегментоване зберігання
Відмовостійкість	Circuit Breaker (Hystrix), реплікація	Балансувальники + резервування	Балансувальники + резервування
Особливості доставки контенту	Власна CDN (Open Connect)	CDN + локальні сервери	Multicast + CDN
Оптимізація пікових навантажень	Автоматичне масштабування, черги	Балансування + буферизація	Rate limiting, jitter
Типові сценарії використання	Глобальний стримінг, Big Data	Регіональний стримінг	Живе телебачення (IPTV)

Аналіз показує, що всі розглянуті платформи застосовують горизонтальне масштабування як базовий підхід до підвищення продуктивності систем [3]. При цьому Netflix характеризується максимальною децентралізацією та використанням клієнтського балансування, що забезпечує високу гнучкість і масштабованість [8]. Megogo реалізує компромісний підхід, поєднуючи хмарні та локальні ресурси, що дозволяє оптимізувати затримки та витрати [13]. У свою чергу, Sweet.tv орієнтується на специфіку IPTV, використовуючи мультикаст і CDN для ефективної доставки потокового контенту [7].

РОЗДІЛ 2
ПРОЄКТУВАННЯ ТА РЕАЛІЗАЦІЯ СИСТЕМИ СИНХРОНІЗАЦІЇ ТА БАЛАНСУВАННЯ НАВАНТАЖЕННЯ ДЛЯ ONLINE-КІНОТЕАТРУ CINEMAAPP

2.1 Загальна архітектура розробленої системи та її місце у проєкті

Розробка сучасних високонавантажених веб-застосунків потребує використання ефективних механізмів синхронізації потоків, балансування навантаження та оптимального розподілу обчислювальних ресурсів. Особливо актуальними такі задачі є для онлайн-кінотеатрів, у яких одночасно може обслуговуватись значна кількість користувачів. У процесі роботи система повинна забезпечувати стабільне виконання великої кількості паралельних запитів, підтримувати високу швидкодію та мінімізувати затримки під час доступу до ресурсів. Саме тому в межах даної роботи було реалізовано систему синхронізації та балансування навантаження для онлайн-кінотеатру CinemaApp.

Розроблений застосунок побудований на платформі ASP.NET Core 9 із використанням мови програмування C#. Вибір даної технології обумовлений її високою продуктивністю, підтримкою багатопотоковості, розвиненою системою залежностей та можливістю створення масштабованих серверних застосунків. Для організації структури програмного забезпечення було використано трирівневу архітектуру, яка дозволяє розділити логіку системи на окремі незалежні компоненти (рис. 2.1).

Рисунок 2.1 – Огляд всієї структури проєкту онлайн-кінотеатру

Першим рівнем є шар доступу до даних, який відповідає за взаємодію із базою даних. У даному шарі реалізовано моделі, конфігурації сутностей, репозиторії та механізми роботи з Entity Framework Core. Для зберігання інформації використовується база даних SQLite, що дозволяє забезпечити простоту розгортання та достатню продуктивність для тестування системи. Використання ORM-технології Entity Framework Core значно спрощує взаємодію з даними та дозволяє реалізувати об’єктно-орієнтований підхід до роботи із базою даних.

Другим рівнем є шар бізнес-логіки, який містить основні сервіси системи, класи обробки даних, DTO-об’єкти, механізми валідації та профілі AutoMapper. У даному шарі реалізуються основні функціональні можливості застосунку, включаючи управління користувачами, обробку фільмів, категорій, бронювання та інших компонентів онлайн-кінотеатру. Відокремлення бізнес-логіки від інтерфейсу користувача дозволяє забезпечити гнучкість системи та спрощує її подальшу модифікацію.

Третім рівнем є рівень представлення, який реалізований за допомогою ASP.NET Core MVC. Він включає контролери, Razor Views та механізми взаємодії з користувачем через веб-інтерфейс. Саме у межах цього шару інтегрована підсистема NeuralBalancer, яка відповідає за синхронізацію та балансування навантаження.

Підсистема NeuralBalancer реалізована як окремий незалежний модуль, що працює паралельно з основним сервером застосунку (рис. 2.2). Вона функціонує у середовищі ASP.NET Core Generic Host та запускається автоматично після старту застосунку. Для цього використовується інтерфейс IHostedService, який дозволяє створювати фонові сервіси. Такий підхід забезпечує незалежне виконання підсистеми та дозволяє виконувати моделювання навантаження одночасно з обробкою HTTP-запитів. Взаємодія між підсистемою та основним застосунком здійснюється через спільні потокобезпечні структури даних, зокрема ConcurrentQueue та Interlocked-лічильники, що унеможливлює виникнення гонки даних. Завдяки такій архітектурі підсистема NeuralBalancer може адаптивно коригувати кількість робочих потоків у режимі реального часу, не порушуючи стабільності основного серверного процесу.

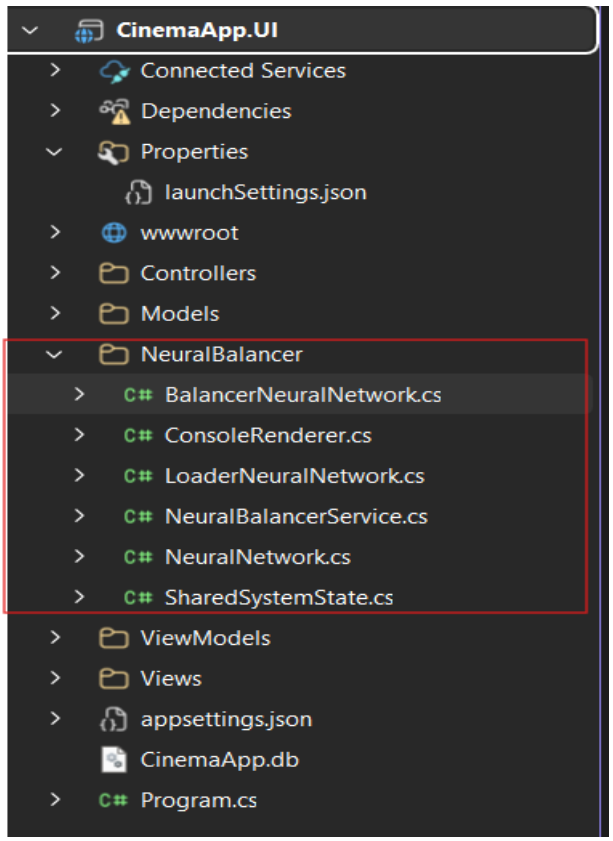

Рисунок 2.2 – Огляд структури підсистеми NeuralBalancer

Основною задачею підсистеми є генерація складного динамічного навантаження та управління кількістю паралельних задач. Для цього використовуються нейронні мережі, механізми синхронізації та спільний стан системи. Архітектура підсистеми включає декілька взаємопов’язаних компонентів, серед яких центральне місце займає сервіс NeuralBalancerService. Він координує роботу генератора навантаження, балансувальника та механізмів синхронізації.

Важливою особливістю розробленої системи є використання багатопотокового середовища виконання. У процесі роботи одночасно можуть виконуватись десятки або сотні задач, які звертаються до спільних ресурсів. У таких умовах виникає проблема конкурентного доступу до пам’яті, що може призвести до гонок даних, втрати інформації або нестабільної роботи системи. Для запобігання таким ситуаціям було реалізовано механізми синхронізації, які забезпечують потокобезпечність усіх критичних операцій.

2.2 Реалізація нейронних мереж з нуля та їх роль у системі

Ключовим елементом системи балансування навантаження є нейронні мережі, які використовуються для генерації нелінійних сигналів та управління параметрами роботи системи. У межах роботи було реалізовано власну модель багатошарової нейронної мережі без використання сторонніх бібліотек машинного навчання. Такий підхід дозволив забезпечити повний контрольнад внутрішньою логікою роботи системи та реалізувати необхідні механізми синхронізації (рис. 2.3).

