Міністерство освіти і науки України
Харківський національний університет радіоелектроніки

Факультет комп'ютерної інженерії та управління
(повна назва)

Кафедра електронних обчислювальних машин
(повна назва)

КВАЛІФІКАЦІЙНА РОБОТА
Пояснювальна записка

Рівень вищої освіти перший (бакалаврський)

Засоби паралельних обчислень для прискорення роботи прикладних веб-застосунків

(тема)

Виконав:
здобувач 4 року навчання,
групи КІУКІ-21-6
Данило ДУДНІК
(власне ім'я, прізвище)

Спеціальність
123 «Комп'ютерна інженерія»
(код і повна назва спеціальності)

Тип програми освітньо-професійна
(освітньо-професійна або освітньо-наукова)

Освітня програма
Комп'ютерна інженерія
(повна назва освітньої програми)

Керівник: ас. Олександр МАМЧИЧ
(посада, власне ім'я, прізвище)

Допускається до захисту

Завідувач кафедри ЕОМ _____ Андрій КОВАЛЕНКО
(підпис) (власне ім'я, прізвище)

2025 р.

Харківський національний університет радіоелектроніки

Факультет комп’ютерної інженерії та управління

Кафедра електронних обчислювальних машин

Рівень вищої освіти перший (бакалаврський)

Спеціальність 123 «Комп’ютерна інженерія»
(код і повна назва)

Тип програми освітньо-професійна
(освітньо-професійна або освітньо-наукова)

Освітня програма Комп’ютерна інженерія
(повна назва)

ЗАТВЕРДЖУЮ:
Зав. кафедри _____
(підпис)
“____” _____ 20__ р.

ЗАВДАННЯ
НА КВАЛІФІКАЦІЙНУ РОБОТУ

здобувачеві Дудніку Данилу Олександровичу
(прізвище, ім’я, по батькові)

1. Тема роботи Засоби паралельних обчислень для прискорення роботи прикладних веб-застосунків

затверджена наказом по університету від “ 26 ” травня 2025 р. № 424Ст

2. Термін подання здобувачем роботи до екзаменаційної комісії 17 червня 2025 р.

3. Вхідні дані до роботи мова розробки: JavaScript, середовище розробки VS Code

4. Перелік питань, що потрібно опрацювати у роботі

1) аналіз проблеми та огляд існуючих рішень;

2) розробка застосунку;

3) впровадження технологій та методів паралельного обчислення;

4) тестування продуктивності;

5) висновки.

5. Перелік графічного матеріалу із зазначенням креслеників, схем, плакатів, комп'ютерних ілюстрацій 11 слайдів

6. Консультанти розділів роботи (заповнюється 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

№	Назва етапів роботи	Строк / терміни виконання етапів роботи	Примітка
1	Аналіз проблеми та огляд існуючих рішень	27.05.25 – 30.05.25	
2	Вибір стеку технологій	31.05.25 – 01.06.25	
3	Розробка застосунку	01.06.25 – 06.06.25	
4	Проведення навантажуваного тестування	07.06.25 – 10.06.25	
5	Оформлення текстової частини	11.06.25 – 16.06.25	

Дата видачі завдання " 26 " травня 2025 р.

Здобувач _____
(підпис)

Керівник роботи _____ ас. Олександр МАМЧИЧ
(підпис) (посада, власне ім'я, прізвище)

РЕФЕРАТ

Пояснювальна записка кваліфікаційної роботи: 59 с., 3 табл., 1 дод., 12 джерел.

NODEJS, ПАРАЛЕЛЬНІ ОБЧИСЛЕННЯ, POSTGRESQL, NGINX, DOCKER, BENCHMARK.

Метою кваліфікаційної роботи є дослідження, розробка та експериментальна перевірка ефективності використання засобів паралельного обчислення для підвищення продуктивності та масштабованості прикладних веб-застосунків. Основний акцент зроблено на реалізації балансувальника навантаження на основі Nginx, який дозволяє розподіляти запити користувачів між кількома екземплярами серверної частини, створеної з використанням технологій Node.js та Express.

Робота включає в себе як теоретичний аналіз існуючих рішень у сфері балансування HTTP-трафіку (HAProxy, Traefik, Envoy Proxy), так і практичну частину, в якій реалізовано прототип інфраструктури з можливістю горизонтального масштабування. Застосування Docker дало змогу забезпечити повну ізоляцію компонентів, легкість у розгортанні та автоматизацію процесів тестування.

Результати експериментів показали, що за умови правильного налаштування балансувальника Nginx і оптимального використання паралелізму, можна досягти суттєвого покращення часу відповіді, стабільності під навантаженням та загального ресурсоощадження.

ABSTRACT

Bachelor’s thesis: 59 pages, 3 tables, 1 appendices, 12 sources.

NODEJS, PARALLEL COMPUTING, POSTGRESQL, NGINX, DOCKER, BENCHMARK

The objective of this qualification project is to investigate, develop, and experimentally evaluate the effectiveness of parallel computing techniques in enhancing the performance and scalability of applied web applications. The primary focus is on implementing a load balancer based on Nginx, which distributes incoming user requests across multiple backend instances built using Node.js and Express.

The work includes both a theoretical analysis of existing solutions in the field of HTTP traffic balancing (such as HAProxy, Traefik, and Envoy Proxy) and a practical part in which a scalable infrastructure prototype was developed. The use of Docker enabled complete isolation of system components, simplified deployment, and streamlined the testing process through automation.

Experimental results demonstrated that with proper Nginx configuration and optimal parallelism, it is possible to significantly improve response times, maintain system stability under load, and achieve more efficient use of computing resources.

ЗМІСТ

СКОРОЧЕННЯ ТА УМОВНІ ПОЗНАЧКИ

API – інтерфейс програмування застосунків (англ., Application programming interface)

REST – передача репрезентативного стану (англ., Representational State Transfer)

HTTP – протокол передавання гіпертексту (англ., Hypertext Transfer Protocol)

gRPC – система віддаленого виклику процедур (англ., Google Remote Procedure Call)

TLS – захист на транспортному рівні (англ., Transport Layer Security)

CLI – інтерфейс командного рядка (англ., Command-line interface)

SIMD – одиничний потік команд, множинний потік даних (англ., Single instruction, multiple data)

ACL – список контролю доступу (англ., Access Control List)

NUMA – архітектура з неоднорідним доступом до пам'яті (англ., Non-Uniform Memory Access)

ВСТУП

Сучасна веб-розробка дедалі частіше стикається з викликами масштабованості, швидкодії та стабільної обробки великої кількості одночасних запитів. У час, коли зростають вимоги до продуктивності веб-застосунків, а апаратне забезпечення активно розвивається у напрямку багатоядерних архітектур, постає актуальне питання: як ефективно задіяти потенціал паралельних обчислень для оптимізації роботи серверної інфраструктури? Незважаючи на стрімкий розвиток технологій, багато веб-систем усе ще побудовані за класичною, послідовною моделлю обробки запитів, яка не дозволяє повною мірою використовувати переваги сучасних процесорів. Тому пошук ефективних підходів до розпаралелювання є одним із ключових напрямів сучасного програмування.

