Міністерство освіти і науки України
Харківський національний університет радіоелектроніки

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

КВАЛІФІКАЦІЙНА РОБОТА
Пояснювальна записка

рівень вищої освіти другий (магістерський)

ДОСЛІДЖЕННЯ МЕТОДІВ ОПТИМІЗАЦІЇ ВИСОКОНАВАНТАЖЕНИХ ПРОЄКТІВ
(тема)

Виконав:
здобувач 2 року навчання,
групи ІНФм-24-2
Олейник М.В.
(прізвище, ініціали)

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

Тип програми освітньо-професійна

Освітня програма Інформатика
(повна назва освітньої програми)

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

Допускається до захисту

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

2025 р.

Харківський національний університет радіоелектроніки

Факультет Інформаційно-аналітичних технологій та менеджменту

Кафедра Інформатики

Рівень вищої освіти другий (магістерський)

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

Тип програми освітньо-професійна

Освітня програма Інформатика
(повна назва)

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

ЗАВДАННЯ
НА КВАЛІФІКАЦІЙНУ РОБОТУ

здобувачеві Олейнику Михайлу Володимировичу
(прізвище, ім'я, по батькові)

1. Тема роботи Дослідження методів оптимізації високонавантажених проєктів

затверджена наказом університету від 14 листопада 2025 року № 1045Ст
2. Термін подання здобувачем роботи до екзаменаційної комісії 8 грудня 2025 р.
3. Вихідні дані до роботи науково-технічна література, матеріали наукових конференсій, середовище розробки WebStorm, обгортка до мови TypeScript

4. Перелік питань, що потрібно опрацювати в роботі
1. Аналіз сучасних методів оптимізацій високонавантажених проєктів.
2. Аналіз літературних джерел щодо апробації методів оптимізації високонавантажених проєктів.
3. Формування покрокового алгоритму для кожного із вибраних методів оптимізації.
4. Проєктування тестового застосунку.
5. Розробка програмного застосунку, що надасть змогу протестувати теорію оптимізації системи кожним із вибраних методів.

5. Перелік графічного матеріалу із зазначенням креслеників, схем, плакатів, комп'ютерних ілюстрацій (п.5 включається до завдання за рішенням випускової кафедри) актуальність проблеми високонавантажених систем, об'єкт та мета дослідження, постановка задачі, блок-схеми алгоритмів вибраних методів класифікації, приклад рішень для оптимізації високонавантажених систем, ілюстрація головного екрану розробленого застосунку із можливістю вибору графіків результатів, ілюстрація результатів тесту оптимізації, висновки, перспективи та апробація роботи.

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

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

КАЛЕНДАРНИЙ ПЛАН

№ з/п	Назва етапів роботи	Строк / терміни виконання етапів роботи	Примітка
1	Отримання завдання на кваліфікаційну роботу	29.09.2025	
2	Аналіз завдання, підбір літератури	30.09.25-07.10.25	
3	Аналіз літератури з досліджуваної проблеми	08.10.25-14.10.25	
4	Особливості архітектур	15.10.25-20.10.25	
5	Дослідження методів оптимізації	21.10.25-27.10.25	
6	Програмна реалізація	28.10.25-05.11.25	
7	Обґрунтування отриманих результатів	06.11.25-11.11.25	
8	Оформлення пояснювальної записки	12.11.25-14.11.25	
9	Перевірка на нормоконтроль	19.11.25-10.12.25	
10	Перевірка на плагіат	12.12.2025	
11	Рецензування	13.12. 2025	
12	Підготовка презентації та доповіді	14.12. 2025	
13	Занесення роботи в електронний архів	17.12. 2025	
14	Попередній захист кваліфікаційної роботи	17.12. 2025	

Дата видачі завдання 29 вересня 2025 р.

Здобувач _____
(підпис)

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

РЕФЕРАТ

Пояснювальна записка до кваліфікаційної роботи: 69 с., 3 табл., 11 рис., 26 джерел.

АСИНХРОННА ОБРОБКА, БАГАТОПОТОЧНІСТЬ, ВИСОКОНАВАНТАЖЕНІ СИСТЕМИ, ГОРИЗОНТАЛЬНЕ МАСШТАБУВАННЯ, МАСШТАБОВАНІСТЬ, НАВАНТАЖУВАЛЬНЕ ТЕСТУВАННЯ, ПРОДУКТИВНІСТЬ, BULLMQ, DOCKER, GRAPHQL, K6, NEST.JS, NODE.JS, P99 LATENCY, REDIS, REST API, WORKER THREADS

Об’єктом дослідження є продуктивність високонавантажених проєктів в умовах критичних пікових навантажень та обмежених ресурсів.

Метою роботи є розробка та реалізація комплексної стратегії оптимізації, що базується на синергії асинхронних патернів, паралельних обчислень та сучасних протоколів передачі даних.

Наукова новизна отриманих результатів полягає в обґрунтуванні гібридного архітектурного підходу, який синтезує механізми персистентних асинхронних черг та виділених робочих потоків (worker threads). Доведено, що така інтеграція в межах єдиного фреймворку дозволяє ефективно розмежувати I/O-bound та CPU-bound операції, нівелюючи проблему блокування циклу подій (Event Loop).

Практичним результатом стала розроблена та протестована горизонтально масштабована архітектура на базі стеку Nest.js, Docker та Nginx. Було реалізовано програмний застосунок, що демонструє переваги патерну «Cache-Aside» з використанням Redis та гарантовану доставку результатів через WebSockets. Запропоноване рішення забезпечує високу доступність (High Availability), відмовостійкість та низьку затримку відповіді, що підтверджено результатами експериментального порівняння з традиційним синхронним підходом.

ABSTRACT

Explanatory note to the qualification work: 69 pages, 3 tables, 11 figures, 26 sources.

ASYNCHRONOUS PROCESSING, MULTIPLE STREAMS, HIGH-PERFORMANCE SYSTEMS, HORIZONTAL SCALING, SCALABILITY, LOAD TESTING, PRODUCTIVITY, BULLMQ, DOCKER, GRAPHQL, K6, NEST.JS, NODE.JS, P99 LATENCY, REDIS, REST API, WORKER THREADS.

The object of the study is the processes of performance optimization of high-load projects under conditions of critical peak loads and limited resources.

The purpose of the work is the development and implementation of a comprehensive optimization strategy based on the synergy of asynchronous patterns, parallel computing, and modern data transfer protocols.

The scientific novelty of the obtained results lies in the substantiation of a hybrid architectural approach that synthesizes mechanisms of persistent asynchronous queues and dedicated worker threads. It is proven that such integration within a single framework allows for the effective separation of I/O-bound and CPU-bound operations, thereby eliminating the problem of Event Loop blocking.

The practical result is a developed and tested horizontally scalable architecture based on the Nest.js, Docker, and Nginx stack. A software application was implemented to demonstrate the advantages of the "Cache-Aside" pattern using Redis and the guaranteed delivery of results via WebSockets. The proposed solution ensures High Availability, fault tolerance, and low response latency, which is confirmed by the results of an experimental comparison with the traditional synchronous approach.

ЗМІСТ

ПЕРЕЛІК УМОВНИХ ПОЗНАЧЕНЬ, СИМВОЛІВ, ОДИНИЦЬ, СКОРОЧЕНЬ І ТЕРМІНІВ

API – Application Programming Interface (інтерфейс прикладного програмування)

CPU – Central Processing Unit (центральний процесор)

DI – Dependency Injection (впровадження залежностей)

EDA – Event-Driven Architecture (подієво-орієнтована архітектура)

GC – Garbage Collector (збирач сміття)

HTTP – Hypertext Transfer Protocol (протокол передачі гіпертексту)

I/O – Input/Output (введення/виведення)

IoC – Inversion of Control (інверсія керування)

k6 – Інструмент для навантажувального тестування

KPI – Key Performance Indicators (ключові показники ефективності)

p99 – 99-й процентиль

p95 – 95-й процентиль

Pub/Sub – Publish/Subscribe (публікація/підписка)

QoS – Quality of Service (якість обслуговування)

REST – Representational State Transfer (передача репрезентативного стану)

RPS – Requests Per Second (запитів за секунду)

SLA – Service Level Agreement (угода про рівень обслуговування)

SLO – Service Level Objective (цільовий рівень обслуговування)

SPA – Single-Page Application (односторінковий вебзастосунок)

SRP – Single Responsibility Principle (принцип єдиної відповідальності)

ВСТУП

Сучасна цифрова інфраструктура, від платформ електронної комерції до соціальних мереж та IoT-сервісів, стикається з експоненційним зростанням навантаження. Зі збільшенням кількості одночасних користувачів, обсягів даних та складності бізнес-логіки традиційні синхронні підходи до обробки запитів стають неефективними. Постає гостра проблема оптимізації продуктивності високонавантажених проєктів, де ключовими інструментами стають перехід до асинхронної обробки, архітектурна децентралізація та паралелізація обчислень.

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

Сучасний стан проблеми характеризується наявністю великої кількості технологій та фреймворків, призначених для вирішення окремих аспектів оптимізації. До них належать брокери повідомлень (як RabbitMQ або Kafka), системи черг на базі Redis (зокрема, BullMQ), сучасні протоколи API (як GraphQL, що вирішує проблеми надлишкової та недостатньої вибірки даних REST), а також вбудовані в платформи механізми паралелізму.

Таким чином, дана робота спрямована на розв’язання актуальної задачі оптимізації продуктивності високонавантажених проєктів шляхом дослідження й порівняння методів асинхронної обробки, оптимізації API-взаємодії та паралельних обчислень. Це забезпечує її наукову новизну та практичну цінність для галузі сучасної веброзробки при проєктуванні масштабованих та відмовостійких систем.

1 АНАЛІЗ ПРЕДМЕТНОЇ ОБЛАСТІ

1.1 Архітектурний аналіз та патерни оптимізації високонавантажених застосунків

1.1.1 Визначення та ключові метрики високонавантажених систем

Побудова та підтримка високонавантажених систем вимагає фундаментального переходу від традиційних підходів до розробки. Недостатньо просто писати функціональний код; необхідно проєктувати системи, що здатні витримувати екстремальний трафік, зберігаючи при цьому низьку затримку, високу доступність та відмовостійкість. Цей розділ досліджує теоретичний фундамент та макро-архітектурні патерни, що слугують основою для оптимізації, аналізуючи ключові метрики, стратегії декаплінгу, оптимізацію протоколів взаємодії та методи паралельної обробки.

Високонавантажена система (high-load system) характеризується не стільки абсолютним числом запитів, скільки необхідністю підтримувати суворі показники якості обслуговування (QoS) в умовах значного та часто непередбачуваного трафіку. Для коректної оцінки та оптимізації таких систем використовуються специфічні ключові показники продуктивності (KPI) [1].

Основними метриками для вимірювання продуктивності застосунків є пропускна здатність і затримка (throughput і latency). Пропускна здатність відповідає за кількість операцій за одиницю часу, затримка – за час необхідний для обробки одного запису.

1.1.2 Критика середніх значень (Average Latency)

Традиційно, продуктивність часто вимірюють за допомогою середнього арифметичного значення затримки. Однак для високонавантажених систем ця метрика є не просто неточною, а й оманливою. Затримка в розподілених системах рідко має нормальний розподіл - вона, як правило, має "довгий хвіст"

(long-tailed distribution), де переважна більшість запитів обробляється швидко, але невелика частина – дуже повільно[2].

Середнє значення надзвичайно чутливе до викидів і приховує реальний досвід користувачів у "хвості". Розглянемо гіпотетичну систему, що обробила 10 000 запитів, при обробці такої кількості запитів на дистанції ми отримаємо приблизно такі цифри – 9400~ запитів буде за 50 мс., 500 запитів скажімо за 120 мс., 90 запитів за 600 і 10 за 8000 мс.

Розрахунок середнього значення дасть приблизно 118 мс, що сигналізує про проблеми. Проте медіанна затримка становить 50 мс, що вказує на те, що "типовий" користувач бачить швидку відповідь. Оптимізація, базована виключно на середньому значенні, змусила б інженерів шукати рідкісні аномалії, ігноруючи реальний досвід більшості.

1.1.3 Критична важливість процентилів (P95, P99)

Для отримання об'єктивної картини продуктивності використовуються процентилі, а саме 95-ий і 99-ий процентиль [3]. 95-й процентиль це значення затримки, яке не перевищують 95% запитів - типовий найгірший випадок, на який орієнтуються при налаштуванні продуктивності. 99-й процентиль в свою чергу це значення затримки, яке не перевищують 99% запитів - іншими словами, 99% запитів виконуються швидше за це значення, і лише 1% – повільніше.

У високонавантажених системах, що обслуговують мільйони запитів на хвилину, 1% – це величезна абсолютна кількість користувачів, які отримують поганий досвід. Якщо 99% затримка становить 5 секунд, це означає, що тисячі користувачів щохвилини чекають 5 секунд або довше [4].

Таким чином, фокус на 99% є фундаментальним зсувом у стратегії. Він означає пріоритетизацію надійності та передбачуваності сервісу для майже всіх користувачів, а не лише для "середнього". Метрика 99% викриває системні, хоч і нечасті, проблеми: "холодні старти" функцій, блокування баз даних, паузи GC,

перевантаження мережі – все те, що середнє значення повністю маскує. Саме тому 99% (або 99.9%) є стандартом.

1.2 Подієво-орієнтована архітектура (EDA) як стратегія декаплінгу

1.2.1 Подієво-орієнтована архітектура (Event-Driven Architecture, EDA)

Однією з ключових проблем масштабування монолітних або тісно пов'язаних систем є каскадні збої. Помилка в одному компоненті (наприклад, сервісі нотифікацій) може спричинити відмову основного бізнес-процесу (наприклад, реєстрації користувача) через жорсткі синхронні залежності. Подієво-орієнтована архітектура (Event-Driven Architecture, EDA) вирішує цю проблему шляхом фундаментального переосмислення комунікації між компонентами.

Концепція EDA базується на 3-х елементах – продюсер подій, споживач подій і медіатор [5]. Продюсер подій відповідає за генерацію подій, споживач – за реакцію на згенеровані події і медіатор являє собою посередник, що гарантує доставку подій від продюсерів до споживачів.

Основними перевагами EDA є такі параметри як слабка зв`язність логіки – продюсер не знає про існування інших компонентів, він просто віддає запит далі, що дозволяє розробляти незалежно компоненти системи та легка масштабованість з тієї ж самої причини – коли кожен компонент не залежить від іншого ми можемо просто створити нові екземпляри тих же самих компонентів якщо на якусь частину було б високе навантаження.