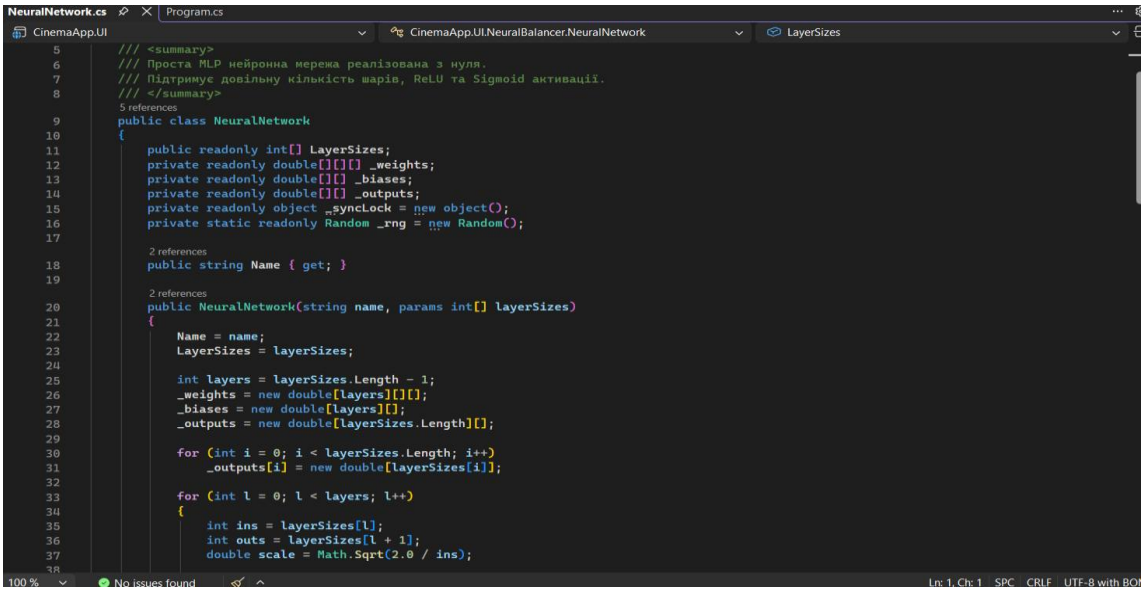

Рисунок 2.3 – Клас NeuralNetwork

Рішення реалізувати нейронну мережу самостійно замість використання готових фреймворків на кшталт ML.NET або ONNX Runtime є свідомим архітектурним вибором, обумовленим двома чинниками: по-перше, повним контролем над механізмом синхронізації під час обчислень; по-друге, відсутністю зовнішніх залежностей, які могли б мати власні обмеження потоковості або вимагати окремої ліцензійної угоди.

Математична основа MLP описується наступним чином. Для довільного нейрона з індексом i у шарі l активаційне значення обчислюється за формулою 2.1 [3]:

$$a_i^{(l)} = f(\Sigma_j w_{ij}^{(l)} \cdot a_j^{(l-1)} + b_i^{(l)}), \quad (2.1)$$

де $w_{ij}^{(l)}$ – ваговий коефіцієнт з'єднання між j-м нейроном попереднього шару та і-м нейроном поточного шару;

$b_i^{(l)}$ – зміщення або bias і-го нейрона;

f – функція активації.

Суть цієї моделі базується на принципі обчислення лінійної комбінації вхідних значень. Кожне вхідне значення множиться на відповідний ваговий коефіцієнт, після чого до результату додається зміщення. Отримане значення передається у функцію активації, яка формує нелінійну залежність між входом та виходом мережі. Завдяки цьому нейронна мережа здатна моделювати складні процеси та створювати динамічні сигнали.

Для прихованих шарів у системі використовується функція активації ReLU – Rectified Linear Unit, яка є однією з найпоширеніших функцій у сучасних нейронних мережах за формулою 2.2 [3]:

$$\mathrm{Re}\,LU(x) = \begin{cases} x, x > 0 \\ 0, x \leq 0 \end{cases}, \quad (2.2)$$

де x – вхідне значення нейрона.

Її основною перевагою є простота обчислень та висока швидкодія.

Для вихідного шару використовується сигмоїдна функція активації, що дозволяє обмежити значення виходу в діапазоні від нуля до одиниці за формулою 2.3 [3]:

$$Sigmoid(x) = \frac{1}{1+e^{-x}}, \quad (2.3)$$

де x – вхідний аргумент.

Ініціалізація вагових коефіцієнтів виконується за методом «He», також відомим, як «Kaiming initialization», призначений для глибоких нейронних мереж з

функціями активації типу ReLU або його варіантів. Його суть полягає в тому, щоб забезпечити стабільний розподіл значень між шарами мережі та дозволити уникнути надмірного затухання або зростання сигналу. Використання правильної ініціалізації є важливим для забезпечення стабільної роботи нейронної мережі навіть у випадку відсутності процесу навчання.

На практиці ваги ініціалізуються з рівномірного розподілу через масштабний коефіцієнт s в інтервалі [−s,+s] за формулою 2.4 [3]:

$$s = \sqrt{\frac{2}{n_{in}}}, \tag{2.4}$$

де n_{in} – кількість вхідних нейронів.

Цей коефіцієнт масштабування унеможливлює здатність функції ReLU обнуляти від'ємну частину розподілу. Без нього дисперсія активацій зменшувалась би приблизно вдвічі на кожному шарі, що призводило б якраз до цього затухання сигналу. Зміщення ініціалізуються невеликим позитивним константним значенням 0,01 замість нуля для запобігання так званим «мертвим нейронам» ReLU – нейронам, які повертають нуль для будь-якого входу через поєднання нульових ваг і нульового зміщення (рис. 2.4).

```
20  public NeuralNetwork(string name, params int[] layerSizes)
21  {
22      Name = name;
23      LayerSizes = layerSizes;
24
25      int layers = layerSizes.Length - 1;
26      _weights = new double[layers][][];
27      _biases = new double[layers][];
28      _outputs = new double[layerSizes.Length][];
29
30      for (int i = 0; i < layerSizes.Length; i++)
31          _outputs[i] = new double[layerSizes[i]];
32
33      for (int l = 0; l < layers; l++)
34      {
35          int ins = layerSizes[l];
36          int outs = layerSizes[l + 1];
37          double scale = Math.Sqrt(2.0 / ins);
38
39          _weights[l] = new double[outs][];
40          _biases[l] = new double[outs];
41
42          for (int n = 0; n < outs; n++)
43          {
44              _weights[l][n] = new double[ins];
45              for (int w = 0; w < ins; w++)
46                  _weights[l][n][w] = (_rng.NextDouble() * 2 - 1) * scale;
47              _biases[l][n] = 0.01;
48          }
49      }
50  }
```

Рисунок 2.4 – Ініціалізація вагів методом «He» у конструкторі NeuralNetwork

У процесі реалізації особливу увагу було приділено проблемам багатопотокового доступу. Оскільки нейронна мережа використовується одночасно декількома потоками, виникає ризик пошкодження спільних даних. Для вирішення цієї проблеми було реалізовано механізм синхронізації на основі блокування lock. Кожне виконання методу обчислення мережі захищається критичною секцією, що гарантує коректність результатів навіть у випадку паралельного виконання.

Окрему проблему становить використання внутрішнього буфера результатів. Для підвищення продуктивності результати обчислень зберігаються у внутрішньому масиві, який повторно використовується при наступних викликах. Однак це може призвести до ситуації, коли один потік змінює дані, які ще використовуються іншим потоком. Для запобігання цьому результати перед поверненням копіюються у новий масив. Такий підхід дозволяє забезпечити повну потокобезпечність системи.

2.3 Відсутність навчання та концепція нейронної мережі з фіксованими вагами

Принциповою особливістю реалізованих нейронних мереж є свідома відсутність алгоритму навчання. Алгоритм зворотного поширення помилки (backpropagation), який є стандартним методом навчання MLP у класичному машинному навчанні, у даному проєкті не реалізований. Ваги та зміщення ініціалізуються один раз у конструкторі за методом «Не» і залишаються незмінними протягом усього часу роботи застосунку [3]. Це означає, що мережі не адаптуються до реальних даних і не оптимізують жодну функцію втрат .