У цьому контексті особливої ваги набуває завдання побудови системи, здатної масштабуватись горизонтально без втрати стабільності, що особливо актуально для прикладних веб-застосунків із великою кількістю користувачів. Одним із найважливіших компонентів такої інфраструктури є балансувальник навантаження – програмне забезпечення, що відповідає за розподіл вхідного трафіку між кількома екземплярами серверів. Саме від ефективності балансування залежить не лише швидкість відповіді, але й загальна відмовостійкість системи.

У межах цієї кваліфікаційної роботи розглядається практична реалізація балансувальника навантаження з використанням Nginx, а також побудова повноцінної серверної архітектури на базі Node.js, Express та Docker. Особливу увагу приділено можливостям паралельної обробки запитів, із залученням контейнеризації та розподілу потоків обробки. Робота включає дослідження принципів роботи сучасних інструментів балансування трафіку, таких як HAProxy, Traefik і Envoy Proxy, а також порівняльний аналіз їхніх сильних і слабких сторін.

Важливим аспектом дослідження стало моделювання реального навантаження на систему за допомогою таких інструментів, як hey та wrk, що дозволило на практиці оцінити вплив різних підходів до паралелізації на продуктивність веб-застосунку. Результати експериментів підтвердили доцільність винесення певних елементів обробки запитів у окремі потоки або контейнери, що дозволяє покращити час відповіді та забезпечити більш стабільну роботу сервера за високого навантаження.

Таким чином, дана робота поєднує теоретичне дослідження можливостей паралельних обчислень із практичним проєктуванням і реалізацією масштабованої веб-інфраструктури. Проєкт демонструє не лише технічну можливість реалізації ефективного балансувальника, але й слугує прикладом того, як грамотне застосування сучасних інструментів може суттєво підвищити якість функціонування веб-застосунків у реальних умовах.

1 АНАЛІЗ ПРЕДМЕТНОЇ ОБЛАСТІ

У ході виконання кваліфікаційної роботи слід вирішити наступні задачі:

- провести аналіз існуючих рішень для балансування навантаження у веб-середовищі, зокрема таких інструментів як HAProxy, Traefik, Envoy Proxy;
- дослідити особливості архітектури веб-застосунків та можливості їх масштабування із використанням паралельної обробки;
- розробити інфраструктуру прикладного веб-застосунку з можливістю горизонтального масштабування;
- реалізувати балансувальник навантаження на основі Nginx з підтримкою розподілу запитів між декількома серверними екземплярами;
- налаштувати інструменти для створення штучного навантаження (Apache Benchmark, wrk) для моделювання інтенсивної роботи системи;
- зробити аналіз досягнутих результатів.

1.1 Аналіз існуючих рішень

У процесі створення ефективного рішення для балансування навантаження критично важливо проаналізувати наявні технології, що вже зарекомендували себе у промислових умовах. Нижче розглядаються три найбільш популярні інструменти, які часто використовуються у сучасних розподілених веб-інфраструктурах: HAProxy , Traefik та Envoy Proxy.

Кожне з цих рішень має свої переваги й недоліки, що впливають на вибір у залежності від конкретних вимог до продуктивності, гнучкості конфігурації, сумісності з хмарними середовищами та рівнем підтримки паралелізму.

1.1.1 HAProxy

HAProxy – це, без перебільшення, один з найвідоміших і найстабільніших інструментів для балансування навантаження у світі корпоративних інфраструктур. Історія розвитку цього програмного забезпечення сягає початку 2000-х років, коли питання масштабування інтернет-сервісів стало особливо актуальним. З того часу HAProxy не лише не втратив своєї актуальності, а навпаки – продовжив активно розвиватися, адаптуючись до нових викликів цифрової епохи.

Його ключова риса – надзвичайно висока продуктивність у режимах великого навантаження. Це стало можливим завдяки оптимізації на низькому рівні, а також ефективному використанню системних ресурсів. HAProxy здатен обробляти сотні тисяч одночасних з’єднань при мінімальному споживанні ресурсів, що робить його ідеальним вибором для критично важливих систем.

У контексті паралелізації HAProxy демонструє якісну реалізацію багатопотоковості на рівні CPU. Завдяки підтримці багатоядерної обробки він може масштабуватись горизонтально, розподіляючи навантаження між потоками досить ефективно.

Проте слід зазначити, що це не завжди відбувається автоматично: для досягнення справді високої продуктивності адміністратору необхідно ретельно налаштувати зв’язок між ядрами, сокетами та обчислювальними потоками. HAProxy не використовує GPU або інші апаратні прискорювачі, а отже, потенціал SIMD-інструкцій або паралельних обчислень на графічних процесорах залишається невикористаним. З іншого боку, стабільність і передбачуваність системи іноді важливіші за інноваційність.

Варто звернути увагу і на питання конфігурації. Вона в HAProxy хоч і гнучка, проте досить аскетична та лаконічна, що робить її непривітною для початківців. Опис маршрутів, пріоритетів, ACL-правил, правил stickiness – усе це потребує уважності, а часто й певного досвіду роботи з мережею на

низькому рівні. Тим не менш, саме ця сувора структурованість дає змогу глибоко контролювати кожен аспект маршрутизації та виконувати дуже тонке налаштування поведінки системи.

1.1.2 Traefik

Traefik є яскравим представником нового покоління балансувальників, створених уже в епоху хмарних технологій і DevOps-підходів. На відміну від HAProxy, Traefik не намагається бути «швейцарським ножем» для всіх випадків життя – натомість він орієнтується на простоту, автоматизацію та глибоку інтеграцію з контейнерними системами, зокрема Docker і Kubernetes. Власне, його справжня сила полягає в тому, що він не вимагає ручної конфігурації маршрутів: достатньо запуску сервісу у кластері, і Traefik самостійно виявить його, налаштує TLS, оновить конфігурацію та почне маршрутизувати трафік.