1.2.2 Синхронний REST проти EDA

Розглянемо практичний приклад мікросервісної системи купівлі квитків, що складається з чотирьох сервісів: "Квитковий", "Платіжний", "Нотифікаційний" та "Сервіс поїздок" - коли "Платіжний" сервіс успішно

обробляє платіж, він зобов'язаний синхронно повідомити три інших сервіси: оновити статус замовлення (в "Квитковому"), відправити квиток (в "Нотифікаційному") та позначити місце як зайняте (в "Сервісі поїздок"). Це створює низку проблем, головні з яких це порушення ізоляції - "платіжний" сервіс змушений знати API та внутрішню логіку трьох інших сервісів, порушуючи принцип єдиної відповідальності і також крихкість: якщо сервіс "нотифікаційний" недоступний, вся операція купівлі квитка може зазнати невдачі або вимагати складної логіки повторних спроб.

Натомість при EDA підході ми отримуємо таку схему**:** "платіжний" сервіс, замість прямих викликів, генерує єдину подію типу "оплата успішна" та відправляє її до брокера. Три інших сервіси підписані на цю подію і, отримавши її, асинхронно виконують свою частину бізнес-логіки. "платіжний" сервіс нічого не знає про них.

1.2.3 Недоліки та виклики EDA

Попри очевидні переваги, перехід на подієво-орієнтовану архітектуру (EDA) супроводжується суттєвим ускладненням системи. Управління інфраструктурою стає менш прозорим через необхідність контролювати життєвий цикл подій, версіонування їхніх схем та загальну доступність у системі [6].

Проте найбільш критичним викликом є налагодження та моніторинг. На відміну від синхронних систем, де шлях запиту легко простежити через stack trace, в асинхронному розподіленому середовищі діагностика помилок чи втрати подій стає нетривіальним завданням. Фактично, впровадження EDA є архітектурним компромісом: розробники обирають доступність і відмовостійкість ціною простоти обслуговування. Саме тому використання такої архітектури вимагає обов'язкової інтеграції інструментів розподіленого

трасування (наприклад, OpenTelemetry або Jaeger), що дозволяє компенсувати втрату прозорості та забезпечити ефективний аналіз потоків даних

1.3 Оптимізація взаємодії "клієнт-сервер": GraphQL проти REST

1.3.1 Аналіз фундаментальних проблем REST API

Традиційний архітектурний підхід REST (Representational State Transfer), незважаючи на свою універсальності, демонструє неефективність у роботі з мережею через жорстку структуру відповідей сервера. Це породжує проблему надмірної вибірки (over-fetching), коли клієнтський застосунок змушений завантажувати повний об'єкт даних, навіть якщо бізнес-логіка вимагає лише окремих полів, що невиправдано навантажує канал зв'язку.

Водночас виникає і протилежна проблема — недостатня вибірка (under-fetching). Оскільки REST-ресурси є атомарними, для формування складного інтерфейсу часто недостатньо одного звернення до API. Це змушує систему виконувати каскад послідовних запитів до різних ендпоінтів (наприклад, окремо для профілю та пов'язаних сутностей), що створює ефект «водоспаду» та суттєво збільшує загальну затримку відповіді.

1.3.2 GraphQL як рішення

Технологію GraphQL було розроблено компанією Facebook у 2012 році (з відкриттям вихідного коду у 2015-му) як відповідь на архітектурні обмеження REST [7], особливо критичні для мобільних мереж. Ключова відмінність підходу полягає у зміні парадигми взаємодії: замість розподіленої системи ресурсних ендпоінтів, GraphQL забезпечує єдину точку входу для всіх операцій.

Завдяки використанню спеціалізованої мови запитів, клієнтська сторона отримує повний контроль над структурою відповіді, запитуючи лише необхідні атрибути. Такий механізм дозволяє отримувати точні дані за одну ітерацію, що

ефективно нівелює проблеми надмірної та недостатньої вибірки. Стабільність взаємодії гарантується використанням Schema Definition Language (SDL) — механізму строгої типізації, який виступає формальним контрактом між клієнтом і сервером, забезпечуючи валідацію запитів та автоматизацію документування API. Далі у таблиці нижче (табл. 1.1) наведені порівняння характеристик стандартного Rest та GraphQL.

Таблиця 1.1 – Порівняльна характеристика REST та GraphQL

Характеристика	**REST**	**GraphQL**
Архітектура	Архітектурний стиль	Мова запитів та runtime
Ендпоінти	Декілька ендпоінтів (на ресурс)	Зазвичай один ендпоінт
Вибірка даних	Фіксована сервером (призводить до over/under-fetching)	Гнучка, визначається клієнтом
Типізація	Слабка (Weakly typed)	Строга (Strongly typed) (через Схему)
Кешування	Вбудоване на рівні HTTP (GET)	Вимагає кастомної стратегії (зазвичай POST)
Обробка помилок	Використовує HTTP статус-коди	Завжди 200 OK, помилки в тілі errors

1.3.3 "Прокляття" кешування GraphQL

Попри ефективність у гнучкій вибірці даних, архітектура GraphQL створює суттєві виклики в контексті кешування. Архітектура REST ідеально поєднується з вбудованим HTTP-кешуванням. GET /api/users/1 є ідемпотентним запитом, який легко кешується на рівні браузера, CDN або проксі-сервера (Nginx).

GraphQL, у свою чергу, для всіх операцій (включаючи читання) зазвичай використовує POST запити до одного ендпоінту (/graphql).11 POST запити за стандартом HTTP не є кешованими. Це означає, що кожен запит GraphQL, навіть якщо він ідентичний, буде "пробивати" всі рівні кешу і потрапляти до сервера застосунку, створюючи надмірне навантаження на базу даних.

Таким чином, впровадження GraphQL вимагає від команди розробки додаткових зусиль для реалізації кастомних стратегій кешування. Це може включати кешування на рівні клієнта (Apollo Client, Relay) або, що більш важливо для серверної оптимізації, кешування на рівні застосунку з використанням інструментів на кшталт Redis, а також використання патерну DataLoader для пакетної обробки N+1 запитів у межах одного запиту GraphQL.

1.4 Оптимізація CPU-bound операцій: Багатопоточність в Node.js

1.4.1 Проблема блокування Event Loop

Фундаментальна архітектура платформи Node.js, а отже і фреймворку Nest.js, базується на асинхронній однопотоковій моделі обробки запитів, яка керується циклом подій (Event Loop). Такий підхід зарекомендував себе як виключно ефективний для виконання операцій введення-виведення (I/O), таких як взаємодія з мережевими інтерфейсами, запити до баз даних або робота з файловою системою. Завдяки неблокуючій природі I/O, система здатна обробляти тисячі конкурентних з'єднань з мінімальними накладними витратами ресурсів [8].

Однак ця ж архітектурна особливість, яка є ключовою перевагою для I/O-задач, перетворюється на критичний недолік у випадках, коли виникає необхідність виконання операцій, що інтенсивно навантажують центральний процесор (CPU-bound) [9]. До категорії таких ресурсомістких завдань належить широкий спектр обчислювальних операцій: від обробки мультимедійних даних, що включає компресію або зміну роздільної здатності зображень та відеопотоків,

до виконання складних математичних алгоритмів або криптографічного хешування. Окремо варто виділити процеси парсингу та серіалізації великих масивів даних у форматі JSON, які також створюють значне навантаження на процесор.

Проблема полягає в тому, що коли подібна задача потрапляє на виконання в Event Loop, вона фактично монополізує єдиний доступний потік виконання. Зважаючи на однопотокову природу середовища, Node.js втрачає можливість виконувати будь-які інші інструкції до моменту повного завершення поточної важкої операції. У цей проміжок часу сервер стає недоступним для зовнішнього світу: він припиняє приймати нові вхідні з'єднання та призупиняє обробку вже ініційованих запитів. На практиці це призводить до різкої деградації продуктивності системи, що виражається у катастрофічному падінні показника запитів за секунду (RPS) та експоненційному зростанні часу затримки відповіді (latency).

Для вирішення цього класу проблем та нівелювання архітектурних обмежень однопотоковості, в екосистемі Node.js було імплементовано механізм багатопотоковості [10]. Цей модуль, що став стабільною частиною платформи, надає розробникам інструментарій для реалізації справжньої багатопоточності. Він дозволяє створювати додаткові потоки виконання, які здатні виконувати JavaScript-код паралельно з основним потоком, ефективно утилізуючи можливості сучасних багатоядерних процесорів для розподілу навантаження.

Важливо підкреслити відмінність цього підходу від використання модуля cluster, який базується на створенні окремих процесів операційної системи. На відміну від процесів, воркери (worker threads) є значно легковажнішими сутностями з точки зору споживання системних ресурсів [11]. Кожен воркер функціонує в межах власного ізольованого контексту рушія V8 та оперує власним простором пам'яті. Така ізоляція є критично важливою, оскільки вона дозволяє уникнути типових для багатопотокового програмування проблем, пов'язаних із конкурентним доступом до спільних даних (race conditions). Взаємодія та синхронізація між основним потоком застосунку та робочими

потоками реалізується через асинхронний механізм передачі повідомлень, що забезпечує стабільність та передбачуваність роботи системи.

1.4.2 Практична реалізація в Nest.js та проблема DI

Практична інтеграція механізму багатопоточності в екосистему фреймворку Nest.js супроводжується специфічними архітектурними викликами, серед яких ключове місце посідає проблема організації ін'єкції залежностей (Dependency Injection) [12]. Фундаментальна складність полягає в тому, що безпосереднє створення екземпляра воркера через конструктор new Worker() ініціює запуск процесу в абсолютно новому, ізольованому середовищі Node.js. Цей новостворений контекст виконання за своєю природою є «стерильним»: він не володіє інформацією про існуючий IoC-контейнер (Inversion of Control) основного застосунку Nest.js. Як наслідок, всередині такого воркера стає неможливим використання стандартних механізмів ін'єкції для доступу до інших компонентів системи, таких як сервіси для взаємодії з базою даних, модулі конфігурації або зовнішні інтеграції, що суттєво обмежує функціональність відокремленого потоку.

Для ефективного вирішення цієї проблеми необхідно застосувати підхід, що передбачає створення спеціалізованого скрипту-завантажувача, який бере на себе відповідальність за ручну ініціалізацію контексту застосунку всередині воркера. Алгоритм реалізації цієї стратегії розгортається у декілька послідовних етапів, забезпечуючи коректну взаємодію між основним потоком та відокремленим обчислювальним процесом. Процес розпочинається на стороні основного потоку виконання, де, зазвичай у межах сервісу бізнес-логіки (наприклад, AppService), відбувається ініціалізація воркера. На цьому етапі здійснюється передача вхідних даних через параметр workerData, а також налаштування слухачів подій для обробки результатів виконання або потенційних помилок. Далі управління передається безпосередньо скрипту

воркера, який функціонує автономно від стандартної точки входу в застосунок. Ключовою операцією в цьому скрипті є виклик методу NestFactory.createApplicationContext(AppModule). Ця команда виконує завантаження повного графа залежностей та ініціалізацію IoC-контейнера з усіма провайдерами та сервісами, проте, що критично важливо для оптимізації ресурсів, не запускає HTTP-сервер і не відкриває мережеві порти.

Після успішного розгортання контексту застосунку, багатопотік отримує можливість програмно звертатися до будь-якого необхідного сервісу, використовуючи метод app.get(). Отримавши доступ до необхідних залежностей, воркер виконує покладену на нього ресурсомістку логіку, наприклад, складні математичні обчислення (як-от розрахунок послідовності чисел Фібоначчі). Завершальним етапом циклу є передача результатів обробки назад у головний потік, що реалізується через канал міжпотокової комунікації за допомогою методу parentPort.postMessage(), забезпечуючи таким чином повну асинхронність та неблокуючий характер роботи системи.

1.4.3 Диференціація: CPU-bound проти I/O-bound

Фундаментальною передумовою для побудови високопродуктивної системи є чітке розуміння меж застосування доступних інструментів паралелізму. Критично важливо усвідомлювати, що механізм потоків було спроєктовано та імплементовано в середовищі Node.js виключно для оптимізації операцій, обмежених обчислювальною потужністю процесора (CPU-bound). Спроби використання робочих потоків для завдань, пов'язаних із введенням-виведенням (I/O-bound), є архітектурною помилкою, оскільки вони не надають жодних переваг у продуктивності. Платформа Node.js вже містить високоефективну, нативну реалізацію керування асинхронним I/O через Event Loop, яка базується на неблокуючих системних викликах операційної системи. Тому створення додаткових потоків лише для очікування відповіді від мережі чи

дискової підсистеми призводить до невиправданих витрат оперативної пам'яті на утримання контексту потоку без будь-якого прискорення виконання операції.

У інженерній практиці часто виникають хибні уявлення щодо природи навантажень, що призводить до вибору неоптимальних стратегій. Показовим прикладом є задача масової розсилки електронних листів, наприклад, відправка 10 000 повідомлень у циклі. Хоча така операція може здаватися "важкою" через великий обсяг даних, з технічної точки зору вона є класичною I/O-bound задачею, де левову частку часу займає очікування відповіді від SMTP-сервера. Використання потоків у цьому сценарії є недоцільним. Натомість, оптимальним архітектурним рішенням для таких випадків є впровадження надійних черг повідомлень, наприклад, на базі бібліотеки BullMQ. Цей підхід дозволяє делегувати керування мережевими запитами спеціалізованому брокеру, який забезпечує асинхронну обробку та механізми повторних спроб, не блокуючи основний потік виконання застосунку.

Діаметрально протилежним є сценарій обробки медіаконтенту, наприклад, компресія зображення високої роздільної здатності, завантаженого користувачем [13]. Цей процес вимагає інтенсивних математичних операцій над матрицями пікселів, що повністю завантажує ядро процесора. У такій ситуації передача задачі у потоки є безальтернативною, оскільки це дозволяє винести обчислення на окреме фізичне ядро CPU, гарантуючи, що Event Loop залишиться вільним для обробки нових вхідних запитів. Найбільш комплексною є задача, що поєднує обидва типи навантажень, наприклад, пакетна компресія 10 000 фотографій за розкладом. Ефективна реалізація такого сценарію вимагає гібридного підходу: система черг (BullMQ) виступає в ролі оркестратора, що керує потоком завдань і запобігає переповненню пам'яті, тоді як безпосереднє виконання "важкої" обчислювальної частини делегується пулу воркерів.