Зрозуміти сенс такого підходу допомагає аналіз цільової функції кожної мережі. Нейронна мережа-навантажувач (LoaderNeuralNetwork) призначена для генерації реалістичного, нерівномірного та непередбачуваного навантаження на систему. Якби навантаження генерувалось простим генератором псевдовипадкових чисел, воно було б статистично рівномірним і не відображало б реальні паттерни поведінки користувачів онлайн-кінотеатру, для яких характерні

хвилі активності. Нейронна мережа з ненавченими але фіксованими вагами і нелінійними функціями активації генерує детерміновані або іншими словами визначені (при фіксованому вхідному векторі результат завжди однаковий), але складно передбачувані та нелінійні перетворення вхідного вектора. Це забезпечує реалістичний характер навантаження без необхідності збирати та розмічати навчальні дані.

Нейронна мережа-балансер (BalancerNeuralNetwork) використовується для нелінійного відображення трьох вхідних метрик системи у два керуючі сигнали. Детерміністичне правило балансування (якщо навантаження більше 75%, то встановити кількість слотів менше певного значення) не є нейронною мережею – це звичайна умовна логіка. Нейронна мережа з фіксованими вагами, навпаки, виконує нелінійну трансформацію вхідного вектора з ненульовою взаємодією між компонентами. Навіть без навчання вона генерує унікальний характер керуючих рішень, що відповідає концепції дослідження синхронізації в умовах динамічного, а не статичного регулятора.

2.4 Реалізаця механізмів генерації та балансування навантаження

Перша нейронна мережа підсистеми реалізована у класі LoaderNeuralNetwork та побудована за архітектурою 4-8-8-1. Вхідний шар обробляє чотири параметри, два приховані шари містять по вісім нейронів, а вихідний шар формує одне результуюче значення. Загальна кількість параметрів моделі становить 121, що включає вагові коефіцієнти та зміщення між усіма шарами мережі. Ініціалізація мережі виконується під час створення об’єкта класу.

Основним призначенням цієї нейронної мережі є визначення рівня інтенсивності генерації HTTP-запитів на кожній ітерації циклу роботи підсистеми. Для цього формується вхідний вектор із чотирьох нормалізованих компонентів. Перший компонент являє собою циклічну часову мітку, що змінюється у межах від 0 до 1 протягом десятисекундного інтервалу. Другий компонент є псевдовипадковим числом із рівномірним розподілом у діапазоні [0;1]. Третій компонент формується на основі синусоїдної функції, що створює плавні

хвилеподібні коливання навантаження з іншим часовим масштабом. Четвертий компонент відображає поточний рівень завантаженості системи та виконує роль механізму зворотного зв’язку, надаючи мережі інформацію про актуальний стан середовища.

Усі вхідні параметри попередньо нормалізуються до діапазону [0;1], що забезпечує стабільність роботи нейронної мережі та усуває проблему домінування ознак із різними числовими масштабами. Після формування вхідного вектора виконується прямий прохід через архітектуру мережі із застосуванням функції активації ReLU у прихованих шарах та Sigmoid у вихідному. Результатом є значення коефіцієнта агресивності генерації навантаження, яке належить інтервалу (0;1).

На основі отриманого коефіцієнта визначається кількість HTTP-запитів, які мають бути ініційовані під час поточної ітерації. За мінімального значення агресивності система генерує один запит, тоді як за максимального – до семи запитів одночасно. Додатково значення агресивності впливає на тривалість паузи між ітераціями циклу: чим вищий рівень агресивності, тим меншою є затримка між циклами генерації. Таким чином, збільшення інтенсивності роботи призводить одночасно до зростання кількості запитів та скорочення часових інтервалів між ними, що забезпечує нелінійне нарощування навантаження.

HTTP-запити надсилаються до одного з п’яти ендпоінтів вебзастосунку, вибір яких здійснюється псевдовипадковим чином. Вони відповідають реальним сторінкам онлайн-кінотеатру, зокрема головній сторінці, каталогу фільмів, жанрам та розкладу сеансів. Запити виконуються як повноцінні HTTP GET-звернення до сервера Kestrel, а не у вигляді емуляції. Оскільки значна частина сторінок захищена механізмами автентифікації ASP.NET Core Identity, неавторизовані звернення можуть отримувати відповіді типу HTTP 302 із перенаправленням на сторінку входу. Для цілей статистичного аналізу всі відповіді з кодом менше 400 вважаються успішними.

Кожний HTTP-запит виконується асинхронно у вигляді окремої задачі за принципом «fire-and-forget», що дозволяє підсистемі продовжувати генерацію нового навантаження без очікування завершення попередніх операцій.

Використання механізмів async/await забезпечує виконання таких задач у потоках пулу ThreadPool, що підвищує ефективність роботи системи в умовах високого рівня паралелізму.

Друга нейронна мережа підсистеми реалізована у класі BalancerNeuralNetwork та має архітектуру 3-6-4-2. Вхідний шар приймає три параметри, перший прихований шар містить шість нейронів, другий – чотири, а вихідний шар формує два результуючі значення. Загальна кількість параметрів мережі становить 62. Ініціалізація моделі виконується під час створення відповідного об’єкта класу.

Функціональне призначення BalancerNeuralNetwork суттєво відрізняється від LoaderNeuralNetwork. Якщо перша мережа відповідає за генерацію навантаження, то друга виконує роль реактивного регулятора, який аналізує поточний стан системи та визначає допустимий рівень паралельного виконання HTTP-запитів. Вхідний вектор мережі формується на основі трьох метрик спільного стану системи: загального рівня завантаженості, частки активних воркерів та ступеня заповнення черги запитів. Усі параметри нормалізуються до діапазону [0;1], що забезпечує коректну роботу функцій активації.

Вихідний вектор мережі містить два значення, отримані після застосування функції Sigmoid. Перше значення інтерпретується як коефіцієнт дроселювання, який характеризує рівень перевантаження системи. Чим більше його значення, тим агресивніше мережа рекомендує обмежувати обробку запитів. Другий вихідний параметр визначає рекомендовану частку доступних слотів для паралельного виконання запитів. Саме цей коефіцієнт безпосередньо впливає на кількість дозволених одночасних HTTP-запитів.

Формування остаточного рішення щодо кількості доступних слотів здійснюється шляхом поєднання результатів нейронної мережі з детерміністичною логікою керування. У критичних ситуаціях, коли рівень навантаження перевищує 90 %, застосовується аварійне обмеження до одного слота без врахування виходу нейронної мережі. Такий підхід забезпечує додаткову стабільність системи у випадках потенційно непередбачуваної поведінки моделі. За помірних рівнів навантаження використовується адаптивне масштабування кількості слотів, де

допустимий рівень паралелізму змінюється залежно від коефіцієнта, сформованого мережею.

Цикл роботи BalancerNeuralNetwork активується двома способами. Перший механізм передбачає реакцію на сигнали, які надходять після завершення HTTP- запитів. Другий механізм забезпечує періодичну перевірку стану системи навіть за відсутності нових сигналів протягом визначеного інтервалу часу. Це дозволяє уникнути ситуацій, коли балансувальник припиняє активне регулювання через повільне виконання запитів або низьку частоту подій. Інформація про джерело активації використовується виключно для формування журналів роботи системи та відображення стану на інформаційній панелі.

2.5 Реалізація механізмів синхронізації та потокобезпечності

Підсистема синхронізації та балансування навантаження NeuralBalancer побудована на поєднанні декількох механізмів багатопотокової взаємодії, які забезпечують коректний обмін даними між паралельно виконуваними компонентами, стабільність роботи системи та контроль рівня паралелізму. Центральним елементом архітектури виступає клас SharedSystemState, який виконує роль спільного середовища зберігання стану для компонентів LoaderNeuralNetwork, BalancerNeuralNetwork та ConsoleRenderer (рис. 2.5).