Однак цей підхід має і свої недоліки. Насамперед, йдеться про продуктивність. Оскільки Traefik використовує в основі Go-архітектуру і орієнтований на високу абстракцію, він відчутно поступається у швидкодії порівняно з оптимізованими на рівні C або C++ системами, такими як HAProxy. Зокрема, при великій кількості одночасних з’єднань або нестандартних маршрутах продуктивність може швидко знижуватись. Крім того, система логування та моніторингу в Traefik хоч і присутня, але не завжди достатньо детальна, що створює труднощі при діагностиці складних сценаріїв.

Щодо паралельної обробки запитів, Traefik реалізує асинхронну модель виконання, яка ґрунтується на можливостях мови Go. Це дозволяє до певної міри масштабувати обробку трафіку, але на практиці ефективність такої паралелізації не завжди відповідає очікуванням. Особливо це помітно у випадках, коли необхідно виконувати складну логіку обробки заголовків, TLS-термінацію або авторизацію за допомогою сторонніх API. У таких

ситуаціях стандартна модель асинхронності не дає бажаного виграшу у продуктивності. Тим не менш, завдяки автоматизації конфігурації і високому ступеню сумісності з хмарною екосистемою, Traefik залишається дуже привабливим вибором для невеликих та середніх команд, які прагнуть швидко запустити свій стек без глибокого занурення у налаштування.

1.1.3 Envoy Proxy

Envoy Proxy – один із найсучасніших балансувальників, створений спеціально для потреб сервіс-орієнтованої архітектури та мікросервісів. Його архітектура побудована з урахуванням гнучкості, розширюваності та потреб корпоративного моніторингу. Envoy підтримує величезну кількість сучасних функцій, включаючи HTTP/2, gRPC, TLS-термінацію, стиснення, аутентифікацію, розширену телеметрію, трасування запитів, а також розгалужену систему фільтрів. Усе це робить його незамінним у великих розподілених системах, де велику роль відіграють глибока інспекція трафіку та автоматизоване управління.

Незважаючи на багатство можливостей, робота з Envoy вимагає чимало зусиль. Його конфігурація реалізована через API xDS, що передбачає управління всіма аспектами системи за допомогою зовнішніх сервісів. Це ускладнює розгортання і потребує наявності окремої інфраструктури для управління проксі. Більш того, при першому знайомстві конфігураційна модель здається надмірно складною через значну кількість вкладених об'єктів, необхідність дотримання суворого синтаксису та взаємозалежність між елементами маршрутизації.

Щодо використання паралельних обчислень, Envoy реалізує асинхронну модель виконання з чітким розділенням потоків обробки запитів, логіки маршрутизації та зворотного зв’язку з бекендами. Це дозволяє досить ефективно використовувати багатоядерні системи, хоча при роботі з великим навантаженням у середовищах з NUMA-архітектурою або нестандартною

топологією CPU потрібне ретельне налаштування параметрів. Поки що Envoy не реалізує розширену оптимізацію на рівні SIMD чи використання GPU, хоча його внутрішня архітектура дозволяє теоретично інтегрувати такі механізми.

Завдяки багатофункціональності та гнучкості, Envoy часто стає базовим компонентом у проектах типу Istio або Linkerd, і саме з ним пов’язують майбутнє глибоко інтегрованих мережевих систем нового покоління.

1.2 Аналіз існуючих інструментів для бенчмаркування

1.2.1 wrk

wrk – це один із найпотужніших і водночас мінімалістичних інструментів для високопродуктивного HTTP-бенчмаркування. Його поява стала відповіддю на потребу інженерів у більш гнучкому та масштабованому рішенні, ніж те, що міг запропонувати ApacheBench. Основною метою розробки wrk було забезпечення максимально можливої швидкості тестування при мінімальних накладних витратах.

Ключовою перевагою wrk є підтримка багатопотоковості, що дозволяє ефективно використовувати ресурси сучасних багатоядерних систем. Завдяки цьому інструмент здатен генерувати мільйони запитів за секунду, що робить його незамінним для тестування високонавантажених систем. Висока продуктивність досягається за рахунок написання ядра на мові C, а також використання неблокуючого вводу/виводу на базі epoll/kqueue.

Окремо варто згадати підтримку скриптів на Lua, які дозволяють задавати динамічні або умовні сценарії запитів. Це значно розширює можливості інструменту, хоча й підвищує вхідний бар’єр для початківців. Lua-сценарії дозволяють емулювати різні типи взаємодії клієнтів із сервером, проте їх реалізація вимагає додаткових знань і практики.

Однак, незважаючи на свої сильні сторони, wrk має і суттєві обмеження. Він не підтримує HTTPS «з коробки», що ускладнює тестування захищених API без додаткових налаштувань. Також, через відсутність підтримки повноцінного аналізу відповідей, wrk більше підходить для навантажувального, а не функціонального тестування.

Таким чином, wrk – це високошвидкісний інструмент для тестування систем, де критичним є саме навантаження, а не складність сценаріїв чи детальний аналіз відповідей.

1.2.2 hey

hey – це сучасний аналог ApacheBench, написаний мовою Go і створений для заповнення ніші між простими CLI-інструментами і складними сценарними бенчмаркерами. Автором hey є Jaana Dogan (Go team @ Google), і він орієнтований на розробників, які хочуть швидко та ефективно перевірити відповіді HTTP-сервера.

Одна з головних переваг hey – це багатопоточність. Інструмент автоматично використовує ядра системи для паралельного виконання, що вже дає йому помітну перевагу над ab. Крім того, підтримка HTTPS та виведення статистики по latency дозволяє розробникам робити перші висновки щодо стабільності бекенду без складної інфраструктури.

hey працює як монолітна програма: немає потреби встановлювати залежності чи конфігурувати оточення. Це особливо зручно при використанні у Docker-контейнерах або CI-середовищах.

Слабкою стороною hey є його відносна обмеженість у сценаріях. Він не дозволяє моделювати складні ланцюги взаємодії, не має підтримки потокових запитів чи складних HTTP-головок.

Таким чином, hey – це інструмент, який ідеально підходить для сучасного розробника, що працює з REST API й хоче перевірити продуктивність сервера без зайвих зусиль.

1.2.3 ApacheBench (ab)

ApacheBench – це один із найстаріших і найвідоміших інструментів для бенчмаркінгу HTTP-серверів, який із самого початку був частиною екосистеми Apache HTTP Server. Його простота та універсальність зробили ab класичним вибором для першого наближення до продуктивності веб-сервісів.