Тож реалізація такої архітектурної моделі дозволяє не лише розмежувати зони відповідальності між компонентами системи, але й забезпечити гнучке керування ресурсами в умовах змінного навантаження. Застосування черг у даному контексті виконує функцію демпфера, який нівелює різкі сплески

вхідних запитів, перетворюючи хаотичний потік даних у впорядковану послідовність операцій для обробки. Це створює передумови для прогнозованої поведінки системи та дозволяє реалізувати стратегію горизонтального масштабування, де кількість обчислювальних вузлів може динамічно змінюватися залежно від поточної довжини черги та наявних апаратних потужностей. Відсутність жорсткої зв'язності між ініціатором завдання та виконавцем гарантує, що навіть у випадку тимчасової недоступності або перевантаження воркерів дані не будуть втрачені, а системна стабільність залишиться на прийнятному рівні, що є критичним показником для дотримання угод про рівень надання послуг у високонавантажених середовищах.

1.5 Постановка задачі дослідження

Ключовою проблематикою, на вирішення якої спрямоване дане дослідження, є подолання бар'єрів продуктивності та обмеженої масштабованості застосунків в умовах переходу до високих навантажень. Традиційні архітектурні патерни, які покладаються на синхронну модель обробки запитів, монолітну структуру застосунку та однопоточне виконання коду, неминуче стикаються з низкою системних обмежень (bottlenecks). Ці вузькі місця при зростанні кількості користувачів призводять до відчутної деградації якості сервісу та зниження загальної стабільності системи.

Проведений критичний аналіз продуктивності сучасних розподілених систем демонструє, що орієнтація виключно на середні показники затримки (average latency) є хибною практикою, оскільки ці метрики приховують реальний досвід користувачів. Більш репрезентативними є показники "довгого хвоста" розподілу, зокрема 99-й процентиль ($p99$). Аналіз цієї метрики виявляє, що значна частина користувачів — в абсолютних числах це можуть бути тисячі або навіть мільйони запитів — отримує незадовільний досвід взаємодії з системою через рідкісні, але системні затримки. Причиною таких аномалій найчастіше

стають блокування на рівні бази даних, ефекти "холодного старту" мікросервісів або паузи, викликані роботою механізмів збирання сміття (Garbage Collection).

Виходячи з цього аналізу, комплекс проблем, що потребують детального дослідження та вирішення, можна структурувати за чотирма основними векторами. Першим вектором є неефективність взаємодії у моделі "клієнт-сервер" на рівні передачі даних. Використання класичних REST API провокує виникнення проблем надлишкової вибірки (over-fetching), коли клієнт отримує надмірний обсяг даних, та недостатньої вибірки (under-fetching), що змушує виконувати каскадні запити. Це призводить до нераціонального використання пропускної здатності мережі та кумулятивного збільшення затримок.

Другим критичним аспектом є проблема блокування потоку виконання в середовищі Node.js. Будь-яка задача, що вимагає значних обчислювальних ресурсів — чи то криптографічні операції, чи парсинг великих JSON-структур — здатна заблокувати Event Loop, зупиняючи обробку всіх інших вхідних з'єднань на час свого виконання.

Третьою проблемою є ненадійність механізмів фонової обробки. Критичні бізнес-процеси вимагають гарантій виконання, проте прості рішення на кшталт Redis Pub/Sub працюють за принципом "fire-and-forget", що створює ризики втрати даних у випадку збоїв споживача.

Четвертий аспект стосується архітектури розгортання. Поєднання API-сервера та обробників фонових завдань в єдиному процесі створює конкуренцію за системні ресурси та унеможливлює незалежне горизонтальне масштабування компонентів. Таким чином, основною метою даного дослідження є розробка, практична реалізація та верифікація комплексної стратегії оптимізації високонавантаженого проєкту, що дозволить нівелювати описані архітектурні недоліки.

Для реалізації максимальної ефективності розроблюваного рішення необхідно застосувати науковий підхід до визначення технологічного стека. Прийняття рішень щодо вибору архітектури може базуватися на методах аналізу ієрархій та багатокритеріальної оптимізації, що дозволяє формалізувати процес

вибору компонентів системи. Використання такого математичного апарату надає можливість мінімізувати суб’єктивність експертних оцінок та провести кількісне порівняння альтернативних варіантів реалізації за сукупністю зважених критеріїв, таких як швидкодія, надійність, легкість інтеграції та вартість супроводу, забезпечуючи баланс між суперечливими вимогами [14].

Щоб досягти поставленої мети в рамках магістерської роботи необхідно вирішити наступні задачі дослідження:

- Провести аналіз сучасних архітектурних патернів та методів оптимізації продуктивності вебсистем, зосередивши увагу на порівнянні синхронних та асинхронних моделей обробки, а також специфіці роботи Event Loop у середовищі Node.js;
- Обґрунтувати вибір технологічного стека, що забезпечує необхідний рівень продуктивності та масштабованості та розробити гібридну архітектуру, яка поєднує механізми черг повідомлень та виділених робочих потоків;
- Розробити програмну реалізацію серверної частини високонавантаженого застосунку, імплементувавши механізми персистентних черг для CPU-bound операцій та оптимізовану стратегію кешування;
- Спроєктувати та реалізувати підсистему моніторингу на базі WebSockets для забезпечення доставки результатів обробки та візуалізації метрик продуктивності у реальному часі на боці клієнта.;
- Розробити стратегію розгортання та оркестрації, що включає контейнеризацію компонентів системи та налаштування балансувальника навантаження Nginx для забезпечення можливості горизонтального масштабування;
- Провести експериментальне дослідження розробленої системи шляхом навантажувального тестування, порівняти показники ефективності з традиційним синхронним підходом та проаналізувати отримані результати;

Реалізація сформульованих задач відкриває значні перспективи для проєктування нового покоління вебсистем, здатних ефективно функціонувати в умовах експоненційного зростання даних. Запропонована гібридна архітектура є універсальною і може бути імплементована не лише у тестових середовищах, а й адаптована для реальних секторів економіки, де критичними є вимоги до мінімальних затримок та гарантованої обробки транзакцій.

Також потенціал впровадження охоплює такі високотехнологічні галузі, як фінансові технології (FinTech), платформи електронної комерції з високою інтенсивністю трафіку та системи Інтернету речей (IoT), де стабільність потокової обробки даних є ключовим фактором конкурентоспроможності.

Архітектурна гнучкість розробленого рішення забезпечує безшовну інтеграцію з хмарними екосистемами та сучасними інструментами оркестрації контейнерів, такими як Kubernetes. Це створює передумови для реалізації динамічного автомасштабування, що дозволяє автоматично адаптувати обчислювальні потужності під поточні потреби бізнесу, суттєво оптимізуючи операційні витрати.

Крім того, отримані результати формують методологічне підґрунтя для подальших наукових досліджень у напрямку переходу від реактивного реагування на навантаження до проактивного керування ресурсами з використанням предиктивної аналітики.

2 ПРАКТИЧНА РЕАЛІЗАЦІЯ ТА МАСШТАБУВАННЯ МЕТОДІВ ОПТИМІЗАЦІЇ В СЕРЕДОВИЩІ NEST.JS

2.1 Керування асинхронними потоками даних у Nest

2.1.1 Redis Pub/Sub: Патерн для ефемерних сповіщень

Перехід від теоретичних архітектурних парадигм до їх практичної інженерної реалізації вимагає ретельного підбору технологічного стеку та глибокого розуміння нюансів інтеграції інструментів в екосистему фреймворку. У цьому розділі зосереджено увагу на детальному дослідженні методів імплементації асинхронних потоків даних, проєктуванні архітектури розгортання, стратегіях горизонтального масштабування, а також верифікації продуктивності системи в контексті застосунків на базі Nest.js. Дослідження фокусується на синергії таких технологій, як Redis, що виступає високопродуктивним сховищем даних in-memory, BullMQ для надійного керування чергами, Docker для контейнеризації середовища та k6 для проведення навантажувального тестування.

Технологія Redis, завдяки своїм винятковим показникам продуктивності та низькій латентності, часто слугує фундаментом для побудови розподілених систем. Проте в інженерній практиці нерідко виникає концептуальна неоднозначність щодо розмежування двох фундаментально відмінних патернів асинхронної комунікації, які базуються на Redis: механізму публікації/підписки (Pub/Sub) та персистентних черг завдань (Queues). Розуміння архітектурної дихотомії між цими підходами є критичним для побудови надійних систем.

Модель Redis Pub/Sub (Publish/Subscribe) реалізує класичну схему обміну повідомленнями, де відправник даних, визначений як видавець (publisher), не адресує інформацію конкретним отримувачам програмно. Натомість повідомлення публікуються в абстрактні логічні сутності, відомі як «канали» (channels). З іншого боку взаємодії знаходяться підписники (subscribers), які виявляють інтерес до одного або декількох каналів і отримують лише ті дані, що асоційовані з ними. Така архітектура забезпечує високий ступінь ізоляції

компонентів: взаємодія залишається повністю асинхронною та анонімною, тобто видавець оперує без знання про існування чи кількість підписників, і навпаки.

Визначальною та критичною характеристикою патерну Pub/Sub у чистому вигляді є принцип «Fire-and-forget». Дані, що передаються цим каналом, мають ефемерну (ephemeral) природу: Redis не забезпечує механізмів персистентності або довготривалого зберігання цих повідомлень. Це накладає суворі вимоги до синхронізації доступності сервісів: якщо сервіс-підписник не є активним або не має мережевого з'єднання з інстансом Redis у точний момент публікації, повідомлення буде безповоротно втрачено. Система не веде історії подій і не підтримує механізмів повторної доставки (acknowledgment), що фундаментально відрізняє цей підхід від черг повідомлень.

З точки зору практичної імплементації у фреймворку Nest.js патерн Pub/Sub інтегровано як одну зі стандартних стратегій транспортування даних для побудови мікросервісної архітектури [15]. Фреймворк надає високорівневі абстракції, що спрощують налаштування комунікації між сервісами. Використання декларативних декораторів, таких як @MessagePattern (для запитів-відповідей) або @EventPattern (для подієвої логіки), дозволяє розробникам безшовно інтегрувати Redis Pub/Sub у модульну структуру застосунку, інкапсулюючи складність низькорівневого керування з'єднаннями.

2.1.2 BullMQ: Надійні черги для фонових завдань

На відміну від механізму Pub/Sub, який є нативною функцією Redis, BullMQ представляє собою високорівневу бібліотеку для середовища Node.js, що імплементує повноцінну систему черг повідомлень [16]. У цій архітектурі Redis виступає не просто як транспортний канал, а й як надійне сховище (backend) для стану системи. Такий підхід дозволяє реалізувати робастну, персистентну модель обробки даних, яка є критично необхідною для виконання відповідальних бізнес-операцій [17].

Функціонування BullMQ базується на класичному патерні «Продюсер-Споживач» (Producer-Consumer). Процес ініціюється продюсером, який генерує «завдання» (job) — структурований об'єкт даних, що містить всю необхідну інформацію для виконання операції. Це завдання не передається миттєво на виконання, а серіалізується та зберігається у відповідних структурах даних Redis. Воркери (workers), що виступають у ролі споживачів, підтримують постійне з'єднання з Redis, моніторять чергу та вибирають завдання для обробки відповідно до налаштованих пріоритетів та стратегій конкурентності.

Ключовою характеристикою, що фундаментально відрізняє BullMQ від простих механізмів сповіщення, є гарантія надійності та персистентності. Оскільки інформація про завдання фізично зберігається в пам'яті Redis (з можливістю збереження на диск через RDB/AOF), система стає стійкою до збоїв. У випадку аварійної зупинки воркера під час виконання операції або перезавантаження сервера, завдання не втрачається. Механізм черги автоматично детектує тайм-аут або помилку обробки та повертає завдання в чергу для повторної спроби виконання іншим доступним воркером. Це забезпечує дотримання принципу «at-least-once delivery» (гарантована доставка мінімум один раз).

В екосистемі Nest.js інтеграція даної технології реалізується через офіційний пакет @nestjs/bullmq, який забезпечує глибоку сумісність з модульною архітектурою фреймворку. Бібліотека надає розробникам набір високорівневих абстракцій: модулі конфігурації BullModule, декоратори @Processor та @Process для декларативного опису логіки обробки, а також механізми ін'єкції черг через @InjectQueue. Такий підхід дозволяє працювати з чергами, використовуючи звичні патерни Dependency Injection та Inversion of Control.

Спектр застосування BullMQ охоплює будь-які критично важливі асинхронні операції, де неприпустима втрата даних. До типових сценаріїв належать транзакційні розсилки (email-підтвердження, сповіщення), фінансові операції (обробка платежів), генерація звітів у форматі PDF, ресурсомістке

транскодування відеоконтенту, а також будь-які складні I/O або CPU-bound задачі, виконання яких у синхронному режимі призвело б до блокування основного потоку та деградації користувацького досвіду.

Узагальнюючи відмінності між розглянутими підходами, нижче в таблиці (табл.2.1) наведено порівняльний аналіз ключових характеристик Redis Pub/Sub та черг на базі BullMQ.

Таблиця 2.1 – Порівняння Redis Pub/Sub та BullMQ у Nest.js

Характеристика	**Redis Pub/Sub (в Nest.js)**	**BullMQ (в Nest.js)**
Основна сутність	Повідомлення (Message)	Завдання (Job)
Персистентність	Ні	Так
Гарантія доставки	Ні (At-most-once)	Так (At-least-once)
Механізм	Publish/Subscribe	Черга
Додаткові функції	Немає	Пріоритети, повтори, rate-limiting
Основний сценарій	Сповіщення в реальному часі	Надійні фонові задачі

2.1.3 Синергія замість конкуренції

В архітектурі високонавантажених розподілених систем протиставлення механізмів Redis Pub/Sub та бібліотеки BullMQ є методологічно некоректним. Ці технології не конкурують між собою, а формують комплементарну екосистему, де кожен інструмент вирішує специфічний клас задач у межах єдиного бізнес-процесу. Фундаментальна відмінність полягає у вимогах до даних: Pub/Sub забезпечує мінімальну латентність для трансляції ефемерних сигналів, тоді як BullMQ гарантує транзакційну цілісність та персистентність для критичних операцій. Найбільш ефективні архітектурні патерни будуються саме на синергії

цих підходів, де надійне виконання важких обчислень поєднується з миттєвим інформуванням користувача.

Для ілюстрації цього принципу доцільно розглянути комплексний сценарій обробки мультимедійного контенту, наприклад, завантаження та транскодування відеофайлу. Життєвий цикл такого запиту починається на рівні API-сервера, який приймає вхідний потік даних від клієнта. Замість спроби виконати ресурсомістку операцію синхронно, сервер лише зберігає вихідний файл і делегує задачу обробки, створюючи відповідний запит (job) у черзі BullMQ. Цей етап є критично важливим, оскільки запис у чергу гарантує, що задача буде виконана навіть у випадку раптового перезавантаження системи або тимчасової недоступності обробників [18].