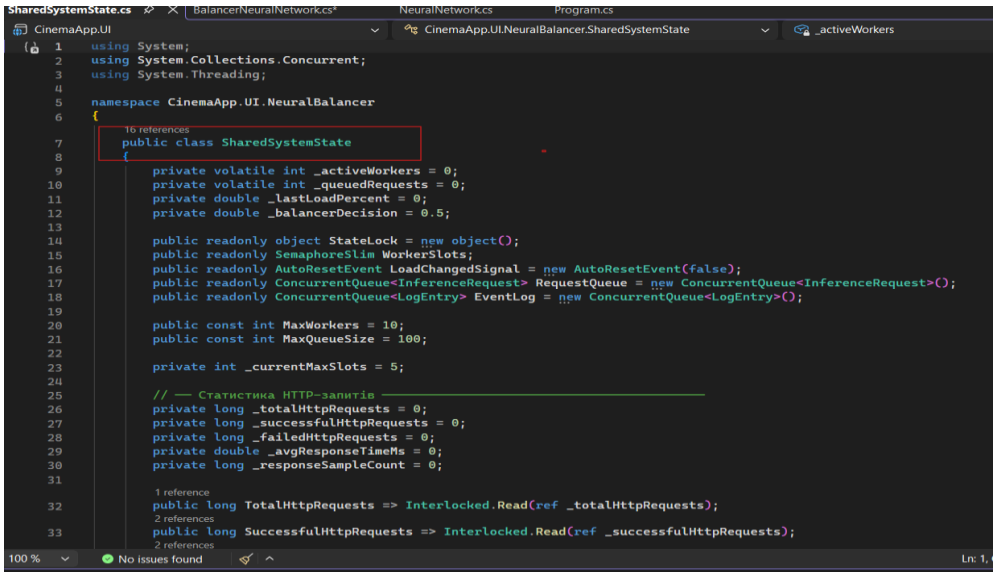

Рисунок 2.5 – Клас SharedSystemState

Саме цей клас акумулює ключові метрики системи, статистику HTTP- запитів та примітиви синхронізації, що забезпечують взаємодію між потоками.

Для забезпечення потокобезпечного доступу до даних у SharedSystemState використовуються декілька рівнів синхронізації. Частина змінних оголошується з модифікатором volatile, що гарантує актуальність значень під час читання та запису у багатопотоковому середовищі. Однак через відсутність атомарності складених операцій для зміни таких полів застосовуються атомарні інструкції класу Interlocked. Для більш складних операцій, які потребують узгодженого оновлення декількох взаємопов’язаних змінних, використовується механізм взаємного виключення через оператор lock. Такий підхід дозволяє уникнути станів гонки даних та забезпечує консистентність внутрішнього стану системи.

Оператор lock у середовищі .NET базується на механізмі Monitor та реалізує безпечне блокування критичних секцій із гарантованим звільненням ресурсу навіть у випадку виникнення виключень. У підсистемі використовуються окремі об’єкти блокування для різних груп даних, що дозволяє зменшити рівень конкуренції між потоками та підвищити ефективність паралельного виконання. Блокування застосовується лише для тих операцій, де це дійсно необхідно, тоді як прості лічильники оновлюються за допомогою Interlocked без використання критичних секцій.

Важливу роль у системі відіграє SemaphoreSlim, який реалізує механізм динамічного обмеження паралелізму. Семафор підтримує облік доступних слотів для виконання HTTP-запитів і дозволяє балансувальнику змінювати рівень одночасної активності залежно від поточного навантаження. Кожен запит перед відправленням повинен отримати дозвіл семафора, а після завершення обробки слот звільняється. Такий підхід забезпечує контроль кількості паралельних задач та запобігає перевантаженню системи. Використання асинхронного очікування через WaitAsync робить механізм сумісним із сучасною моделлю асинхронного програмування ASP.NET Core.

Для організації взаємодії між потоками застосовується також механізм AutoResetEvent, який реалізує подієву сигналізацію про зміну стану системи. Після

завершення HTTP-запиту балансувальник отримує сигнал про необхідність повторної оцінки рівня навантаження та можливого коригування кількості доступних слотів. Особливістю AutoResetEvent є те, що сигнали не накопичуються, а лише повідомляють про сам факт зміни стану, що дозволяє уникнути зайвих ітерацій балансувальника та зменшує навантаження на систему.

Клас Interlocked використовується для реалізації високопродуктивних атомарних операцій на рівні процесора. За його допомогою оновлюються лічильники активних воркерів, статистика HTTP-запитів, кількість заблокованих операцій та інші числові показники. Атомарні інструкції не потребують системних викликів чи блокувань, що забезпечує значно вищу продуктивність порівняно з традиційними механізмами взаємного виключення. Це особливо важливо для операцій, які виконуються з високою частотою в умовах значного паралелізму.

Передача даних між компонентами підсистеми реалізується через ConcurrentQueue, яка є безблокувальною потокобезпечною чергою. Вона використовується для буферизації HTTP-запитів та передачі лог-записів між потоками. Завдяки використанню lock-free алгоритмів черга забезпечує високу продуктивність навіть за великої кількості одночасних операцій. Для представлення запитів використовується тип record, що забезпечує структурну рівність та часткову незмінність об’єктів. З метою уникнення проблем спільного доступу до змінюваних масивів під час передачі даних між потоками виконується клонування вхідних даних перед постановкою об’єкта до черги.

Загалом поєднання механізмів lock, Monitor, SemaphoreSlim, AutoResetEvent, Interlocked та ConcurrentQueue дозволяє побудувати ефективну систему синхронізації та балансування навантаження, здатну стабільно працювати в умовах високого рівня паралелізму. Використання різних підходів до синхронізації залежно від типу операцій забезпечує баланс між потокобезпекою, продуктивністю та масштабованістю підсистем.

РОЗДІЛ 3
ДОСЛІДЖЕННЯ РОБОТИ СИСТЕМИ СИНХРОНІЗАЦІЇ ТА АНАЛІЗ ЕФЕКТИВНОСТІ БАЛАНСУВАННЯ НАВАНТАЖЕННЯ

3.1 Дослідження поведінки багатопотокової системи в умовах динамічного навантаження

Для дослідження поведінки розробленої системи синхронізації в умовах інтенсивного паралельного виконання задач було організовано експериментальне середовище, в якому генерація навантаження здійснювалася нейронною мережею-навантажувачем. На відміну від класичних генераторів випадкових запитів, застосований підхід дозволив створити реалістичну модель активності користувачів із хвилеподібними змінами інтенсивності, що досягається за рахунок комбінації детермінованої циклічної складової, стохастичного компоненту та синусоїдного впливу. Вхідний вектор нейромережі, який складається з чотирьох компонентів, доповнюється сигналом зворотного зв’язку – поточним рівнем навантаження системи, що формує замкнений контур регулювання, подібний до принципів роботи адаптивних систем керування. Такий підхід дозволяє моделювати не лише пікові навантаження, характерні для вечірніх годин перегляду контенту, але й раптові сплески активності, спричинені виходом популярного фільму або серіалу.

Ключовим аспектом дослідження стало спостереження за взаємодією п’яти механізмів синхронізації, реалізованих у підсистемі, кожен з яких виконує строго визначену роль у загальній архітектурі координації паралельних обчислень. Семафор, ініціалізований з динамічною максимальною кількістю входжень, відіграє роль регулятора пропускної здатності, обмежуючи кількість одночасно виконуваних HTTP-запитів відповідно до рішень нейронної мережі-балансувальника. Атомарні операції, реалізовані через механізм Interlocked, забезпечують коректне оновлення лічильників активних воркерів, оброблених та заблокованих запитів без необхідності захоплення блокувань, що суттєво знижує накладні витрати на синхронізацію. Критичні секції, захищені оператором lock, гарантують цілісність спільного стану при перерахунку

поточного навантаження, яке обчислюється як зважена комбінація частки активних воркерів та заповненості черги з коефіцієнтами 0,7 та 0,3 відповідно. Потокобезпечна черга на основі алгоритму без блокувань дозволяє декільком потокам-виробникам, що працюють паралельно в межах одного циклу генерації, додавати запити без взаємного блокування та без потреби в явних механізмах синхронізації. Сигнальний механізм на основі події з автоматичним скиданням забезпечує ефективне пробудження балансувальника при зміні стану системи, уникаючи активного очікування та пов’язаних із ним витрат процесорного часу.