Архітектурно ab є однопотоковим, що значно обмежує його масштабованість на сучасних багатоядерних системах. Проте, навіть у такому вигляді він залишається корисним для базових перевірок доступності, відповідності таймінгам та поведінки серверів при середньому навантаженні.

Найсильніший бік ab – це його надзвичайна простота. Інструмент не потребує попередньої конфігурації чи написання скриптів, і достатньо лише однієї команди для запуску тесту. Для багатьох це головна причина використання ab у ролі швидкого smoke-тестера у середовищі розробки або на попередніх етапах CI/CD.

Серед недоліків слід відзначити повну відсутність підтримки сценарного тестування, а також обмежену аналітику: ab не пропонує глибокого аналізу затримок або варіативності відповіді. Більше того, при високих рівнях паралельності результати можуть спотворюватися через те, що інструмент сам є вузьким місцем.

Отже, ApacheBench і сьогодні залишається актуальним для локального профілювання і швидкого тестування REST-ендпоінтів, однак не є інструментом для повноцінного стрес-тесту у продуктивному середовищі.

1.2.4 k6

k6 – це бенчмаркер нового покоління, який поєднує в собі функції тестування навантаження та інтеграційної перевірки поведінки системи. Його головна відмінність – можливість опису сценаріїв у вигляді JavaScript-коду,

що дозволяє емулювати складну взаємодію користувача з додатком. Інструмент створено компанією Grafana Labs, що також гарантує високу інтеграцію з системами моніторингу.

k6 дозволяє не лише генерувати HTTP-запити, але й перевіряти правильність відповідей, логувати поведінку, вимірювати тривалість окремих фаз запиту, і навіть аналізувати помилки на рівні логіки. Це виводить його за межі звичайного бенчмаркінгу в сферу QA і DevOps.

Однією з ключових можливостей k6 є його масштабованість. Він підтримує як локальні тести, так і хмарні (через k6 Cloud), може бути інтегрований у CI/CD пайплайни, а також відправляти метрики до систем збору статистики (Prometheus, InfluxDB).

Недоліками k6 є відносно висока складність при старті: необхідно знати JavaScript та основи тест-дизайну. Також, порівняно з wrk, k6 не є таким легким у плані ресурсів, адже споживає більше пам’яті та CPU при високому навантаженні.

Підсумовуючи, k6 – це чудовий вибір для команд, які хочуть не лише протестувати навантаження, але й перевірити поведінку системи при складних сценаріях реального використання.

1.2.5 autocannon

autocannon – це інструмент, розроблений спеціально для Node.js-середовища, що дозволяє розробникам швидко протестувати продуктивність HTTP-серверів безпосередньо зсередини коду. Написаний повністю на JavaScript, він легко інтегрується з будь-яким проектом, де потрібне швидке профілювання.

Його головна перевага – це зручність. Autocannon має як CLI-інтерфейс, так і Node.js API, що дає можливість запускати бенчмарк прямо зі скрипта або під час юніт-тестів. Він підтримує keep-alive-з'єднання, HTTP pipelining, а також високий рівень паралелізму.

У плані швидкодії autocannon не поступається інструментам типу hey або wrk при локальному тестуванні. Завдяки асинхронній природі Node.js він ефективно масштабується при одночасному використанні багатьох з'єднань.

З іншого боку, як і більшість Node-орієнтованих інструментів, autocannon не має підтримки складних сценаріїв або аналітики на рівні k6. Він також не призначений для тривалих або масштабованих навантажень, де важлива стабільність роботи протягом годин.

Отже, autocannon – це ідеальний lightweight-інструмент для тестування REST API під час розробки в екосистемі JavaScript, але не замінить повноцінні засоби для сценарного чи хмарного навантаження.

1.3 Висновки з аналізу вже існуючих рішень

Отже, узагальнюючи отримані результати, можна стверджувати, що кожне з проаналізованих рішень орієнтоване на різні типи навантажень і має власні сценарії ефективного застосування.

HAProxy демонструє максимальну продуктивність і стабільність у традиційних інфраструктурах з чіткими правилами маршрутизації, але вимагає ручної конфігурації та добре розуміється на низькорівневих механізмах.

Traefik натомість надає вищий рівень автоматизації та інтеграції у хмарне середовище, однак жертвує продуктивністю та гнучкістю. Envoy є найпотужнішим з точки зору функціональності та можливостей моніторингу, але вимагає значних ресурсів і досвіду при налаштуванні.

Для задач, пов'язаних із дослідженням можливостей паралелізації обробки, HAProxy слугує прикладом максимальної ефективності при використанні CPU, тоді як Traefik і Envoy демонструють потенціал для подальшого розвитку у напрямку глибокої оптимізації паралельних обчислень.

1.4 Висновки з аналізу інструментів для бенчмаркування

Інструменти на кшталт k6 та autocannon ідеально підходять для ситуацій, де необхідно емулювати складні користувацькі сценарії або глибоко інтегрувати тестування у DevOps-процеси. Вони забезпечують гнучкість, проте потребують більше часу на налаштування, а також споживають більше ресурсів, особливо при тривалому навантаженні. ApacheBench, попри свою простоту та популярність, суттєво поступається конкурентам у продуктивності через обмеження однопотоковості та відсутність сучасних механізмів паралелізації.

У контексті задач, пов’язаних із аналізом ефективності серверної архітектури та дослідженням поведінки системи під високим навантаженням, найбільш доцільним вибором є інструменти, що поєднують максимальну продуктивність із простотою використання. Саме тому було обрано wrk та hey.

wrk є еталоном високошвидкісного навантажувального тестування. Його здатність генерувати мільйони запитів за секунду, підтримка багатопотоковості та гнучкість сценаріїв на Lua робить його незамінним для перевірки системи у пікових режимах. Він дозволяє глибоко дослідити вплив рівня паралелізму на продуктивність серверу та ефективно використовує ресурси CPU. Це робить його надзвичайно корисним для технічного аналізу та оптимізації низькорівневих аспектів обробки запитів.

З іншого боку, hey забезпечує зручність, швидкість старту і простоту у використанні. Завдяки підтримці багатоядерної обробки та можливості легко інтегруватися у CI/CD, він стає чудовим інструментом для повсякденного використання розробниками. Крім того, його реалізація на Go гарантує стабільність і високу швидкодію навіть без глибокого налаштування. hey дає змогу швидко отримати базове уявлення про навантажувальні характеристики системи, що особливо актуально на етапах розробки та первинного профілювання.