На наступному етапі вмикається механізм асинхронної обробки: ізольований воркер, підписаний на відповідну чергу BullMQ, вилучає завдання та розпочинає виконання CPU-bound операцій (наприклад, компресію або зміну форматів). Цей процес відбувається у відокремленому контексті, не впливаючи на продуктивність основного API. Після успішного завершення обчислень виникає необхідність повідомити користувача про результат. Саме на цьому етапі доцільно задіяти механізм Redis Pub/Sub.

Воркер, завершивши роботу із завданням, публікує подію завершення у специфічний канал Pub/Sub (наприклад, user_notifications:<ID>). Ця операція є легковагою і миттєвою. Паралельно з цим, клієнтський застосунок користувача утримує відкрите WebSocket-з'єднання, яке на серверній стороні підписане на цей самий канал. Отримавши сигнал від Pub/Sub, сервер миттєво транслює повідомлення через WebSocket безпосередньо клієнту. Така архітектурна комбінація дозволяє досягти оптимального балансу: BullMQ забезпечує надійність та відмовостійкість фонових обчислень, а Redis Pub/Sub — реактивність інтерфейсу та миттєвий зворотний зв'язок у реальному часі.

2.2 Налаштування BullMQ для високої надійності.

Функціональні можливості бібліотеки BullMQ суттєво виходять за межі примітивної реалізації буфера FIFO (First-In-First-Out), де завдання обробляються суворо в порядку надходження. В архітектурі високонавантажених розподілених систем цей інструмент виступає як повноцінний оркестратор асинхронних потоків, надаючи набір механізмів для керування життєвим циклом завдань, що є критично важливим для забезпечення стабільності сервісу (SLA).

Одним із фундаментальних механізмів керування навантаженням є система пріоритезації черг. BullMQ дозволяє призначати завданням числові вагові коефіцієнти, що гарантує детермінований порядок вибірки: в умовах дефіциту ресурсів система автоматично перерозподіляє обчислювальні потужності на користь бізнес-критичних операцій. На практиці це дозволяє реалізувати сценарії, де транзакційні процеси (наприклад, фіналізація платежів) отримують безумовний пріоритет обробки порівняно з фоновими задачами, чутливість яких до затримок є низькою, такими як маркетингові розсилки або генерація аналітичних звітів.

Поряд із пріоритезацією, важливу роль відіграє темпоральне керування виконанням через механізм відкладених завдань (delayed jobs). Ця функціональність дозволяє планувати виконання операцій на чітко визначений момент у майбутньому або із заданою затримкою, перетворюючи чергу на надійний планувальник подій. Ще більш критичним для стабільності розподіленої системи є вбудований механізм обробки збоїв (retry strategies). Враховуючи, що помилки в мікросервісній архітектурі часто носять тимчасовий характер (мережеві флуктуації, недоступність сторонніх API), BullMQ дозволяє налаштувати автоматичні повторні спроби виконання. При цьому використовується стратегія експоненційної затримки (exponential backoff), яка поступово збільшує інтервал між спробами, запобігаючи перевантаженню зовнішнього сервісу та виникненню ефекту лавиноподібних відмов.

Додатковий рівень захисту забезпечується через контроль конкурентності (concurrency control). Система надає можливості для тонкого налаштування пропускної здатності, дозволяючи жорстко лімітувати кількість завдань, що обробляються одним воркером одночасно. Це запобігає вичерпанню системних ресурсів (CPU, RAM) та блокуванню Event Loop самого воркера. Сукупність описаних механізмів забезпечує реалізацію архітектурного патерну «graceful degradation» (плавної деградації). Замість повної відмови в умовах екстремального навантаження, система зберігає працездатність, адаптивно знижуючи швидкість обробки непріоритетних завдань та забезпечуючи автоматичне відновлення після збоїв без втрати даних.

2.3 Архітектура розгортання та горизонтальне масштабування.

2.3.1 Горизонтальне масштабування - Docker та Nginx

У контексті проєктування відмовостійких систем стратегія горизонтального масштабування розглядається як фундаментальний підхід до підвищення пропускної здатності. На відміну від вертикального масштабування, яке обмежене апаратними можливостями окремого фізичного сервера, горизонтальний підхід передбачає розподіл навантаження шляхом розгортання додаткових обчислювальних вузлів (екземплярів застосунку). Для екосистеми Nest.js це означає перехід від виконання коду в одному процесі до запуску кластера ідентичних серверних застосунків, що працюють паралельно. Стандартом індустрії для реалізації такої архітектури є використання технологій контейнеризації (Docker) у поєднанні з високопродуктивним балансувальником навантаження (Nginx) [19].

Процес побудови масштабованої інфраструктури розпочинається з контейнеризації застосунку Nest.js. Шляхом створення Dockerfile забезпечується пакування коду разом з усіма системними залежностями та середовищем виконання у незмінний артефакт (image). Такий підхід гарантує детермінованість

поведінки системи: застосунок функціонуватиме ідентично незалежно від середовища розгортання. Для оркестрації взаємодії компонентів використовується інструмент Docker Compose, який у файлі конфігурації docker-compose.yml описує топологію системи. Типова конфігурація включає не лише сам API-сервіс, але й супутні компоненти інфраструктури, такі як база даних (PostgreSQL) та система кешування (Redis), об'єднані у спільну віртуальну мережу.

Власне масштабування реалізується через директиву deploy: replicas, яка дозволяє декларативно вказати необхідну кількість екземплярів сервісу. Наприклад, встановлення параметра replicas: 4 ініціює запуск чотирьох ізольованих контейнерів з API Nest.js. Однак наявність кількох запущених екземплярів створює проблему маршрутизації: клієнтський застосунок не повинен знати про внутрішню структуру кластера. Цю задачу вирішує Nginx, який конфігурується як зворотний проксі-сервер (reverse proxy) та єдина точка входу в систему.

Для коректного розподілу трафіку критично важливим є налаштування блоку upstream у конфігурації Nginx. Нижче наведено приклад такої конфігурації, де визначається група серверів для обробки запитів.

Лістинг 2.1 Конфігураційний файл Nginx (nginx.conf):

```
upstream backend_servers {
   server nestjs-app:3000;
}
server {
   listen 80;
   location / {
      proxy_pass http://backend_servers;
      #...
   }
}
```

У наведеній архітектурі Nginx прослуховує стандартний HTTP-порт 80, приймаючи 100% вхідного трафіку. Директива proxy_pass перенаправляє запити до групи backend_servers. Завдяки інтеграції з внутрішнім DNS-сервером Docker, хостнейм nestjs-app автоматично резолвиться у динамічний список IP-адрес усіх запущених реплік контейнера. Це дозволяє Nginx рівномірно розподіляти навантаження між чотирма екземплярами Nest.js, що працюють на внутрішньому порту 3000, забезпечуючи утилізацію ресурсів усіх доступних ядер процесора [20].

Окрему увагу варто приділити технічному нюансу, пов'язаному з механізмом резолвінгу імен (DNS resolution). У застарілих конфігураціях або при некоректному використанні змінних, Nginx може кешувати IP-адресу першої отриманої репліки на етапі запуску. Це призводить до ситуації, коли весь трафік продовжує надсилатися на один контейнер, навіть якщо він вийшов з ладу або були додані нові екземпляри. Використання блоку upstream є архітектурно правильним рішенням, оскільки воно змушує Nginx коректно взаємодіяти з динамічним сервіс-діскавері механізмом Docker, автоматично оновлюючи маршрути при зміні кількості реплік або перезапуску контейнерів.

2.4 Найкращі практики розгортання воркерів

У процесі проєктування розподілених систем, що включають механізми фонової обробки даних, однією з найбільш розповсюджених архітектурних помилок є суміщення контекстів виконання. Йдеться про практику запуску воркерів (зокрема, споживачів черг BullMQ) в єдиному процесі або спільному контейнері з основним API-сервером. Такий підхід, що часто називають «монолітним розгортанням», створює низку критичних вразливостей, які нівелюють переваги асинхронної архітектури.

Першою фундаментальною проблемою такої конфігурації є пряма конкуренція за системні ресурси. У ситуації, коли воркер розпочинає виконання ресурсомісткої задачі (будь то CPU-bound операція, на кшталт обробки зображень, або інтенсивна I/O-операція), він неминуче монополізує доступні ресурси процесора та оперативної пам'яті, фактично «відбираючи» їх у API-сервера, що функціонує в тому ж ізольованому середовищі. Наслідком цього стає непередбачуване збільшення часу відповіді на HTTP-запити та загальна деградація продуктивності користувацького інтерфейсу, що є неприпустимим для високонавантажених систем.

Другим критичним недоліком є неможливість реалізації стратегії незалежного масштабування. Аналіз навантаження реальних систем демонструє, що метрики активності користувачів та обсяги фонових завдань рідко корелюють лінійно. Можливі сценарії, коли система обслуговує 10 000 активних користувачів, які генерують легкі запити, що не потребують фонової обробки. І навпаки, невелика група адміністраторів може ініціювати пакетну обробку мільйона запитів, створюючи пікове навантаження на підсистему черг при мінімальному трафіку на API. В умовах монолітного розгортання інженери змушені масштабувати весь контейнер цілком, що призводить до неефективного використання інфраструктури та невиправданих фінансових витрат.

Виходячи з цього, стандартом індустрії та найкращою практикою є сувора сегрегація компонентів: API-сервер та воркери повинні бути розділені на рівні процесів та контейнерів. Технічна реалізація цього підходу передбачає створення окремих сервісів у конфігурації оркестратора. Нижче наведено приклад конфігураційного файлу docker-compose.yml, що демонструє цей принцип.

Лістинг 2.2 Конфігураційний файл Docker (docker-compose.yml):

```
services:
  api:
    build:.
```

```
  command: node dist/main.js  # Запускає main.ts
  ports:
    - "80:3000"
  deploy:
    replicas: 4
worker:
  build:.
  command: node dist/worker.js
  deploy:
    replicas: 2

redis:
  image: "redis:alpine"
  ...
```

Запропонована архітектура є ключовим елементом оптимізації розгортання, оскільки вона забезпечує можливість гранулярного керування обчислювальними потужностями. Система отримує гнучкість: у випадку зростання вхідного трафіку проводиться масштабування сервісу api, тоді як при накопиченні завдань у черзі збільшується кількість реплік сервісу worker. Такий підхід гарантує максимальну ефективність використання ресурсів та забезпечує високий рівень відмовостійкості, ізолюючи потенційні збої фонових процесів від основного інтерфейсу взаємодії з користувачем.

3 РЕАЛІЗАЦІЯ ТА ЕКСПЕРИМЕНТАЛЬНЕ ДОСЛІДЖЕННЯ

3.1 Вибір інструментальних засобів для реалізації систем

Ефективність, відмовостійкість та можливість масштабування високонавантаженої системи перебувають у прямій залежності від архітектурної обґрунтованості обраного технологічного стека. Для реалізації поставлених у дослідженні завдань було сформовано комплекс інструментальних засобів, який забезпечує баланс між швидкістю розробки, суворістю типізації та продуктивністю виконання коду.

В якості фундаментальної платформи для серверної частини обрано фреймворк Nest. Його архітектура, натхненна Springboot, базується на широкому використанні патернів проєктування, зокрема ін'єкції залежностей (Dependency Injection) та декораторів. Це дозволяє створювати модульні, слабко зв'язані компоненти, що є критично важливим для тестування та подальшої підтримки кодової бази. Використання мови TypeScript забезпечує статичну типізацію на етапі компіляції, що мінімізує кількість помилок під час виконання (runtime errors) та підвищує надійність бізнес-логіки.

Для організації взаємодії між клієнтом та сервером, замість класичного REST, застосовано парадигму GraphQL, реалізовану за допомогою Apollo Server. Цей вибір обумовлений необхідністю вирішення проблем надлишкової вибірки даних (over-fetching) та оптимізації мережевого трафіку. Особливу роль у цій підсистемі відіграє технологія WebSockets, яка забезпечує реактивність системи. Це дозволяє підтримувати постійне двонаправлене з'єднання для миттєвої доставки сповіщень користувачам без необхідності використання ресурсомісткого паттерну long-polling.

Ключовим елементом архітектури для роботи з асинхронними процесами обрано Redis — високопродуктивне сховище даних типу "ключ-значення", що працює в оперативній пам'яті. У розробленій системі Redis виконує подвійну функцію: виступає як шар кешування для зменшення навантаження на основну базу даних та слугує інфраструктурною основою для брокера повідомлень. На

базі Redis імплементовано бібліотеку BullMQ, яка забезпечує надійне керування чергами фонових завдань. Вибір BullMQ продиктований необхідністю гарантованої обробки критичних операцій (таких як транскодування медіа чи фінансові транзакції) з підтримкою механізмів пріоритезації, відкладеного запуску та автоматичних повторних спроб у разі збоїв.

Стратегія розгортання та експлуатації системи базується на принципах контейнеризації з використанням Docker. Ізоляція кожного сервісу в окремому контейнері гарантує ідентичність середовища виконання на етапах розробки, тестування та продакшену, усуваючи проблему "works on my machine". Для оркестрації вхідного трафіку та балансування навантаження між екземплярами контейнерів застосовано вебсервер Nginx, що працює в режимі зворотного проксі (reverse proxy). Він забезпечує єдину точку входу в кластер, термінацію SSL-з'єднань та ефективний розподіл запитів, що є необхідною умовою для горизонтального масштабування системи.

3.2 Опис застосунку

В основі тесту концепції підходу оптимізації роботи високонавантажених проєктів лежить симуляція важкої до виконання логіки. Оскільки в рамках кваліфікаційної роботи реалізація повноцінної продуктової важкої логіки займає багато часу, тому для прикладу будемо використовувати розрахунок якоїсь математичної формули. Для наших цілей ідеально підходить розрахунок фібоначчі: у примітивному виді це рекурсія без ліміту по навантаженню – чим вище цифра яка надходить на функцію, тим складніше буде розрахунок.

3.2.1 Архітектура серверної частини застосунку

Продовжуючи ідею примітивної реалізації архітектури оркестрації високонавантаженого проєкту вся логіка буде реалізована в одному сервісі який

буде дублюватись в докер контейнері в залежності від навантаження. В ідеальному ж середовищі логіка яка виходить за рамки стандартного виконання повинна знаходитись в своєму сервісі і вже цей сервіс повинен або використовувати більше потоків паралельно якщо технологія сервісу підтримує це, або створити більше інстансів цього сервісу для обробки більшої кількості паралельних обчислень.

У якості серверної технології, як раніше і було описано, використовується фреймворк Nest, тому більшість кодової архітектури вже продиктована в рамках цього фреймворку. Стандартна архітектура Nest.js являє собою деревоподібну систему (рис.3.1), де кожна гілка це зв`язок між модулями серверу. Модулі в свою чергу це частина коду яка об'єднана між собою одною тематикою: UserModule, AuthModule, AdminModule, тощо. Така архітектура надає змогу створити ізольовані шматки серверу, за допомогою чого можна контролювати зв`язаність коду між собою і набагато легше модифікувати якісь його частини не турбуючись про те що це може зачепити взагалі іншу логіку.