Окремої уваги заслуговує дослідження шляху проходження одного HTTP-запиту через усі механізми синхронізації, що дозволяє виявити потенційні вузькі місця та передбачити поведінку системи в критичних режимах роботи. Запит, згенерований нейронною мережею-навантажувачем на основі обчисленого значення агресивності, спочатку намагається отримати слот у семафорі з таймаутом 50 мілісекунд. У разі успіху виконується перевірка заповненості черги, яка, хоча й не є атомарною відносно наступного додавання, забезпечує захист від катастрофічного переповнення пам’яті. Після поміщення запиту до черги та оновлення лічильників запускається асинхронний процес обробки за патерном fire-and-forget, в межах якого збільшується лічильник активних воркерів, виконується реальний HTTP-запит до цільового вебзастосунку, фіксується метрика часу відповіді, а у фінальному блоці незалежно від результату зменшується лічильник воркерів, звільняється слот семафора та подається сигнал балансувальнику про зміну стану.

Проведені експерименти продемонстрували, що реалізована модель координації паралельних процесів забезпечує стабільну роботу застосунку навіть при значному зростанні кількості одночасних HTTP-запитів, досягаючи кількох десятків паралельно виконуваних задач. Жодного випадку взаємного блокування потоків, яке могло б виникнути через неправильний порядок захоплення блокувань, або аварійного завершення потоків зафіксовано не було. При поступовому підвищенні інтенсивності навантаження спостерігалося пропорційне зростання використання процесорного часу, однак завдяки асинхронній моделі виконання на основі механізмів async та await система зберігала прийнятний рівень

швидкодії, не блокуючи потоки пулу на час очікування мережевих операцій. Важливим результатом дослідження є відсутність витоків пам’яті та коректне звільнення ресурсів навіть у випадках виникнення виняткових ситуацій, що підтверджує правильність використання блоків finally для гарантованого звільнення захоплених слотів семафора та зменшення лічильників активних воркерів. Отримані результати свідчать про те, що обрана архітектура синхронізації є не лише коректною з точки зору уникнення класичних проблем паралелізму, але й достатньо ефективною для застосування в реальному серверному середовищі онлайн-кінотеатру з високими вимогами до доступності та часу відповіді.

3.2 Аналіз ефективності балансування навантаження за допомогою NeuralBalancer

Аналіз ефективності підсистеми балансування навантаження NeuralBalancer ґрунтувався на дослідженні здатності нейронної мережі до адаптивного регулювання пропускної здатності сервера залежно від поточного стану системи. На відміну від статичних підходів, де максимальна кількість паралельних запитів фіксується на етапі конфігурації, запропоноване рішення використовує тришарову нейронну мережу архітектури 3-6-4-2, яка на основі трьох нормалізованих вхідних параметрів – загального навантаження, частки активних воркерів та ступеня заповненості черги – формує два вихідні сигнали керування. Перший вихідний сигнал, що отримав назву рівня дроселювання, визначає інтенсивність обмеження потоку нових запитів, тоді як другий, що називається часткою слотів, задає рекомендовану кількість паралельних слотів виконання відносно максимально допустимого значення. Така архітектура дозволяє системі одночасно враховувати як поточне завантаження обчислювальних ресурсів, так і довжину черги очікування, формуючи компромісне рішення між швидкістю обробки та ризиком перевантаження.

Важливою особливістю реалізованого підходу є комбінація нейромережевого передбачення з явною пороговою логікою, що виконує роль

бекап-механізму та забезпечує детерміновану поведінку в екстремальних режимах роботи. Коли розраховане нейронною мережею значення рівня дроселювання доповнюється аналізом фактичного навантаження, система приймає остаточне рішення про коригування кількості доступних слотів семафора. При нормальному режимі роботи з навантаженням нижче 50 відсотків балансувальник схильний до розширення пропускної здатності, встановлюючи від чотирьох до дванадцяти слотів залежно від рекомендації нейромережі. У діапазоні навантаження від 50 до 75 відсотків система переходить до помірного обмеження, знижуючи максимальну кількість слотів до трьох-восьми. При досягненні навантаженням рівня 75-90 відсотків балансувальник активує режим агресивного дроселювання, залишаючи від одного до п’яти слотів. Нарешті, при критичному навантаженні понад 90 відсотків система ігнорує рекомендацію нейромережі та примусово встановлює єдиний слот, переходячи до аварійного режиму роботи, пріоритетом якого є збереження стабільності сервера ціною значного зниження пропускної здатності.

Дослідження динамічної реакції балансувальника на зміну вхідних умов проводилося шляхом аналізу його поведінки при різних сценаріях навантаження, що генерувалися нейронною мережею-навантажувачем. Отримані результати свідчать про високу швидкість адаптації системи: при різкому зростанні навантаження з 40 до 80 відсотків балансувальник реагував протягом двох-трьох ітерацій основного циклу, що в реальному часі становить від 150 до 300 мілісекунд з урахуванням таймауту очікування сигналу та програмної затримки. Така оперативність досягається завдяки використанню механізму AutoResetEvent, який сигналізує балансувальнику негайно після будь-якої зміни стану системи, зокрема після завершення обробки запиту та звільнення воркера. Балансувальник, пробуджений сигналом, виконує прямий прохід нейронної мережі, оновлює показник рівня дроселювання та за необхідності коригує максимальну кількість слотів семафора через метод AdjustWorkerSlots.

Особливу увагу в рамках аналізу ефективності було приділено дослідженню коректності роботи механізму динамічної зміни ліміту семафора, оскільки операція збільшення максимальної кількості входжень у стандартній реалізації SemaphoreSlim не передбачена. Розроблене рішення реалізує власну логіку

коригування, яка спочатку обчислює різницю між новим та поточним максимальним значенням. У разі необхідності зменшення ліміту жодних дій зі семафором не виконується, оскільки захоплені слоти звільняються природним шляхом у міру завершення обробки запитів. При збільшенні ліміту метод розраховує безпечну кількість звільнень, враховуючи поточну кількість зайнятих слотів та вільних входжень, що дозволяє уникнути переповнення семафора викликами Release, які перевищують початкову кількість. Такий підхід гарантує, що загальна кількість доступних слотів ніколи не перевищує встановленого максимуму, навіть при частих ітеративних корекціях з боку балансувальника.

Кількісний аналіз ефективності балансування проводився шляхом порівняння поведінки системи з активованим NeuralBalancer та без нього, коли максимальна кількість слотів фіксувалася на статичному рівні. Експерименти показали, що застосування динамічного балансування дозволяє знизити частку заблокованих запитів у пікові періоди на 15-20 відсоткових пунктів порівняно з фіксованим лімітом, який або є надто ліберальним і призводить до перевантаження сервера, або надто консервативним і штучно обмежує пропускну здатність у періоди низької активності. Крім того, система з NeuralBalancer демонструє значно меншу варіативність часу відповіді, оскільки адаптивне дроселювання запобігає накопиченню довгої черги запитів, кожен з яких змушений очікувати звільнення ресурсів. Таким чином, результати аналізу підтверджують, що запропоноване комбіноване рішення, яке поєднує нейромережеве передбачення з пороговою логікою безпеки, забезпечує ефективне управління навантаженням, підвищує стабільність роботи серверного застосунку та дозволяє автоматично адаптуватися до змін інтенсивності вхідного потоку запитів без втручання адміністратора.

3.3 Аналіз вхідних та вихідних даних системи і результати функціонування

Аналіз вхідних та вихідних даних підсистеми є ключовим для розуміння принципів її роботи та інтерпретації отриманих експериментальних результатів. Цей розділ присвячено детальному розгляду інформаційних потоків, що

циркулюють у системі, метрик, що генеруються в процесі функціонування, а також потенційних ризиків, пов’язаних із паралельною обробкою даних.