Таким чином, поєднання wrk і hey дозволяє охопити як глибоке технічне дослідження системи на межі її можливостей (wrk), так і швидке прикладне тестування у розробницькому процесі (hey). Такий підхід забезпечує баланс між точністю, продуктивністю і зручністю, що є оптимальним для цілей цього дослідження.

1.5 Аналіз підходів до паралелізації

У сучасних системах балансування навантаження паралелізація є ключовим інструментом для забезпечення високої пропускної здатності, низьких затримок і масштабованості. Наш проект спрямований на розробку ефективної системи, яка здатна розподіляти запити між серверами з урахуванням різних стратегій та динамічних умов мережі. Паралелізація в такому контексті охоплює різні рівні обробки даних, розподілу завдань і керування станом.

1.5.1 Паралелізація на рівні обробки вхідних запитів

Одним із найприродніших підходів є паралельна обробка вхідних мережевих запитів. У цьому випадку кожен запит, що надходить на балансувальник, обробляється окремим потоком або таском. Це дозволяє значно збільшити кількість одночасних підключень, які система може обслуговувати без затримок. Важливою перевагою цього підходу є масштабованість: система легко розширюється з ростом кількості ядер процесора, дозволяючи розподіляти навантаження між потоками.

Водночас, така модель паралелізації має свої виклики. Зокрема, для коректного розподілу запитів необхідно підтримувати глобальний стан системи – інформацію про доступні сервери, їхню поточну завантаженість, сесії користувачів тощо. Зберігання та оновлення цього стану у багатопоточному середовищі вимагає синхронізації, що може призводити до

затримок і вузьких місць, особливо при високому рівні паралелізму.

Переваги цього підходу:

- добре масштабування при збільшенні кількості ядер процесора;
- зменшення часу відповіді за рахунок одночасної обробки запитів;
- простота розподілу роботи між потоками, інтуїтивна модель.

Недоліки цього підходу:

- виникає необхідність у синхронізації доступу до спільного стану (інформація про сервери, сесії), що може створювати вузькі місця;
- високий рівень паралелізму може спричиняти проблеми з блокуваннями і станами гонки;
- потреба ретельного управління потоками, щоб уникнути витрат на контекстні переключення.

1.5.2 Паралелізація на рівні управління станом і логікою балансування

Крім обробки запитів, паралельно можуть виконуватись завдання з управління логікою балансування: моніторинг стану серверів, прийняття рішень про переадресацію трафіку, оновлення метрик. Відокремлення цих допоміжних процесів у паралельні потоки або окремі компоненти допомагає знизити затримки основної обробки запитів і підвищити відмовостійкість системи.

Однак такий поділ вимагає добре спроектованої системи взаємодії між потоками, наприклад, через черги повідомлень або подій. Невдало реалізована синхронізація може викликати несвоєчасне оновлення стану або втрату актуальності інформації, що негативно впливає на якість балансування і може призвести до нерівномірного розподілу навантаження.

Основні виклики цього підходу:

- зменшує навантаження на основний потік обробки запитів;
- підвищує відмовостійкість системи завдяки розділенню функціональності;

- дозволяє оновлювати стан системи більш оперативно, що покращує якість балансування.

Основні недоліки цього підходу:

- складність у реалізації ефективної комунікації між потоками (черги, події);
- можлива несвоєчасність оновлення стану через затримки у синхронізації;
- підвищена складність тестування і відлагодження системи.

1.5.3 Паралелізація на рівні мережевого вводу/виводу

Окремою важливою складовою є паралельна обробка мережевих операцій, що включає прийом, розбір і надсилання пакетів. Використання неблокуючих I/O, асинхронних операцій і пулів потоків дозволяє ефективно розвантажувати CPU та зменшувати час відповіді.

У нашому проекті акцент зроблено на застосуванні сучасних бібліотек з підтримкою асинхронного вводу/виводу, що дозволяють масштабувати обробку мережевого трафіку без значного росту накладних витрат на контекстні переключення потоків.

Цей підхід дає можливість одночасно обробляти тисячі підключень з мінімальним впливом на продуктивність.

Проте, асинхронний підхід вимагає ретельного проєктування, аби уникнути складних ситуацій з блокуваннями та забезпечити коректне завершення операцій, особливо в умовах високої конкуренції за ресурси.

Перевагами можуть бути:

- ефективне використання ресурсів процесора;
- можливість обробляти тисячі одночасних підключень з мінімальними затримками;
- зниження накладних витрат на контекстні переключення у порівнянні з традиційними блокуючими операціями.

Недоліками можуть бути:

- складність реалізації і відлагодження асинхронного коду;
- високі вимоги до проєктування, щоб уникнути блокувань і помилок;
- можливі труднощі з коректним завершенням операцій у разі аварій.

1.5.4 Паралелізація алгоритмів балансування навантаження

Балансувальники реалізують різні алгоритми розподілу трафіку – від простих Round Robin і Least Connections до більш складних із урахуванням метрик продуктивності, затримок і доступності. Паралельне виконання обчислювальної логіки для цих алгоритмів може суттєво прискорити прийняття рішень, особливо у великих системах із сотнями або тисячами бекенд-серверів.

У нашому проекті передбачено можливість динамічного вибору і комбінації алгоритмів, що вимагає розподілу обчислень між кількома потоками або навіть розподіленими вузлами.

Проте, паралелізація алгоритмів балансування пов’язана з необхідністю узгодження даних між потоками, що вимагає зваженого підходу до синхронізації і контролю цілісності стану.

Основні переваги даного підходу:

- прискорення прийняття рішень у великих системах з великою кількістю серверів;
- можливість використання складних алгоритмів, які вимагають інтенсивних обчислень;
- гнучкість у застосуванні різних стратегій балансування одночасно.

Недоліки цього підходу:

- потреба у складних механізмах синхронізації для узгодження стану між потоками;
- ризик появи непослідовності даних або затримок у оновленні інформації;

- підвищена складність реалізації і тестування.

1.6 Аналіз архітектури проекту

На перший погляд, задача балансування навантаження виглядає просто: клієнт надсилає запит – балансувальник вибирає сервер та передає запит далі.

Проте, при реалізації високопродуктивної системи, здатної обслуговувати тисячі запитів щосекунди з мінімальною затримкою, виникають суттєві архітектурні виклики, особливо у контексті паралелізації.

Ключова складність полягає у необхідності поєднати реактивність обробки запитів із синхронним станом системи. Кожен запит повинен швидко отримати рішення про те, на який бекенд-сервер його перенаправити. Водночас це рішення має ґрунтуватися на актуальному стані системи: завантаженість серверів, доступність, відповіді на останні health-check перевірки, політика маршрутизації тощо. Паралельна обробка запитів породжує проблему: як забезпечити, щоб усі потоки працювали з актуальним, консистентним станом, але без надлишкової синхронізації, яка могла б знівелювати виграш у продуктивності?