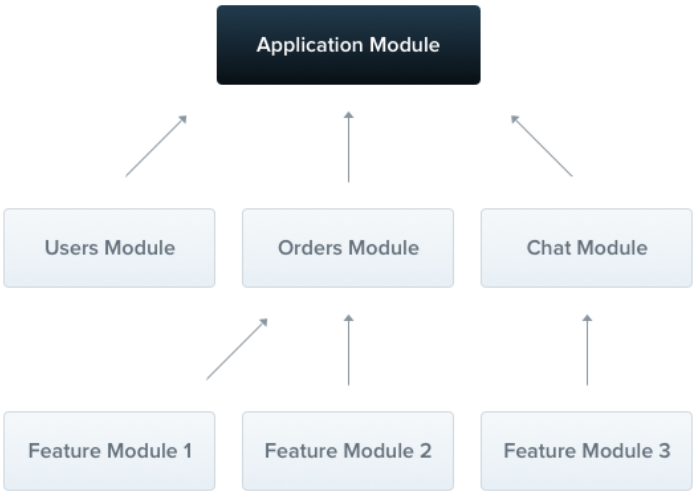

Рисунок 3.1 – UML-діаграма стандартної архітектури Nest.js

Основні компоненти стандартного модуля в Nest.js реалізують CSR (Controller-Service-Repository, рис. 3.2) паттерн і являють собою Controller та Service (UserModule, UserController, UserService). Цей підхід також забезпечує незалежність коду що є основною концепцією фреймворку.

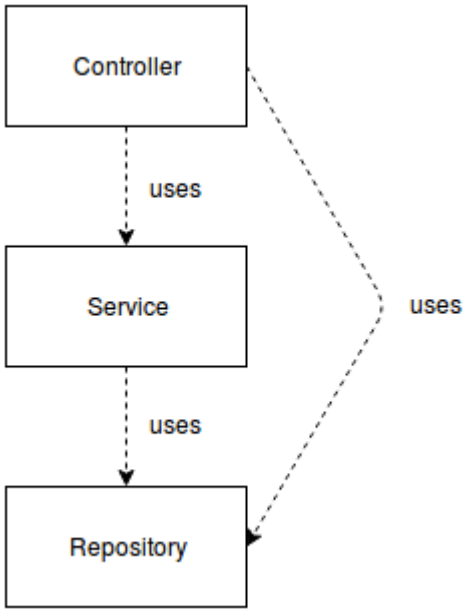

Рисунок 3.2 – UML-діаграма CSR паттерну

В застосунку будуть використовуватись такі модулі(рис 3.3):

- AppModule (головний модуль який імпортує всі інші);
- CacheModule (модуль який реалізує кешування);
- EventModule (модуль івентів);
- QueueModule (модуль черг)
- LoadTestModule (модуль реалізації використання обчислення Фібоначчі);

Про кожний з цих модулів буде розписано більш детальніше в подальших розділах.

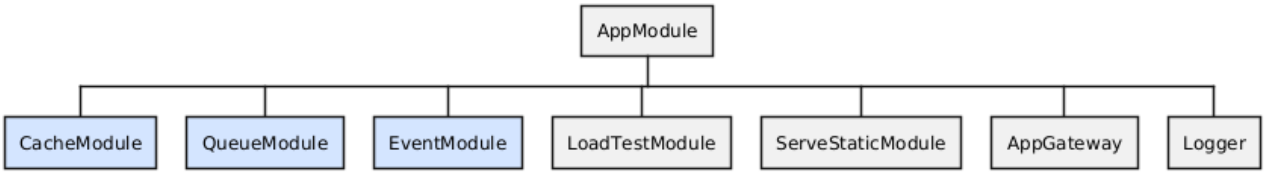

Рисунок 3.3 – UML-діаграма структури проєкту

3.2.2 Архітектура клієнтської частини застосунку

Структура клієнтської частини також буде максимально примітивною. Замість використання вже стандартних фреймворків або бібліотек ми будемо просто віддавати юзеру HTML/CSS/JS based інтерфейс за допомогою технології статичних сторінок. Враховуючи що в поточній схемі тестування навантаження UI не відіграє майже ніякої ролі це ніяк не повпливає на фінальний результат – в данному випадку UI існує лише як інструмент зручної репрезентації та контролювання результатів тестування (рис 3.4).

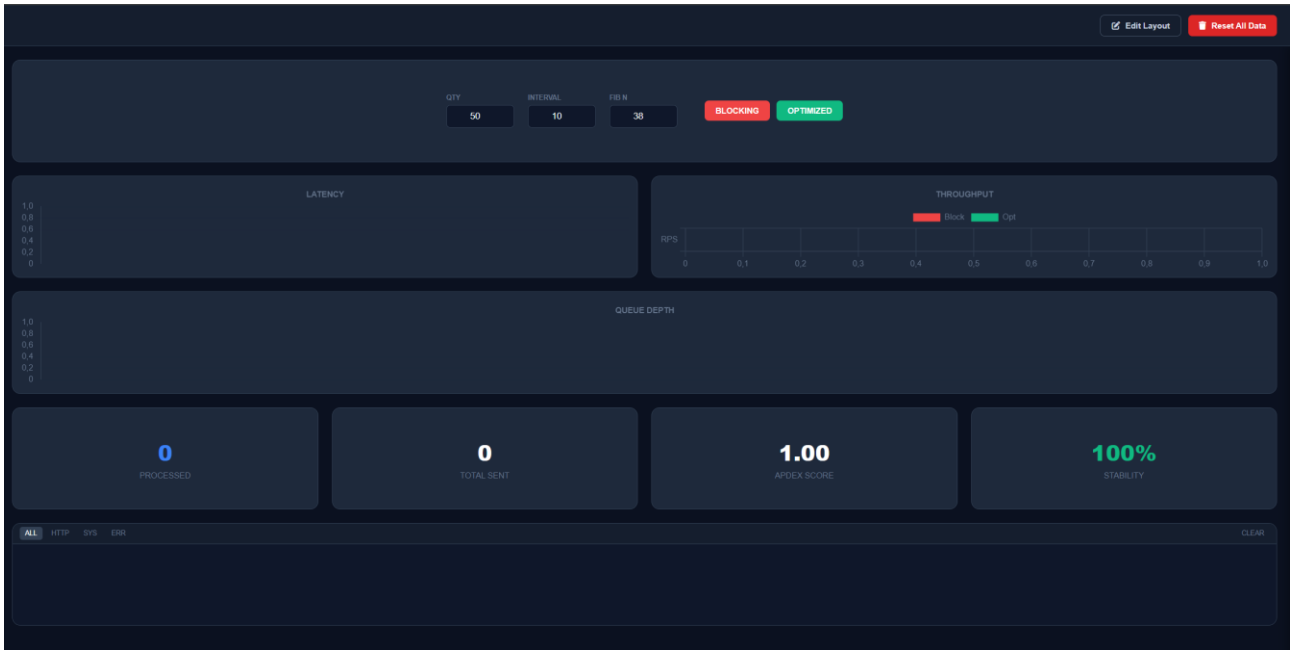

Рисунок 3.4 – Приклад UI застосунку

3.3 Програмна реалізація серверної частини застосунку

3.3.1 Налаштування підключень до серверу

Виходячи з вищеописаної структури почнемо реалізовувати сам застосунок. Для початку розпишемо скрипти старту застосунку який буде відповідати за логіку підняття проекту. У файлі конфігурації package.json визначено набір сценаріїв автоматизації, які регламентують процеси компіляції

коду, форматування, лінтингу та запуску сервера у різних середовищах. Особливу увагу приділено безпеці та гнучкості конфігурації через інтеграцію бібліотеки dotenv. Це дозволяє завантажувати змінні оточення із файлів .env, розділяючи конфігураційні параметри (такі як хости баз даних, порти, секретні ключі) від основного коду. Сценарій запуску в режимі розробки (start:dev) додатково ініціює режим відлагодження та відстеження змін у файловій системі для «гарячого» перезавантаження модуля.

Лістинг 3.1 Скрипти старту застосунку;

```
{ ...
  "scripts": {
    "build": "nest build",
    "format": "prettier --write \"src/**/*.ts\" \"test/**/*.ts\"",
    "start:dev": "pnpm dotenv:load -- nest start --config nest-cli.json --watch --preserveWatchOutput --debug",
    "dotenv:load": "dotenv -e ../../.env -e ../../.env.local -o",
    "lint": "eslint \"{src,apps,libs,test}/**/*.ts\" --fix",
    "test": "jest",
    "test:watch": "jest --watch",
    "test:cov": "jest --coverage",
    "test:debug": "node --inspect-brk -r tsconfig-paths/register -r ts-node/register node_modules/.bin/jest --runInBand",
    "test:e2e": "jest --config ./test/jest-e2e.json",
...
  },
...
}
```

Далі підключимо Redis - шар транспорту даних, який забезпечує взаємодію між клієнтом та сервером у реальному часі. Для цього було розроблено клас

AppGateway, який імплементує основні інтерфейси життєвого циклу WebSocket з’єднання при старті серверу (OnGatewayInit, OnGatewayConnection, OnApplicationShutdown).

Ключовим архітектурним рішенням на цьому етапі є використання Redis не лише як сховища даних, а і як брокера повідомлень для синхронізації WebSocket-серверів. У методі ініціалізації afterInit відбувається підключення двох клієнтів Redis, а саме pubClient для публікації подій та subClient для підписки на них. Це необхідно для коректної роботи адаптера IoAdapter у кластерному середовищі, де користувач може бути підключений до одного екземпляра сервера, а подія про завершення обробки даних може надійти від іншого. Також поверх цього використання методу waitForRedisConnection гарантує, що сервер почне приймати з'єднання лише після повного встановлення зв'язку з інфраструктурою Redis, а реалізація onApplicationShutdown забезпечує коректне закриття з'єднань для запобігання витоку пам'яті при зупинці застосунку.

Лістинг 3.2 Реалізація логіки AppGateway:

```
@WebSocketGateway(gatewayConfig)
export class AppGateway
  implements OnGatewayInit, OnGatewayConnection, OnApplicationShutdown
{
  @WebSocketServer()
  server: Server;

  private pubClient: Redis;
  private subClient: Redis;

  constructor(private eventService: EventService) {}

  async afterInit(server: Server) {
```

```
    // attach server to service
    this.eventService.server = server;

    // setup Redis adapter
    this.pubClient = new Redis({ host: REDIS_HOST, port: REDIS_PORT });
    this.subClient = this.pubClient.duplicate();
    await this.waitForRedisConnection();
    server.adapter(createAdapter(this.pubClient, this.subClient));
  ...
  }

  waitForRedisConnection = async () =>
    Promise.all([
      new Promise((resolve) => this.pubClient.once('ready', resolve)),
      new Promise((resolve) => this.subClient.once('ready', resolve)),
    ]);

  async handleConnection(client: ClientSocket) {
   ...
  }

  async onApplicationShutdown() {
    ...
    await safeQuit(this.pubClient, 'pubClient');
    await safeQuit(this.subClient, 'subClient');
  }
}
```

3.3.2 Налаштування логіки черг

Далі реалізуємо логіку черги. Організація логіки асинхронних фонових черг для обробки ресурсомістких завдань була реалізована на базі архітектурного модуля QueueModule, який використовує бібліотеку BullMQ. Цей інструмент виступає високорівневою абстракцією над структурами даних Redis (Streams та Lists), забезпечуючи надійну оркестрацію черг. Конфігурація модуля виконується декларативно: метод BullModule.forRoot встановлює глобальне з'єднання з Redis-сервером, використовуючи параметри, отримані зі змінних оточення. Важливим аспектом налаштування є використання динамічних префіксів ключів (redisKeyPrefix), що дозволяє ізолювати дані різних середовищ в межах одного фізичного екземпляра Redis, запобігаючи колізіям даних.

Для оптимізації використання оперативної пам'яті сховища Redis було налаштовано політики автоматичного очищення через параметр defaultJobOptions. Опції removeOnComplete: true та removeOnFail: true забезпечують видалення метаданих завдань одразу після їх завершення (незалежно від статусу успіху чи помилки), що є критично важливим для високонавантажених систем, де накопичення історії мільйонів виконаних завдань може призвести до вичерпання пам'яті. Реєстрація конкретної черги для обчислень чисел Фібоначчі відбувається через метод registerQueue, що робить її доступною для ін'єкції в сервіси-продюсери та сервіси-споживачі через механізм Dependency Injection фреймворку Nest.js.

Лістинг 3.3 Реалізація логіки QueueModule:

```
const redisEnv = process.env.REDIS_ENV;

export const redisKeyPrefix = () => `bull:${redisEnv}`;

@Module({
  imports: [
    BullModule.forRoot({
```

```
      connection: redisConnection,
      prefix: redisKeyPrefix(),
      defaultJobOptions: {
        removeOnComplete: true,
        removeOnFail: true,
      },
    }),
    BullModule.registerQueue({ name: QUEUE.Fib }),
  ],
  exports: [BullModule, QueueService, QueueEventsListenerService],
  providers: [QueueService, QueueEventsListenerService],
})
export class QueueModule {}
```

3.3.3 Налаштування логіки кешування

Далі реалізуємо логіку кешування за допомогою Pub/Sub методів з Redis у окремому модулі CacheModule. Це також один з важливих механізмів оптимізації, завдяки якому ми зможемо зберігати отримані обчислення і перевикористовувати їх в подальшому для оптимізації ресурсів. Модуль CacheModule використовує бібліотеку @nestjs/cache-manager разом із драйвером @keyv/redis. Це забезпечує абстракцію над сховищем, дозволяючи при необхідності замінити Redis на Memcached або in-memory кеш без зміни бізнес-логіки. Також у сервісі CacheService було реалізовано патерн "Cache-Aside" (відкладене кешування) через метод getOrSet - цей підхід є оптимальним для систем з високим співвідношенням читання до запису. Алгоритм цього паттерну наступний:

Крок 1. Виконується запит до Redis за ключем key. Якщо дані знайдено, то вони негайно повертаються, уникаючи виконання ресурсомісткої операції.

Крок 2. Якщо дані відсутні виконується передана функція callback.

Крок 3. Результат виконання функції записується в Redis з встановленим TTL (Time To Live).

Крок 4. Блок try-catch гарантує, що у випадку недоступності Redis система не припинить роботу, а перейде у режим прямого виконання.