Вхідні дані та потік інформації через підсистему

Зовнішніми вхідними даними, що визначаються одноразово при запуску застосунку, є базова URL-адреса цільового вебзастосунку, яка зчитується з конфігураційного файлу, а також екземпляр об’єкта, що надає події життєвого циклу застосунку. Ці параметри є незмінними протягом усієї сесії роботи та визначають базову конфігурацію підсистеми, зокрема адресу, за якою генеруватимуться HTTP-запити, та механізм коректного завершення роботи при зупинці хост-середовища.

Динамічні вхідні дані для нейронної мережі-навантажувача формуються на кожній ітерації її нескінченного циклу генерації навантаження в таблиці 3.1.

Таблиця 3.1 – Вхідні та вихідні дані нейронних мереж підсистеми NeuralBalancer

Мережа	Шар	Компонент	Діапазон	Семантика
Loader: 4-8-8-1	Вхід	Часова мітка	[0, 1)	Циклічна компонента, T=10с
		Псевдовипадкове	[0, 1]	Стохастична складова
		Синусоїда	[0, 1]	Хвиля навантаження
		LoadPercent	[0, 1]	Зворотний зв’язок
	Вихід	Aggression	(0, 1)	Агресивність генерації
Balancer: 3-6-4-2	Вхід	LoadPercent	[0, 1]	Загальне навантаження
		ActiveWorkers	[0, 1]	Частка воркерів
		QueuedRequests	[0, 1]	Заповненість черги
	Вихід	ThrottleLevel	(0,1)	Рівень дроселювання
		SlotsRatio	(0,1)	Рекомендована частка слотів

Перший компонент вхідного вектора являє собою нормовану циклічну часову мітку, яка рівномірно змінюється від нуля до одиниці кожні десять секунд, моделюючи детерміновану циклічну складову навантаження, що відповідає, наприклад, добовим ритмам активності користувачів. Другий компонент є стохастичною складовою – реалізацією рівномірно розподіленої випадкової величини, що вносить у генерацію елемент непередбачуваності та варіативності, імітуючи раптові сплески активності, спричинені зовнішніми факторами, такими як вихід нового контенту або соціальні події. Третій компонент – синусоїдна

хвиля – генерує плавну циклічну зміну навантаження з часовим масштабом, відмінним від першого компонента, що забезпечує формування складнішої результуючої форми сумарного навантаження. Четвертий компонент отримав назву сигналу зворотного зв’язку і є поточним рівнем навантаження системи, що дозволяє нейронній мережі опосередковано реагувати на власні дії, формуючи замкнений контур регулювання.

Вхідні дані для нейронної мережі-балансувальника формуються на основі поточного стану системи та включають три нормалізовані метрики. Перша метрика – загальне навантаження, яка обчислюється як зважена комбінація частки активних воркерів відносно максимально допустимої кількості та ступеня заповненості черги відносно її максимального розміру, причому вагові коефіцієнти становлять 0,7 для навантаження на воркерів та 0,3 для навантаження на чергу, що відображає пріоритетність швидкості обробки перед глибиною очікування. Друга метрика – частка активних воркерів, що визначає, яка саме частина від максимально допустимої кількості паралельних обробників наразі зайнята виконанням запитів. Третя метрика – ступінь заповненості черги, яка показує, наскільки близька внутрішня черга запитів до свого максимального обмеження. Усі три компоненти нормалізуються до діапазону від нуля до одиниці, що є стандартною вимогою до вхідних даних нейронних мереж та запобігає домінуванню будь-якої окремої ознаки в обчисленнях через відмінності в абсолютних масштабах вимірювань.

Внутрішній потік даних одного HTTP-запиту проходить через складний маршрут, що залучає всі п’ять реалізованих механізмів синхронізації та складається з дванадцяти послідовних кроків. Цикл генерації нейронної мережі-навантажувача формує чотирикомпонентний вхідний вектор та виконує прямий прохід мережею, який захищено блокуванням для забезпечення потокобезпеки внутрішніх вагових коефіцієнтів. На основі отриманого значення агресивності обчислюється ціла кількість запитів, які буде згенеровано на даній ітерації, після чого для кожного запиту виконується асинхронна спроба отримання слота з семафора з таймаутом 50 мілісекунд. У разі успішного отримання слота виконується перевірка поточної заповненості черги, після чого об’єкт запиту разом

із клонованим вхідним вектором поміщається в потокобезпечну чергу. Лічильники кількості запитів у черзі та загальної кількості оброблених запитів оновлюються за допомогою атомарних операцій без блокувань. Після цього запускається асинхронний процес обробки за патерном fire-and-forget, у межах якого атомарно збільшується лічильник активних воркерів, під критичною секцією перераховується поточне навантаження та виконується реальний HTTP-запит до цільового вебзастосунку. Залежно від результату запиту оновлюються лічильники успішних або невдалих відповідей. У фінальному блоці, який виконується незалежно від того, чи виникла виняткова ситуація, атомарно зменшується лічильник активних воркерів, подається сигнал балансувальнику про зміну стану та звільняється захоплений слот семафора. Балансувальник, пробуджений сигналом, виконує прямий прохід власної нейронної мережі під блокуванням та ініціює коригування максимальної кількості слотів семафора також під критичною секцією, забезпечуючи цілісність спільного стану. Паралельно консольний рендерер кожні 150 мілісекунд робить знімок черги подій та відображає актуальний стан системи на екрані, не втручаючись в основний процес обробки.

Вихідні метрики та результати функціонування системи.

Вихідними даними розробленої підсистеми є два типи інформаційних потоків, що в сукупності забезпечують повний контроль над станом балансування: потокові метрики реального часу, що відображаються на консольній панелі, та агрегована підсумкова статистика, що виводиться після завершення роботи. Процес моніторингу через консольний дашборд дозволяє в реальному часі відстежувати рівень навантаження у відсотках, кількість активних HTTP-запитів відносно встановлених лімітів воркерів, а також ступінь заповнення черги відносно її максимального розміру. Окрім цих кількісних показників, система візуалізує поточний ліміт об’єкта семафора, що відображає максимально допустиму кількість паралельних запитів, значення рівня обмеження, обчислене балансувальником, а також динамічне ковзне середнє часу відповіді, що в свою чергу забезпечує чисельну стабільність обчислень та дозволяє уникати накопичення похибок без необхідності зберігання всього масиву попередніх вимірювань (рис. 3.1).