Ще одним викликом є паралельна обробка мережевого вводу/виводу, яка часто створює вузьке місце через обмеження системних ресурсів – сокети, дескриптори, обробка TCP-з’єднань. Ефективна обробка вимагає використання неблокуючих I/O та пулів потоків, що само по собі додає складності при координації між компонентами системи.

Також існує проблема централізації логіки балансування. Якщо рішення про вибір сервера приймається у спільному модулі, який одночасно викликається багатьма потоками, це призводить до необхідності блокування, що знижує масштабованість, але дублювання логіки у кожному потоці або кешування рішень може призвести до розсинхронізації та неефективного розподілу навантаження.

Класичні підходи до паралелізації з використанням блокуючих структур або черг повідомлень можуть дати виграш лише на малому навантаженні.

Таким чином, основним архітектурним викликом стало досягнення ефективної паралелізації із гарантованим узгодженням стану системи у режимі реального часу. Система повинна бути здатною до масштабування, реактивної обробки тисяч запитів одночасно і при цьому зберігати узгоджений, актуальний стан – без перевантаження каналів синхронізації або створення «ботлнеків».

1.7 Обраний підхід для виконання проекту

Для досягнення високої продуктивності при збереженні точності маршрутизації, у розробленій системі балансування навантаження було обрано гібридний підхід до паралелізації.

Ядро системи – було реалізовано за принципом асинхронної обробки запитів у пулі потоків, де кожен вхідний запит миттєво передається на обробку незалежним таском. Це дозволило повністю задіяти всі доступні ядра процесора, забезпечити масштабування та мінімізувати час простою між запитами.

Щоб уникнути «болтнеків» у доступі до спільного стану системи, було впроваджено lock-free snapshot-кеш, який періодично оновлюється окремим потоком. Таким чином, усі робочі потоки читають з консистентного стану без блокувань, а лише один контрольний потік займається оновленням цієї інформації. [10] Це дозволило уникнути проблеми надмірної синхронізації та блокувань під час високого навантаження.

Завдяки цьому поєднанню – асинхронна обробка, відокремлення стану, спеціалізовані потоки – було досягнуто значного підвищення продуктивності системи при повному збереженні точності логіки маршрутизації.

2 АНАЛІЗ ВИКОРИСТАНИХ ТЕХНОЛОГІЙ

У рамках дипломного проекту було реалізовано програмне забезпечення, яке виконує функції балансувальника навантаження для веб-застосунків на основі Nginx з елементами власної конфігурації, моніторингу та інтеграції з бекенд-сервісами. Основна мета – оптимізувати розподіл запитів між кількома екземплярами застосунку, забезпечуючи рівномірне навантаження, підвищену відмовостійкість та можливість масштабування.

2.1 JavaScript

JavaScript – це високорівнева мова програмування, яка спочатку створювалася для реалізації інтерактивної поведінки на вебсторінках. Вона є однією з ключових технологій сучасної веб-розробки разом з HTML та CSS. Сьогодні JavaScript підтримується усіма сучасними браузерами та використовується як на клієнтській, так і на серверній стороні [1].

Завдяки своїй універсальності, мова стала основою для розробки повноцінних веб-застосунків, і навіть десктопних або мобільних програм. JavaScript виконується в середовищі браузера, де він може взаємодіяти з DOM, обробляти події користувача, а також виконувати запити до серверів.

Середовище виконання також може бути серверним, наприклад, у випадку з Node.js, де JavaScript використовується для створення бекенду.

Синтаксис JavaScript гнучкий і вміщує як процедурні конструкції, так і підтримку об’єктно-орієнтованої моделі. Завдяки введенню стандарту ECMAScript 6 (ES6) та подальшим оновленням, мова стала ще більш потужною, отримавши такі можливості, як класи, модулі, стрілкові функції та синтаксис async/await. JavaScript динамічно типізований, що робить його легким для швидкого написання, проте це також може призводити до складних помилок, які важко виявити без додаткових інструментів.

JavaScript має подієво-орієнтовану модель програмування, що ідеально підходить для роботи з веб-інтерфейсами та реакцією на дії користувача. Важливою особливістю мови є її асинхронність, яка реалізується через колбеки, проміси та ключові слова async/await.

Це дозволяє ефективно працювати з мережею або базами даних без блокування виконання коду. Завдяки JavaScript було створено багато популярних фреймворків і бібліотек, таких як React, Angular та Vue, які суттєво пришвидшують розробку.

Більшість сучасних вебсайтів неможливо уявити без JavaScript, адже саме він відповідає за логіку взаємодії та динаміку інтерфейсу.

2.2 Node.js

Node.js – це середовище виконання JavaScript на стороні сервера, яке базується на високопродуктивному рушії V8 від Google. На відміну від традиційних серверних платформ, які використовують модель з багатьма потоками, Node.js працює за принципом однопотокової, неблокуючої обробки подій. [2] Основу його архітектури становить подієвий цикл (event loop), який забезпечує асинхронну взаємодію з операційною системою без блокування потоку виконання.

Ця модель дозволяє обробляти велику кількість одночасних запитів, не створюючи окремий потік або процес на кожен запит. Всі I/O-операції, включно з читанням файлів, роботою з базами даних, мережею та файловою системою, виконуються асинхронно з використанням колбеків, промісів або async/await.

Node.js підтримує дві основні модульні системи: CommonJS (традиційна для Node.js) та ECMAScript Modules (ESM). Це забезпечує сумісність з широким спектром бібліотек і фреймворків. Також він підтримує TypeScript – надмножину JavaScript з статичною типізацією, що покращує масштабованість і підтримуваність великих проєктів.

Щодо паралелізму, незважаючи на однопоточну модель виконання, Node.js підтримує створення окремих процесів (child_process) та воркерів (worker_threads). Перший підхід зручний для повністю незалежних задач, другий – для обчислювально затратних операцій, які можуть виконуватись у фонових потоках без блокування основного потоку. [5] Обидва механізми дозволяють реалізувати ефективну багатопоточність, при цьому зберігаючи простоту основної архітектури.

Node.js ідеально підходить для побудови високонавантажених мережевих сервісів. Завдяки вбудованому модулю http, або через фреймворки, такі як Express.js, можна швидко створювати сервери, які обробляють тисячі запитів одночасно.