Лістинг 3.4 Реалізація логіки CacheService

```
@Injectable()
export class CacheService implements OnApplicationShutdown {
 constructor(
  @Inject(CACHE_MANAGER) private cacheManager: cacheManager_1.Cache,
 ) {}

 async getOrSet<T>(
  key: string,
  callback: () => Promise<T>,
  ttlSeconds = 60,
 ): Promise<T> {
  try {
   const cached = await this.cacheManager.get<T>(key);
   if (cached) {
    return cached;
   }

   const fresh = await callback();
   await this.cacheManager.set(key, fresh, ttlSeconds * 1000);
   return fresh;
  } catch (error) {
   console.error('[CacheError]', error);
   return callback();
  }
```

```
  }

  async clear(key: string): Promise<void> {
    await this.cacheManager.del(key);
  }

  async onApplicationShutdown() {
    const stores = this.cacheManager.stores;
    for (const store of stores) {
      if (store && typeof store.disconnect === 'function') {
        await store.disconnect();
      }
    }
  }
}
```

3.3.4 Налаштування точки входу

У фінальному етапі налаштування треба доробити точку входу для того щоб вона коректно працювала і тестово запустити проект (рис.3.5).

Використання NestFactory.create(AppModule) ініціалізує IoC-контейнер. Критично важливим налаштуванням для сучасних вебзастосунків є конфігурація CORS (Cross-Origin Resource Sharing). У реалізованій системі активовано app.enableCors, що дозволяє приймати запити з будь-яких джерел (origin: true) та підтримує передачу облікових даних (credentials: true). Це необхідно для коректної роботи WebSocket-з'єднань та передачі авторизаційних токенів у заголовках запитів, але цей підхід можна використовувати лише в тестових застосунках. У всіх інших випадках треба правильно налаштувати CORS тому що це доволі потужний інструмент захисту.

Для моніторингу вхідного HTTP-трафіку інтегровано middleware morgan у режимі dev. Це дозволяє логувати кожен запит, його статус та час виконання, що є первинним інструментом для виявлення "вузьких місць" (bottlenecks) на рівні мережевої взаємодії ще до етапу профілювання коду.

Лістинг 3.5 Реалізація логіки точки входу:

```
async function bootstrap() {
  const app = await NestFactory.create(AppModule);
  app.enableCors({
    origin: true,
    methods: 'GET,HEAD,PUT,PATCH,POST,DELETE',
    credentials: true,
  });
  app.use(morgan('dev', {
    stream: {
      write: (message) => console.log(`[HTTP] ${message.trim()}`)
    }
  }));
  const port = process.env.PORT ?? 3000;
  await app.listen(port);
  console.log(`Static path: ${ join(process.cwd(), 'public')}`)
  console.log(`App started at: ${port}`)
  console.log(`Warning! Cors disabled, fix before prod`)
  console.log(`API Node started on PID: ${process.pid}`);
}
bootstrap();
```

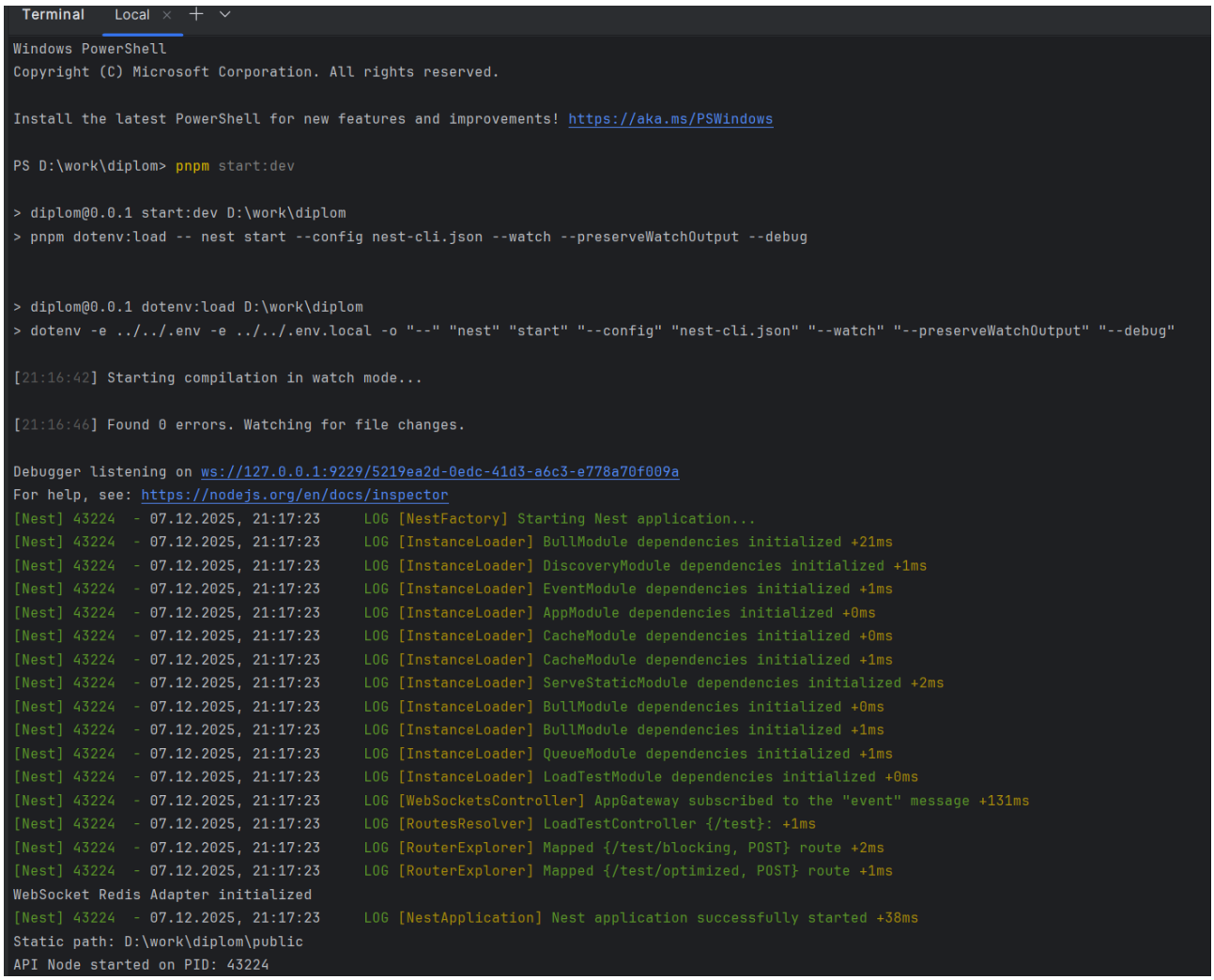

Рисунок 3.5 – Приклад успішного запуску серверу

3.3.5 Порівняльна реалізація бізнес-логіки

Для проведення експерименту в сервісі LoadTestService реалізовано два методи, що демонструють діаметрально протилежні підходи до утилізації ресурсів CPU.

Почнемо з методу синхронне блокувальне виконання - цей метод виконує розрахунок чисел Фібоначчі рекурсивним способом безпосередньо в основному потоці виконання. Алгоритм має експоненційну часову складність O(2^n).

При n=40, кількість операцій перевищує 100 мільйонів. Оскільки Node.js є однопотоковим, на час виконання цієї функції Event Loop блокується. Це означає, що сервер не може приймати нові вхідні з'єднання, обробляти таймери

або виконувати I/O операції. Усі вхідні запити стають у чергу на рівні TCP/IP стеку операційної системи або відхиляються балансувальником навантаження (Nginx) за тайм-аутом.

Далі розберемо метод асинхронного виконання через чергу - метод не виконує обчислень, а лише діє як продюсер. Він формує об'єкт завдання з параметрами { n, clientId, requestTimestamp ...} та додає його до черги fibQueue.

Час виконання цього методу складає мілісекунди - час мережевого запиту до Redis. Event Loop залишається вільним для обробки тисяч інших запитів. Це реалізує патерн "Fire-and-Forget" з точки зору HTTP-запиту, але гарантує виконання завдяки персистентності черги.

Лістинг 3.6 Реалізація логіки тестового сервісу:

```
@Injectable()
export class LoadTestService {
 constructor(@InjectQueue(QUEUE.Fib) private fibQueue: Queue, private eventService: EventService) {}

 async calculateSync(clientId: string, count?: number) {
  const n = count || 40;
  const start = Date.now();

  const result = this.fibonacci(n);

  await this.eventService.emitToUser(SocketIOEvent.Fib, {
   result,
   time: Date.now() - start,
   clientId,
   requestTimestamp: start
  }, clientId);
 }
```

```
  async calculateAsync(clientId: string, count?: number) {
    const num = count || 40;
    return this.fibQueue.add('calculate', { n: num, clientId, requestTimestamp:
Date.now() });
  }
  fibonacci(n: number): number {
    if (n <= 1) return n;
    return this.fibonacci(n - 1) + this.fibonacci(n - 2);
  }
}
```

3.3.6 Реалізація подієво-орієнтованої взаємодії та моніторингу

Оскільки асинхронна модель не повертає результат у HTTP-відповіді, необхідний механізм доставки результату клієнту (Server-Push). Для цього реалізовано підсистему WebSockets.

Клас AppGateway 1 реалізує WebSocket сервер на базі бібліотеки Socket.IO. Ключовою особливістю реалізації є використання Redis Adapter @socket.io/redis-adapter.

У кластерному середовищі, де працює кілька реплік API-сервера, клієнт може мати WebSocket-з'єднання з першим сервером, а Воркер, що обробив завдання, може бути підключений до другого серверу (або взагалі бути окремим мікросервісом).

Redis Adapter вирішує проблему маршрутизації повідомлень:

- Воркер публікує подію в Redis Pub/Sub.
- Redis транслює подію всім підписаним екземплярам Socket.IO серверів.
- Екземпляр, що тримає з'єднання з цільовим клієнтом, отримує повідомлення і відправляє його через WebSocket.

Далі цей механізм використовується в EventModule, де реалізовуються методи, завдяки яким сервер може віддавати якусь інформацію по WebSocket-у.

Лістинг 3.7 Реалізація логіки EventService:

```
@Injectable()
export class EventService implements OnModuleInit {
   public server: Server;
   constructor() {}
   onModuleInit() {
      this.interceptConsole();
   }
   async emit<T extends SocketIOEvent>(
      event: T,
      payload: SocketIOPayload[T],
   ) {
      this.server.emit(event, payload);
   }
   async emitToUser<T extends SocketIOEvent>(
      event: T,
      payload: SocketIOPayload[T],
      clientId: string
   ) {
      const eventPayload: SocketIOEventPayload<T> = {
         payload,
        clientId,
      };
      this.server.to(clientId).emit(event, eventPayload);
   }
   ...
}
```

3.4 Програмна реалізація клієнтської частини застосунку

Як і було зазначено раніше – для забезпечення можливості візуального контролю за ходом експериментів та динамічного керування параметрами навантаження було розроблено клієнтський вебзастосунок. Враховуючи вимоги до мінімізації впливу клієнтської сторони на результати тестування, архітектуру фронтенду побудовано без використання високорівневих JavaScript-фреймворків. Реалізація виконана на базі нативних вебтехнологій HTML5, CSS3 та стандарту ECMAScript 6+ (Vanilla JS), що забезпечує високу швидкодію інтерфейсу та мінімальне споживання ресурсів браузера. Для побудови графіків та організації мережевої взаємодії у реальному часі інтегровано бібліотеки Chart.js та Socket.IO Client відповідно.

Графічний інтерфейс спроєктовано у вигляді модульного дашборду, основу якого складає адаптивна сітка CSS Grid Layout. Це дозволило реалізувати гнучку систему віджетів, які користувач може довільно розміщувати у робочій області. Функціональний набір компонентів включає панель керування параметрами тестування (кількість запитів, інтервал відправки, складність алгоритму Фібоначчі), графічні візуалізатори динаміки затримок (Latency) та пропускної здатності (Throughput), індикатори заповненості черги (Queue Depth), а також блок статистичних метрик, що відображають кількість оброблених запитів, індекс Apdex та стабільність системи. Окремим елементом реалізовано консоль логування, яка підтримує фільтрацію системних повідомлень, помилок та HTTP-логів, що надходять від сервера.

Особливістю програмної реалізації є розробка режиму редагування макета, логіка якого базується на використанні HTML5 Drag and Drop API. У цьому режимі користувач отримує можливість динамічно змінювати розміри віджетів, переміщувати їх між комірками сітки, видаляти з екрана та відновлювати через бічну панель бібліотеки компонентів. Така архітектура дозволяє адаптувати

інтерфейс під конкретні задачі моніторингу, фокусуючись на найбільш релевантних метриках у момент часу (рис. 3.6 – рис. 3.9).

Взаємодія клієнтського застосунку із сервером реалізована через гібридну модель комунікації. Генерація навантаження ініціюється шляхом відправлення асинхронних HTTP POST-запитів на відповідні ендпоінти API (/test/blocking або /test/optimized) з використанням Fetch API. Отримання результатів обчислень та телеметрії відбувається через постійне WebSocket-з'єднання. Клієнт підписується на події job-Fib та server-log, обробка яких відбувається асинхронно: при надходженні нових даних скрипт виконує локальний розрахунок похідних метрик (наприклад, оцінка Apdex Score на основі часу відгуку) та оновлює масиви даних для бібліотеки Chart.js. Такий підхід дозволяє візуалізувати тисячі подій за секунду без блокування основного потоку інтерфейсу, забезпечуючи плавність відображення графіків у реальному часі.