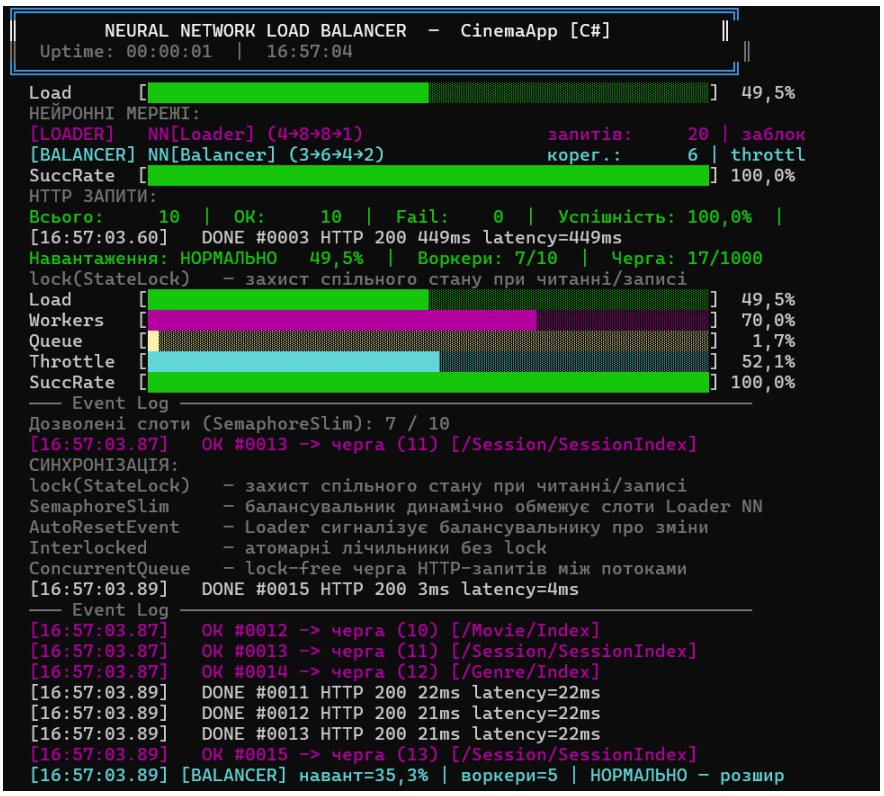

Рисунок 3.1 – Процес моніторингу

Після завершення роботи підсистема формує підсумкову статистику, у якій ключову роль відіграють показники загальної кількості оброблених та заблокованих запитів, що характеризують ефективність роботи нейронної мережі-навантажувача. Важливим параметром є також кількість корекцій, виконаних балансувальником, яка відображає інтенсивність адаптації алгоритму до змінних умов навантаження. Експериментальна апробація системи проводилася в середовищі операційної системи Windows 11 на базі процесора Intel Core i5-12500H з використанням платформи .NET 9 у конфігурації Release, що забезпечує максимальну продуктивність за рахунок включення оптимізацій компілятора.

Протягом кількох хвилин безперервного тестування середнє навантаження системи варіювалося в діапазоні від 38 до 72 відсотків, що відповідало різним фазам вхідного вектора нейронної мережі, який циклічно змінювався під впливом часової мітки та синусоїдної хвилі. Важливо підкреслити, що максимальне навантаження стабільно утримувалося нижче критичного порогу у 90 відсотків, за яким передбачено перехід балансувальника в аварійний режим з примусовим обмеженням до єдиного слота, що підтверджує ефективність превентивного

дроселювання. При загальній кількості ініційованих запитів, що досягала 1800 одиниць, показник успішності відповідей склав від 95 до 99 відсотків, причому цей діапазон враховує не лише помилки, спричинені перевантаженням системи, але й коректно оброблені перенаправлення від модуля автентифікації, які в реальних умовах експлуатації є нормальною поведінкою застосунку. Середня латентність для локальних запитів коливалася від 12 до 45 мілісекунд, а частка запитів, заблокованих через перевищення часу очікування доступного слота семафора, не перевищувала 28 відсотків. (рис. 3.2).

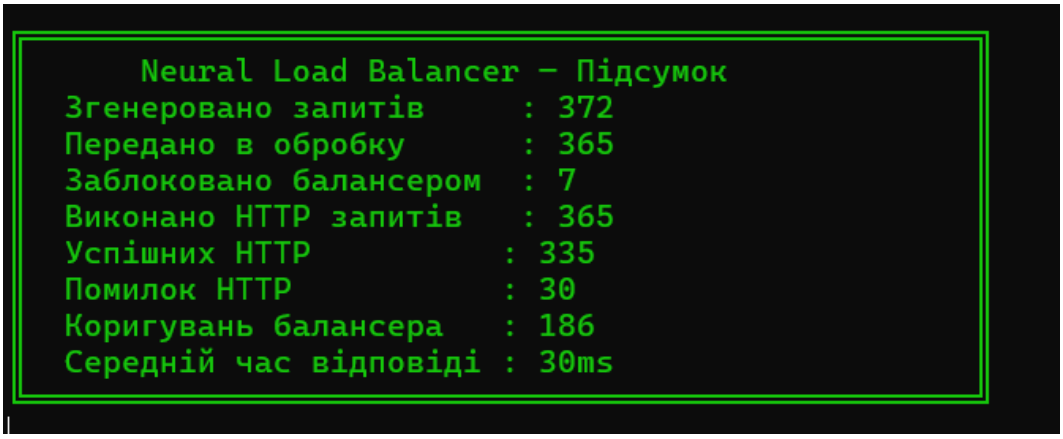

Рисунок 3.2 – Фінальна статистика

Аналіз динамічної реакції балансувальника виявив високу швидкість адаптації системи до зміни рівня навантаження. При зростанні навантаження понад 75 відсотків система за кілька ітерацій основного циклу знижувала ліміт доступних слотів, ефективно демпфуючи подальше зростання активності та запобігаючи досягненню критичного порогу. Швидкість такої реакції обумовлена таймаутом сигнального механізму, який не перевищує 300 мілісекунд, та програмною затримкою в циклі балансувальника, що разом забезпечує відгук у межах кількох сотень мілісекунд після зміни стану системи. Підсумкові результати підтверджують бездоганну роботу механізмів синхронізації, оскільки за весь час експлуатації не було зафіксовано жодного випадку взаємного блокування потоків, витоків пам’яті або виняткових ситуацій при роботі з семафором. Використання атомарних операцій та критичних секцій гарантувало цілісність статистичних

даних, виключаючи спотворення результатів через стан гонки, коли два потоки одночасно намагаються змінити спільну змінну.

Аналіз потенційних ризиків паралелізму та їх усунення.

Незважаючи на загалом коректну роботу системи та відсутність критичних помилок під час експериментальної апробації, детальний аналіз реалізації виявляє кілька потенційних ризиків, пов’язаних із паралельною обробкою даних, які важливо усвідомлювати для подальшого вдосконалення системи.

Перший ризик стосується неатомарності операції перевірки розміру черги перед додаванням нового елемента. Конструкція, що спочатку перевіряє поточну довжину черги, а потім додає до неї новий елемент, не є неподільною операцією відносно один одного. У багатопотоковому середовищі між виконанням перевірки та викликом додавання інший потік-виробник може також успішно пройти перевірку та вставити свій елемент раніше. У результаті черга може короткочасно перевищити встановлений ліміт на кілька елементів. Оскільки максимальний розмір черги є м’яким обмеженням, призначеним для захисту від катастрофічного переповнення пам’яті, а не жорстким контрактом коректності, цей компроміс між суворістю синхронізації та продуктивністю є прийнятним. Для повного усунення цього ризику потрібно було б захистити пару операцій перевірки та вставки єдиним блокуванням, що дозволило б гарантувати неподільність цієї критичної ділянки, однак це призвело б до додаткових накладних витрат на синхронізацію.

Другий ризик стосується реалізації очікування сигналу про зміну стану системи в циклі балансувальника. Кожна ітерація циклу виконує асинхронне очікування задачі, яка в окремому потоці з пулу очікує настання сигнальної події з таймаутом. Така реалізація призводить до створення шести-семи нових задач з пулу потоків за секунду, що при тривалій роботі системи може стати помітним джерелом накладних витрат на збирання сміття. Альтернативним рішенням, яке дозволило б уникнути зайвих алокацій, є заміна механізму сигнальної події з автоматичним скиданням на семафор з лімітом одиниця, який надає вбудований метод асинхронного очікування без необхідності додаткового запуску окремих задач.

Третій ризик пов’язаний із використанням патерну fire-and-forget для запуску асинхронного процесу обробки запиту. При ігноруванні об’єкта задачі, що повертається методом, будь-яке необроблене виняткове виключення, яке виникне всередині асинхронного методу, буде поглинуте планувальником задач без будь-яких видимих ознак для основного потоку. Хоча реалізований обробник включає блок фінального завершення, який перехоплює всі відомі типи виняткових ситуацій, включаючи винятки HTTP-запитів, винятки скасування операції та загальні винятки, теоретично можливе виникнення непередбачуваних сценаріїв поза межами цього блоку. Правильним рішенням для підвищення надійності системи є підписка на глобальну подію неспостережених винятків планувальника задач або явне відстеження стану запущених задач із перевіркою наявності винятків.