У таких сценаріях Node.js дозволяє реалізувати високоефективне балансування навантаження не лише на рівні інфраструктури (наприклад, через Nginx або HAProxy), але й безпосередньо в коді додатку, наприклад через розподіл запитів між підпроцесами чи кластеризацію (cluster API).

2.3 Express.js

Express – це мінімалістичний веб-фреймворк для Node.js, який забезпечує базову інфраструктуру для створення серверів і API. Він дозволяє легко обробляти HTTP-запити та маршрутизацію шляхів завдяки вбудованій системі роутінгу. Express підтримує методи GET, POST, PUT, DELETE та інші HTTP-методи, що дозволяє створювати RESTful-сервіси. Middleware-функції в Express дозволяють виконувати обробку запитів на різних етапах: перевірку авторизації, логування, обробку помилок тощо.

Фреймворк підтримує шаблонізатори, такі як EJS або Pug, для генерації HTML на стороні сервера. Обробка тіла запиту реалізується через бібліотеки типу body-parser або вбудовану підтримку JSON і URL-даних. Express сумісний із CORS, cookie, сесіями та іншими веб-механізмами.

Структура проекту залишається повністю під контролем розробника – фреймворк не нав’язує жорстких правил. Його легко інтегрувати з базами даних, такими як MongoDB, PostgreSQL або SQLite. Express часто використовується як основа для побудови масштабованих бекенд-сервісів та мікросервісної архітектури.

2.4 Середовище розробки Visual Studio Code

Visual Studio Code (VS Code) – це сучасне, легке та водночас потужне середовище розробки, створене компанією Microsoft. Воно працює на більшості популярних операційних систем, включно з Windows, Linux і macOS, і є одним із найпопулярніших інструментів серед розробників завдяки поєднанню гнучкості, високої продуктивності та широких можливостей розширення. Основна перевага VS Code полягає в тому, що воно підтримує безліч мов програмування та технологій із самого початку або за допомогою численних розширень, які можна легко встановити через вбудований маркетплейс.

Це середовище забезпечує інтелектуальне автодоповнення коду, контекстні підказки, навігацію по файлах та символах, а також інтегровану перевірку помилок, що значно пришвидшує процес розробки. Особливістю VS Code є наявність вбудованого терміналу та налагоджувача, що дозволяє розробникам тестувати і запускати свій код безпосередньо в редакторі, не перемикаючись між додатковими інструментами. Інтеграція з системами контролю версій, зокрема Git, також реалізована на високому рівні: можна створювати коміти, гілки, переглядати історію змін та проводити злиття змін у зручному графічному інтерфейсі.

У нашому проєкті Visual Studio Code використовувався як основне середовище для розробки серверної частини системи балансування навантаження. Завдяки зручному інтерфейсу, підтримці TypeScript і Node.js, розробка була швидкою та організованою.

Під час написання коду також активно використовувались розширення для форматування, lint-аналізу, автодоповнення, а також для взаємодії з базою даних PostgreSQL безпосередньо з редактора. Це дозволило працювати з усіма компонентами системи в одному вікні, що позитивно вплинуло на ефективність розробки.

2.5 pg.js

pg – це офіційна бібліотека клієнта PostgreSQL для Node.js, яка забезпечує ефективну, надійну та функціонально повну взаємодію з базою даних PostgreSQL. Вона дозволяє виконувати SQL-запити безпосередньо з Node.js, підтримує як прості запити, так і підготовлені інструкції, транзакції, пул з'єднань і обробку помилок. Завдяки своїй архітектурі, pg працює асинхронно, використовуючи проміси або зворотні виклики, що добре поєднується з неблокуючою природою Node.js і дозволяє ефективно масштабувати додатки.

Бібліотека забезпечує простий інтерфейс для створення підключень до бази даних, конфігурації параметрів (як-от хост, порт, користувач, пароль і назва бази), а також дозволяє використовувати пул з’єднань для оптимізації продуктивності у високонавантажених середовищах. pg підтримує виконання складних запитів із параметрами, що зменшує ризики SQL-ін’єкцій і підвищує безпеку системи. Крім того, вона має можливість роботи з потоками результатів (streaming), що корисно при обробці великих обсягів даних.

У рамках нашого проєкту, бібліотека pg використовувалася для реалізації частини, що відповідає за збереження та отримання інформації про активні сервери, баланс навантаження та облік історії запитів. Завдяки її стабільності та широкому функціоналу, вдалося легко реалізувати доступ до бази, провести міграції структури таблиць і забезпечити ефективну взаємодію між балансувальником і базою даних.

2.6 PostgreSQL

PostgreSQL – це об'єктно-реляційна система керування базами даних (СКБД), яка підтримує як традиційні реляційні таблиці, так і складніші типи даних, включаючи JSON, XML, масиви та користувацькі типи. Вона забезпечує повну підтримку транзакцій за принципами ACID, що гарантує цілісність і надійність даних. PostgreSQL використовує багатоверсійну модель контролю паралелізму (MVCC), яка дозволяє одночасно виконувати читання і запис без блокування. Система підтримує складні SQL-запити, вбудовані функції, індекси (включаючи GIN і GiST), представлення (views), матеріалізовані представлення та тригери. Завдяки потужній системі розширень

PostgreSQL дозволяє додавати нові функції, зокрема за допомогою популярного розширення PostGIS для роботи з геоданими. Підключення до бази відбувається через стандартні інтерфейси, такі як psql, JDBC, або бібліотеки для мов програмування (наприклад, pg для Node.js). PostgreSQL підтримує одночасну роботу багатьох користувачів і має розвинену систему прав доступу. Завдяки механізму реплікації база може масштабуватись горизонтально для підвищення продуктивності. Вона також підтримує збережені процедури на різних мовах, включаючи SQL, PL/pgSQL, Python і інші.

PostgreSQL активно використовується для зберігання складних структурованих даних у веб-застосунках, мікросервісах, аналітичних системах та великих корпоративних рішеннях.

2.7 Nginx

Nginx – це високопродуктивний веб-сервер і зворотний проксі-сервер, який також може працювати як балансувальник навантаження та кешуючий проксі. Він розроблений для обробки великої кількості одночасних з'єднань

із мінімальним використанням ресурсів завдяки подієво-орієнтованій, асинхронній архітектурі. [3] Nginx здатен обслуговувати десятки тисяч одночасних клієнтів з використанням лише кількох потоків, що робить його ідеальним для високонавантажених систем.