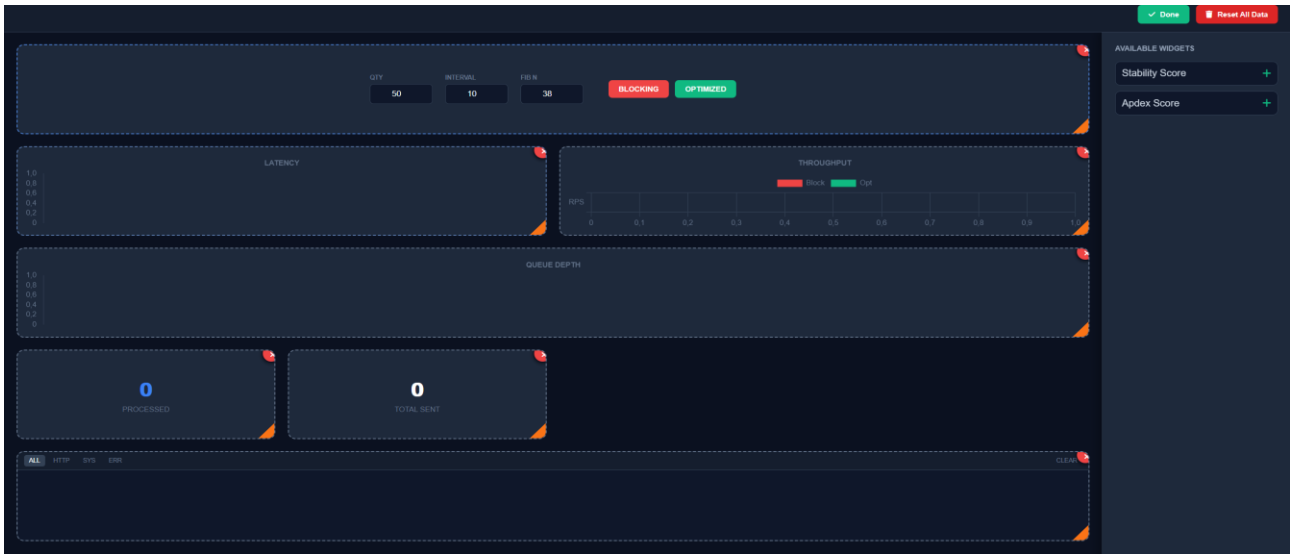

Рисунок 3.6 – Приклад 1 UI застосунку

Рисунок 3.7 – Приклад 2 UI застосунку

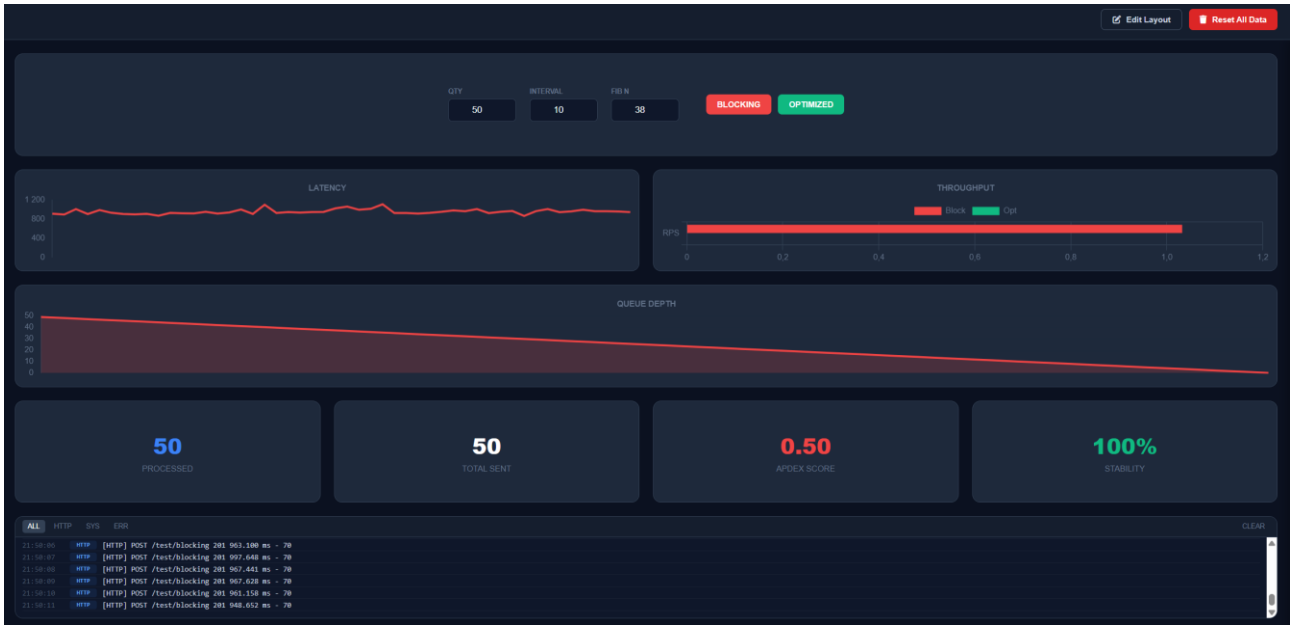

Рисунок 3.8 – Приклад 3 UI застосунку

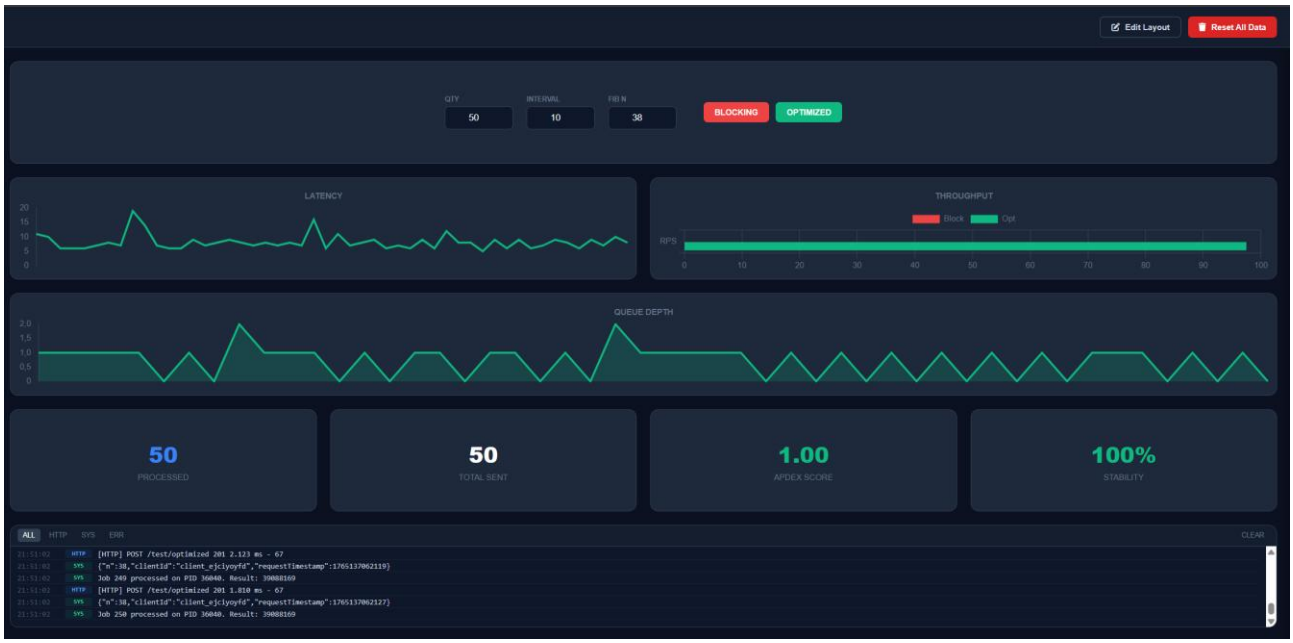

Рисунок 3.9 – Приклад 4 UI застосунку

3.5 Оркестрація та налаштування деплою

Для проведення експериментального дослідження методів оптимізації було розроблено архітектуру розгортання, що базується на принципах

контейнеризації. Використання контейнерів дозволяє забезпечити ізоляцію процесів, керованість ресурсів та легкість масштабування, що є критично важливим для емуляції високонавантаженого середовища.

Лістинг 3.8 Конфігурація оркестрації сервісів:

```
version: '3.8'
services:
  api:
    build:
      context: src
      dockerfile: ./apps/api/Dockerfile
    environment:
      - REDIS_HOST=redis
      - REDIS_PORT=6379
    ports:
      - "3000"
    depends_on:
      - redis
    deploy:
      replicas: 2
      resources:
        limits:
          cpus: '0.5'
          memory: 512M
  redis:
    image: redis:alpine
    command: redis-server --save 60 1 --loglevel warning
    ports:
      - "6379:6379"
    volumes:
```

```
    - redis_data:/data
  nginx:
    image: nginx:latest
    volumes:
      - ./nginx/nginx.conf:/etc/nginx/nginx.conf:ro
    ports:
      - "8080:80"
    depends_on:
      - api
# ...
```

У даній архітектурі реалізовано наступні ключові аспекти деплою - у якості горизонтального масштабування було встановлено 2 репліки для створення двох незалежних екземплярів серверу застосунку. Для тесту цього більш ніж достатньо, але при потребі кількість реплік можна збільшити.

Також було встановлені жорсткі ліміти ресурсів - cpus: '0.5', memory: 512M для того щоб дослідити поведінку системи в умовах обмежених обчислювальних потужностей та перевірити ефективність методів оптимізації пам'яті

Додатково доволі важливою частиною архітектури є зворотний проксі-сервер Nginx, який виконує роль балансувальника навантаження. Конфігурація, представлена в лістингу нижче, забезпечує ефективну маршрутизацію запитів до реплік API [21].

Лістинг 3.9 Конфігурація Nginx:

```
events {
    worker_connections 2048;
}
http {
    upstream api_upstream {
        server api:3000;
```

```
        keepalive 64;
    }
    server {
        listen 80;
        location / {
            proxy_pass http://api_upstream;
            proxy_http_version 1.1;
            proxy_set_header Host $host;
            proxy_set_header Connection "";
        }
        location /socket.io/ {
            proxy_pass http://api_upstream;
            proxy_http_version 1.1;
            proxy_set_header Upgrade $http_upgrade;
            proxy_set_header Connection "upgrade";
            #...
        }
    }
}
```

Структурна схема розгортання та взаємодії компонентів системи зображена на рис. 3.10. Діаграма демонструє маршрутизацію трафіку від клієнта через балансувальник навантаження до кластера застосунків та їх взаємодію з базою даних.

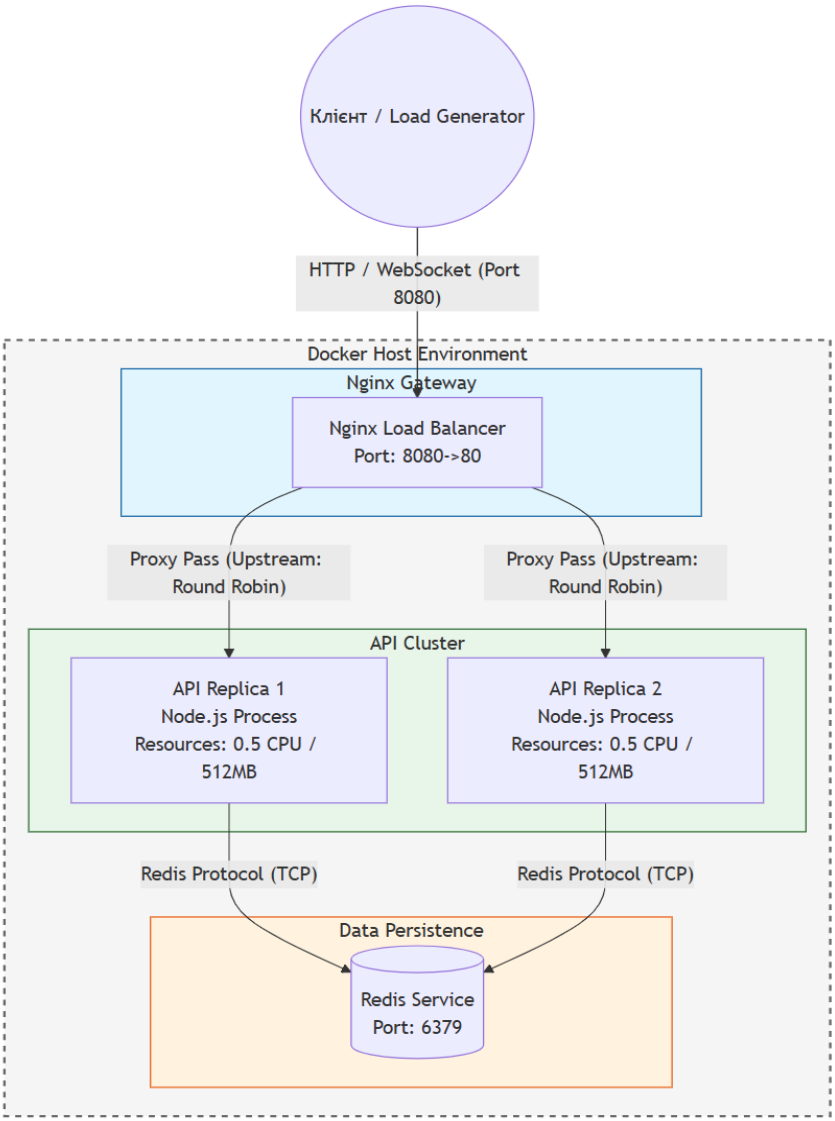

Рисунок 3.10 – UML-діаграма розгортання тестового середовища

3.6 Тестування розробленого застосунку та аналіз результатів

Для проведення тестування передбачено використання наступних контрольних точок та метрик, доступних через розроблені API [22]:

- Latency (час від моменту відправки HTTP-запиту до отримання відповіді);
- Processing Time (реальний час, витрачений CPU на виконання обчислень);
- Total Turnaround Time (Час від ініціалізації запиту до отримання результату через WebSocket);

- Concurrency (здатність системи обробляти кілька запитів одночасно);

Тестування проводиться шляхом подачі запитів на два різні ендпоінти, що реалізують протилежні архітектурні підходи: синхронна обробка (POST /test/blocking) і асинхронна обробка через чергу (POST /test/optimized). Після тестування було зроблено порівняння підходів (табл. 3.1).

Для початку розглянемо перший сценарій з використанням синхронної обробки: при надходженні запиту з n=40 головний потік Node.js переходить у режим інтенсивних обчислень, при чому протягом часу обчислення сервер не відповідає на жодні інші запити тож той же health check запити від балансувальника навантаження завершуються тайм-аутом, що може призвести до помилкового перезапуску контейнера оркестратором.

Якщо приходить 5 одночасних запитів, вони будуть виконуватися суворо послідовно – тож якщо балансувальник не налаштований на створення нових інстансів при перевантаженні основного - час очікування для 5-го клієнта складе суму часів обробки 4-х попередніх запитів. Така поведінка є неприпустимою для високонавантажених систем, оскільки порушує вимоги щодо доступності та чутливості.

При другому сценарії, де використовувалась асинхронна оптимізація та черги, операція додавання завдання в чергу (fibQueue.add) є асинхронною I/O операцією, яка займає лише декілька мілісекунд, завдяки чому HTTP-відповідь із ідентифікатором завдання (jobId) повертається клієнту майже миттєво. Головний потік Event Loop одразу звільняється для обробки наступних мережевих запитів, тому пропускна здатність API (Throughput) обмежується лише швидкістю запису в Redis, а не тривалістю виконання алгоритму Фібоначчі.

Для забезпечення реального паралелізму, на відміну від псевдо-багатопоточності в межах одного процесу Node.js, воркери виносяться в окремі процеси або мікросервіси, що дозволяє горизонтально масштабувати систему шляхом додавання нових обчислювальних вузлів [23].