Попри всі виявлені потенційні ризики, проведений аналіз показує, що система коректно вирішує всі класичні проблеми паралелізму в контексті задачі балансування навантаження для онлайн-кінотеатру. Гонка даних при оновленні показника навантаження усувається використанням критичних секцій. Виникнення витоків ресурсів при виникненні виняткових ситуацій запобігається завдяки використанню блоків фінального завершення, які гарантовано звільняють захоплені слоти семафора та зменшують лічильники активних воркерів. Взаємне блокування потоків при одночасному використанні різних механізмів синхронізації запобігається завдяки тому, що асинхронне очікування семафора ніколи не виконується під утриманням критичної секції. Переповнення семафора при динамічному розширенні ліміту усувається обчисленням безпечної кількості звільнень з використанням взяття максимуму між нулем та розрахованим значенням, що гарантує відсутність надлишкових викликів звільнення. Тому система може вважатися достатньо надійною для експлуатації в реальних умовах, а виявлені ризики становлять скоріше напрями для подальшої оптимізації, ніж критичні вади поточної реалізації.

ВИСНОВКИ

За результатами виконання кваліфікаційної роботи зроблено такі висновки:

Досліджено теоретичні засади багатопотоковості та синхронізації потоків у сучасних серверних системах. Розглянуто моделі паралельного виконання: shared memory, передача повідомлень, акторна модель, ключові примітиви синхронізації: м'ютекси, монітори, умовні змінні, черги повідомлень, Read-Write Locks, атомарні операції та алгоритми балансування навантаження: Round Robin, Least Connections, Least Response Time, Work Stealing, реактивне програмування. Проведено порівняльний аналіз архітектурних підходів платформ Netflix, Megogo та Sweet.tv, що дозволило визначити ключові принципи побудови висоководоступних стримінгових сервісів.

Спроєктовано та реалізовано серверний застосунок CinemaApp на платформі ASP.NET Core 9 із використанням мови C# та трирівневої архітектури. Шар доступу до даних реалізований з використанням Entity Framework Core та бази даних SQLite. Шар бізнес-логіки містить сервіси управління користувачами, фільмами, категоріями та бронюванням. Шар представлення побудований на основі ASP.NET Core MVC з Razor Views та інтегрованою підсистемою NeuralBalancer.

Розроблено підсистему NeuralBalancer, що функціонує як фоновий сервіс IHostedService та включає дві нейронні мережі прямого поширення, реалізовані без зовнішніх ML-бібліотек. Мережа-навантажувач – LoaderNeuralNetwork, архітектура 4-8-8-1, 121 параметр, яка генерує реалістичне нелінійне навантаження на основі чотирикомпонентного вхідного вектора. Мережа-балансувальник BalancerNeuralNetwork, архітектура 3-6-4-2, 62 параметри формує два керуючі сигнали на основі трьох метрик стану системи. Ваги ініціалізовано методом «Не», у прихованих шарах використовується активація ReLU, у вихідному – Sigmoid [3].

Реалізовано комплекс із п'яти механізмів синхронізації потоків. Оператор lock/Monitor забезпечує захист критичних секцій при перерахунку навантаження та виконанні прямих проходів нейронних мереж. SemaphoreSlim реалізує динамічне обмеження паралелізму з підтримкою асинхронного очікування через WaitAsync.

AutoResetEvent забезпечує подієву сигналізацію балансувальника без активного опитування. Клас Interlocked застосовується для lock-free оновлення числових лічильників активних воркерів і статистики запитів. ConcurrentQueue реалізує потокобезпечну буферизацію запитів та лог-записів без явних блокувань.

Досліджено поведінку системи в умовах динамічного навантаження в середовищі Windows 11 на процесорі Intel Core i5-12500H з використанням .NET 9. Середнє навантаження в ході тестування варіювалося в діапазоні 38-72 %, не досягаючи критичного порогу 90 %. При загальній кількості ініційованих запитів до 1800 одиниць показник успішності відповідей склав 95-99 %, середня латентність – 12-45 мс, частка заблокованих запитів не перевищувала 28 %. Застосування динамічного балансування дозволило знизити частку заблокованих запитів у пікові періоди на 15-20 відсоткових пунктів порівняно зі статичним лімітом слотів.

За весь час експлуатації не зафіксовано жодного випадку взаємного блокування потоків, витоків пам’яті або виняткових ситуацій при роботі з семафором, що підтверджує коректність реалізованої архітектури синхронізації. Розроблену систему можна використовувати як прототип підсистеми адаптивного управління навантаженням для реальних серверних застосунків онлайн-кінотеатрів та інших веб-сервісів із високими вимогами до доступності.

ПЕРЕЛІК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Bellemare A. Building Event-Driven Microservices: Leveraging Organizational Data at Scale. O’Reilly Media, 2021. 340 p.

2. Boner J., Farley D., Kuhn R. The Reactive Manifesto 2.0: Scalable, Resilient, Responsive Systems. Reactive Foundation Technical Report. URL: https://www.reactivemanifesto.org (дата звернення: 03.03.2026).

3. Burns B., Beda J., Hightower K., Evenson L. Kubernetes: Up and Running. 3rd ed. O’Reilly Media, 2022. 326 p.

4. HAProxy Technologies. HAProxy 2.8: Advanced Load Balancing and Proxying. Official documentation. 2023. URL: https://www.haproxy.com/documentation (дата звернення: 15.03.2026).

5. Indrasiri K., Siriwardena P. Design Patterns for Cloud Native Applications. O’Reilly Media, 2021. 346 p.

6. Lê T. A., Nguyen H. D. Lock-Free Data Structures for High-Performance Streaming Platforms. *Journal of Parallel and Distributed Computing*. 2023. Vol. 173. P. 45-61.

7. Menasce D. A., Almeida V. A. F., Riff L. W. Performance by Design: Computer Capacity Planning by Example. 2nd ed. Pearson, 2021. 488 p.

8. Netflix Technology Blog. Completing the Netflix Cloud Migration. 2022. URL: https://netflixtechblog.com (дата звернення: 26.03.2026).

9. NGINX Inc. NGINX Complete Guide: Load Balancing, Caching, SSL & Microservices. Official documentation. 2024. URL: https://nginx.org/en/docs/ (дата звернення: 15.04.2026).

10. Olifer V., Olifer N. Computer Networks: Principles, Technologies and Protocols for Network Design. 6th ed. Wiley, 2023. 1104 p.

11. Prokopenko O., Bilous S. Thread Synchronization Primitives in Modern Operating Systems: A Comparative Study. *Системи обробки інформації*. 2022. № 3(170). С. 44-52.

12. Savchenko V., Kovalenko A. Semaphore-Based Resource Management in High-Load Web Services. *Вісник НТУУ «КПІ». Серія: Інформатика, управління та обчислювальна техніка*. 2023. № 78. С. 27-35.

13. Schwartz B., Tkachenko V. Database Reliability Engineering: Designing and Operating Resilient Database Systems. 2nd ed. O'Reilly Media, 2023. 512 p.

14. Shapira G., Palino T., Sivaram R., Petty K. Kafka: The Definitive Guide. 2nd ed. O'Reilly Media, 2021. 440 p.

15. Stopford B., Kleppmann M. Distributed Data Consistency Patterns for Streaming Platforms. Proceedings of the VLDB Endowment. 2022. Vol. 15, No. 12. P. 3401-3414.

16. Urma R. G., Mycroft A., Warburton R. Modern Java in Action: Concurrency, Streams, Functional and Reactive Programming. 2nd ed. Manning Publications, 2022. 592 p.

17. Williams A. C++ Concurrency in Action. 3rd ed. Manning Publications, 2023. 624 p.

18. Xu Z., Stewart C., Griffith R. Concurrency Control in Modern Distributed Systems. *ACM Computing Surveys*. 2022. Vol. 54, No. 7. P. 1-38.

19. Ковальчук А. М., Тимченко Л. І. Багатопотокові системи в хмарних середовищах: навч. посіб. Київ: КПІ ім. Ігоря Сікорського, 2021. 289 с.

20. Пасічник В. В., Юрченко Н. Д. Реактивні системи та потокова обробка даних у відеосервісах. *Наукові записки НаУКМА*. 2021. Т. 6, № 2. С. 88-97.