Ефективність додатково підвищується за рахунок підсистеми кешування (Cache-Aside): при повторних запитах результат не обчислюється заново, а миттєво витягується з Redis, зменшуючи час реакції на 99.9%. Оскільки з'єднання з клієнтом завершується одразу після постановки задачі в чергу, фінальний результат обчислень доставляється асинхронно через WebSocket, який завдяки Redis Pub/Sub адаптеру коректно працює навіть у розподіленому кластерному середовищі.

Таблиця 3.1 – Порівняння синхронного підходу і оптимізованого

Характеристика	**Синхронний підхід**	**Оптимізований підхід**
Архітектурний патерн	Request-Response	Asynchronous Queue-Based
Вплив на Event Loop	Повне блокування на час виконання	Мінімальне (час серіалізації в Redis)
HTTP Time to First Byte	Дорівнює часу виконання	< 50 мс
User Experience (UX)	"Зависання" інтерфейсу на час виконання запиту	Негайний відгук, результат через WebSocket
Стійкість до збоїв	При падінні процесу запит втрачається	Завдання зберігається в Redis, можливий Retry
Ефективність кешування	Відсутня	Висока

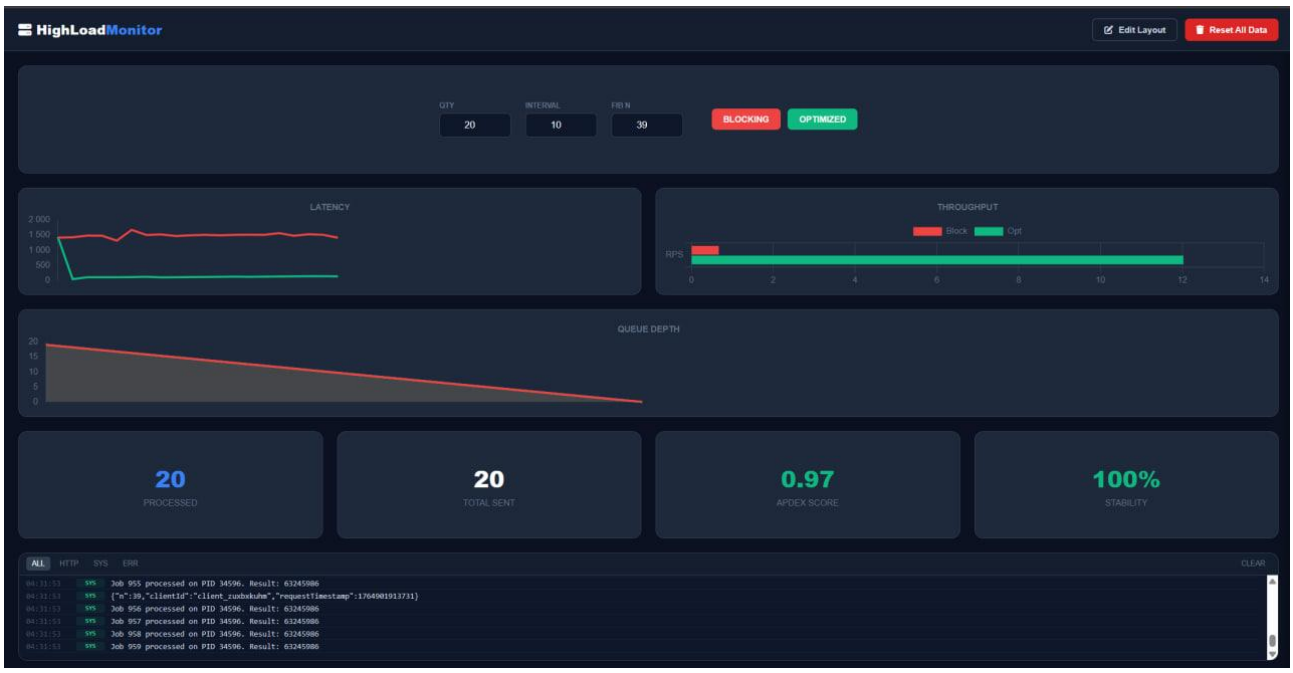

Рисунок 3.11 – Результати тестування

Проведений статистичний аналіз отриманих метрик продуктивності з урахуванням підходів до обробки даних [24] та комплексний порівняльний аналіз архітектурних підходів дозволили отримати емпіричне підтвердження висунутої гіпотези щодо критичної необхідності структурної декомпозиції процесу обробки запитів при проєктуванні високонавантажених систем. Отримані результати свідчать, що перехід від монолітної синхронної моделі до асинхронної архітектури, побудованої на використанні черг повідомлень, є фундаментальною умовою для забезпечення стабільності сервісу. Застосування цього підходу дозволило досягти ефективного логічного та фізичного розмежування підсистеми прийому вхідних HTTP-запитів та підсистеми їх безпосереднього виконання. Така сегрегація відповідальності гарантує збереження високої чутливості API та доступності сервісу для нових користувачів навіть в умовах екстремальних пікових навантажень, повністю усуваючи архітектурний недолік блокування головного потоку подій (Event Loop), який є властивим традиційним синхронним реалізаціям у середовищі Node.js.

Визначальним фактором підвищення загальної продуктивності розробленої системи стала інтеграція розподіленого шару кешування на базі технології Redis. Впровадження стратегії відкладеного кешування (Cache-Aside) забезпечило суттєву мінімізацію затримок відповіді шляхом нівелювання непродуктивних витрат обчислювальних ресурсів процесора на повторну обробку ідентичних завдань. Це дозволило трансформувати складність обробки повторних запитів до константного часу доступу, що є критично важливим для оптимізації використання апаратного забезпечення. Разом із тим, реалізоване архітектурне рішення щодо винесення обчислювальних воркерів у ізольовані процеси операційної системи, у поєднанні з адаптивними механізмами асинхронного зворотного зв'язку через протокол WebSocket, підтвердило високий потенціал запропонованої системи до лінійного горизонтального масштабування. Така архітектура дозволяє проводити гранулярне нарощування потужностей окремих компонентів системи залежно від типу навантаження, що свідчить про повну готовність рішення до розгортання у розподіленому кластерному середовищі під управлінням оркестраторів контейнерів.

Окрему увагу в дослідженні приділено аспектам відмовостійкості та гарантії цілісності даних. Додаткова надійність системи забезпечується впровадженими механізмами автоматичних повторних спроб (retry strategies) на рівні черги завдань, що гарантує стійкість програмного комплексу до тимчасових мережевих збоїв або короткострокової недоступності окремих мікросервісів без втрати даних користувача. Таким чином, розроблений та всебічно протестований програмний стенд демонструє високу ефективність запропонованого комплексного підходу до оптимізації. Спроєктована архітектура, що поєднує асинхронність, паралелізм та інтелектуальне кешування, може бути рекомендована як базовий архітектурний патерн для промислової експлуатації високонавантажених вебзастосунків, які вимагають дотримання суворих угод про рівень обслуговування (SLA).

Отримані в ході виконання кваліфікаційної роботи результати підтвердили ефективність запропонованої гібридної архітектури, що поєднує механізми

персистентних черг та асинхронної обробки для оптимізації високонавантажених систем. Логічним продовженням дослідження є перехід від статичних методів розгортання до впровадження динамічної оркестрації контейнерів, зокрема інтеграція з платформою Kubernetes, що дозволить реалізувати стратегії автоматичного масштабування обчислювальних потужностей відповідно до поточних потреб бізнесу та оптимізувати операційні витрати. Важливим вектором подальшої наукової роботи є зміщення фокусу з реактивного реагування на навантаження до проактивного керування ресурсами, що передбачає використання методів предиктивної аналітики для прогнозування пікових навантажень.

Перспективи дослідження охоплюють адаптацію розробленого архітектурного рішення для вирішення прикладних задач у реальних секторах економіки, таких як фінансові технології (FinTech) та системи Інтернету речей (IoT), де вимоги до стабільності потокової обробки даних є критичними. Ускладнення архітектури через впровадження подієво-орієнтованих патернів (EDA) актуалізує проблему діагностики та моніторингу, тому подальші дослідження доцільно спрямувати на інтеграцію інструментів розподіленого трасування, таких як OpenTelemetry або Jaeger, які дозволяють компенсувати втрату прозорості та забезпечити ефективний аналіз потоків даних у розподіленому середовищі. Також необхідним етапом розвитку системи є посилення контуру інформаційної безпеки, оскільки поточна реалізація містить спрощення на рівні політик CORS, які потребують доопрацювання перед впровадженням у промислову експлуатацію.

ВИСНОВКИ

У ході виконання кваліфікаційної роботи було вирішено актуальну науково-практичну задачу підвищення продуктивності та надійності високонавантажених вебсистем шляхом розробки комплексної стратегії оптимізації та її експериментальної верифікації. Проведений аналіз архітектурних патернів дозволив виявити критичні недоліки традиційних підходів до оцінки ефективності систем, зокрема доведено ненадійність використання середніх значень затримки, які приховують реальний досвід користувачів у моменти пікових навантажень. Натомість обґрунтовано необхідність орієнтації на метрики високих процентилів, що дозволяє виявляти системні аномалії та забезпечувати стабільність сервісу.

В якості технологічного фундаменту дослідження було обрано екосистему Node.js та фреймворк Nest.js завдяки їхній модульній архітектурі, яка сприяє побудові гнучких рішень. Центральним елементом розробленої системи стала технологія Redis, що виконує подвійну функцію високопродуктивного кешу та інфраструктурної основи для брокера повідомлень. Реалізація гібридної архітектури дозволила ефективно поєднати надійність персистентних черг для гарантованої обробки транзакційних задач із високою швидкістю механізмів публікації та підписки для забезпечення реактивності інтерфейсу. Такий підхід спростував тезу про конкуренцію цих технологій та довів ефективність їхньої синергії в межах єдиного бізнес-процесу.

Результати експериментального дослідження на базі розробленого програмного стенда переконливо продемонстрували переваги асинхронної моделі обробки запитів. Порівняльний аналіз показав, що традиційний синхронний підхід при виконанні ресурсомістких операцій призводить до повного блокування циклу подій та експоненційного зростання часу очікування, що робить його непридатним для високонавантажених проєктів. Натомість запропонований асинхронний метод забезпечив миттєву відповідь сервера та звільнення основного потоку для обробки нових з’єднань, делегуючи важкі

обчислення ізольованим процесам. Пропускна здатність такої системи обмежувалася виключно швидкістю мережевої взаємодії, а не складністю алгоритмів.

Вагомим фактором підвищення загальної продуктивності стало впровадження патерну відкладеного кешування, що дозволило мінімізувати використання обчислювальних ресурсів при повторних запитах. Архітектурне рішення щодо розділення серверної частини та фонових обробників у різні контейнери забезпечило можливість незалежного горизонтального масштабування компонентів системи відповідно до поточного навантаження. Застосування балансувальника навантаження гарантувало ефективний розподіл трафіку між репліками застосунку, що підтвердило готовність розробленої архітектури до роботи у кластерному середовищі. Отримані результати свідчать про те, що комплексна стратегія оптимізації забезпечує високу доступність та відмовостійкість системи, тому запропоновані архітектурні рішення можуть бути рекомендовані для впровадження у промислову експлуатацію сучасних вебзастосунків.

Результати роботи апробовано у вигляді 2 тез доповідей під час IX Міжнародної студентської конференції «Актуальні питання та перспективні проведення наукових досліджень» [25], XIII Міжнародної науково-практичної конференції «Innovative directions for improving science, research and practice» [26].

ПЕРЕЛІК ДЖЕРЕЛ ПОСИЛАННЯ

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

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

3. P50 vs P95 vs P99 Latency: What These Percentiles Actually Mean (And How to Use Them). URL: https://oneuptime.com/blog/post/2025-09-15-p50-vs-p95-vs-p99-latency-percentiles/view (дата звернення 10.11.2025).

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

5. Event-Driven Architecture: Pros and Cons. Solace Blog. URL: https://solace.com/blog/event-driven-architecture-pros-and-cons/ (дата звернення 10.11.2025).

6. Особливості event-driven architecture на прикладах з практики. DOU.ua. URL: https://dou.ua/forums/topic/38182/ (дата звернення 10.11.2025).

7. What's the difference between GraphQL and REST? URL: https://aws.amazon.com/compare/the-difference-between-graphql-and-rest/ (дата звернення 10.11.2025).

8. Worker threads. Node.js v20.12.2 Documentation. URL: https://nodejs.org/api/worker_threads.html (дата звернення 15.11.2025).

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

10. Understanding Worker Threads in Nest.js: A Hands-on Guide with Fibonacci Example. Medium. URL: https://medium.com/@Abdelrahman_Rezk/understanding-worker-threads-in-nestjs-a-hands-on-guide-with-fibonacci-example-6f09998e9129 (дата звернення 30.10.2025).

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

12. Nest.js Dependency Injection in Worker Threads. dev.to. URL: https://dev.to/zenstok/nestjs-dependency-injection-in-worker-threads-5deh (дата звернення 10.11.2025).

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

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

15. Redis Nest.js Documentation. URL: https://docs.nestjs.com/microservices/redis (дата звернення 22.10.2025).

16. BullMQ: fast and robust background job processing library for Redi. BullMQ.io. URL: https://bullmq.io/ (дата звернення 10.11.2025).

17. Vechirska A.D., Shyrokorad K.A., Supervisor: Vechirska I.D. (2022) Visual modeling of purchase process management in the electronics online store.71 Діджиталізація науки як виклик сьогодення: матеріали III Міжнародної студентської наукової конференції, м. Львів. ГО «Молодіжна наукова ліга». Вінниця: ГО «Європейська наукова платформа», с.189- 192.

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

19. Dockerizing our application with Docker Compose. wanago.io. URL: https://wanago.io/2023/01/16/api-nestjs-docker-compose/ (дата звернення 27.10.2025).

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

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

22. Schiff, S. D., Piccirilli, J. J., Iser, C. M., & Anderson, K. J. (2006). Load Testing for Assessment and Rating of Highway Bridges. Research Project, (655).

23. Yenugula, M., Kodam, R., & He, D. (2019). Performance and load testing: Tools and challenges. International Journal of Engineering in Computer Science, 1, 57-62.

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

25. Олейник М.В. (2025), Дослідження методів оптимізації високонавантажених проєктів. IX Міжнародної студентської конференції «Актуальні питання та перспективні проведення наукових досліджень». Рівне, Україна, с. 581-582.

26. Олейник М.В. (2025), Дослідження методів оптимізації високонавантажених проєктів. XIII Міжнародної науково-практичної конференції «Innovative directions for improving science, research and practice». Krakow, Poland, с. 57-58.