Nginx підтримує розподілення навантаження між кількома серверами за алгоритмами round-robin, least connections, або IP-hash. Він дозволяє створювати правила для кешування відповідей, що зменшує кількість запитів до бекенду та пришвидшує завантаження. Сервер також має потужну систему конфігурацій, де можна налаштувати редиректи, переписування URL, обмеження доступу, захист від DDoS та інші параметри. [6]

Nginx підтримує SSL/TLS-шифрування, включаючи сучасні алгоритми та автоматичну інтеграцію з Let's Encrypt. Завдяки малій вазі та високій ефективності, він підходить як для невеликих вебсайтів, так і для великих розподілених систем. У дипломному проекті Nginx використовується як балансувальник навантаження для розподілу запитів між декількома екземплярами Node.js-сервера.

2.8 Docker

Docker – це платформа для контейнеризації застосунків, яка дозволяє запускати програмне забезпечення в ізольованому середовищі, званому контейнером. Кожен контейнер містить усе необхідне для виконання застосунку: код, бібліотеки, залежності та системні інструменти. [4] Це забезпечує однакову поведінку застосунку незалежно від середовища, у якому він запускається.

Docker використовує образи (images), які є шаблонами для створення контейнерів. Образи описуються у файлах Dockerfile, де можна вказати базову систему, залежності, конфігурацію мережі, команди запуску тощо. Завдяки кешуванню шарів, побудова образів виконується швидко, а зміни вносяться лише у змінені шари.

Docker Engine відповідає за створення, запуск і управління контейнерами. Контейнери взаємодіють між собою через віртуальні мережі, які створює Docker, зокрема bridge, host, або overlay у випадку з Docker Swarm.

Для збереження даних, які повинні залишатися між перезапусками, Docker використовує томи (volumes) або bind-монтування. [7] Docker також дозволяє масштабувати систему, розгортаючи багато контейнерів одночасно, що зручно для мікросервісної архітектури.

У веб-розробці Docker дозволяє легко запускати середовища для Node.js, Nginx, PostgreSQL, Redis тощо без необхідності локального встановлення всіх компонентів. У дипломному проекті Docker використовується для розгортання окремих екземплярів веб-сервера Node.js, що спрощує тестування роботи балансувальника навантаження (Nginx) в умовах ізольованого середовища.

Також з його допомогою можна симулювати велику кількість інстансів та керувати ними централізовано. Завдяки підтримці стандарту OCI, Docker-сумісні контейнери можуть бути використані у багатьох хмарних системах, таких як Kubernetes чи Amazon ECS. [8]

2.9 hey

Hey – це сучасний інструмент для навантажувального тестування HTTP-серверів, створений як спрощена альтернатива Apache Benchmark, але з покращенням продуктивності та масштабованості. Розроблений мовою Go, hey орієнтований на високу швидкодію, мінімальне споживання ресурсів та простоту у використанні.

На відміну від свого попередника ab, hey підтримує багатопоточну генерацію запитів: кожен робочий потік працює незалежно, створюючи власні HTTP-з’єднання, що дозволяє ефективно використовувати багатоядерні процесори. Це особливо важливо для тестів, де необхідно

зімітувати поведінку великої кількості паралельних клієнтів. Утиліта дозволяє конфігурувати кількість запитів, кількість одночасних потоків, час очікування відповіді, HTTP-заголовки, методи запиту, авторизацію та тіло запиту. [12]

Hey забезпечує вивід ключової статистики: середній час відповіді, середню пропускну здатність (requests per second), помилки з'єднання, кількість збоїв, а також відсотковий розподіл часу відповіді (перцентилі). Хоча функціонально hey не підтримує складні сценарії тестування на основі скриптів, його швидкодія та простота запуску роблять його ідеальним інструментом для швидких і надійних перевірок, особливо під час розробки або при CI/CD-інтеграції.

У межах дипломного проєкту hey використовується для оцінки початкової продуктивності веб-сервера та перевірки ефективності базової конфігурації балансувальника навантаження. Його простота дозволяє швидко отримати перші вимірювання та проаналізувати вплив параметрів конфігурації на поведінку системи під навантаженням.

Таким чином, hey у нашому дослідженні виконує роль «легкого стартера» – інструменту для швидкого порівняння продуктивності конфігурацій до та після змін. Він добре масштабується у багатоядерних середовищах, забезпечує стабільну роботу навіть при високому рівні паралелізму та ідеально підходить для інтеграційних сценаріїв. Його обмеження у функціональності не критичні для завдань, орієнтованих на аналіз продуктивності та пропускної здатності балансувальника запитів.

2.10 wrk

wrk – це високопродуктивний інструмент для навантажувального тестування веб-серверів, який дозволяє створювати значне одночасне навантаження за допомогою багатопотокової архітектури. Він підтримує генерацію тисяч запитів за секунду, використовуючи кілька потоків і

подієво-орієнтований цикл для максимально ефективного використання ресурсів CPU. Користувач може налаштовувати кількість потоків, загальну кількість запитів, тривалість тесту та рівень одночасних з’єднань. [11]

wrk підтримує Lua-скрипти для кастомізації поведінки запитів, що дозволяє моделювати складні сценарії, включно з різними HTTP-методами, заголовками, тілом запиту і авторизацією. Після завершення тестування утиліта надає детальну статистику, включаючи середню пропускну здатність (requests per second), час відповіді, розподіл часу по перцентилях і загальну кількість успішних та неуспішних запитів.

wrk часто використовується для тестування веб-серверів, балансувальників навантаження та API, особливо коли потрібно виміряти поведінку системи під високим реальним навантаженням. У дипломному проекті цей інструмент дозволяє створити інтенсивне паралельне навантаження на Nginx як балансувальник і перевірити його здатність ефективно розподіляти запити між екземплярами Node.js. Завдяки гнучкості конфігурації і високій продуктивності wrk є одним з основних інструментів для тестування продуктивності сучасних веб-застосунків.

3 ПРОГРАМНА РЕАЛІЗАЦІЯ

У цьому розділі детально описано структуру та логіку роботи розробленого проекту з особливим акцентом на паралельну обробку запитів.

В описаній системі клієнтські запити розподіляються між кількома інстансами Node.js-додатку, що працюють як мікросервіси з базою даних PostgreSQL, за допомогою Nginx як реверс-проксі та балансувальника навантаження. Nginx у ролі реверс-проксі перехоплює всі вхідні HTTP-запити, кешує статичний контент, виконує стиснення відповідей (gzip) і розподіляє трафік між бекенд-серверами за заданим алгоритмом. Таке рішення дозволяє підвищити продуктивність і стійкість сервісу: Nginx може балансувати навантаження, згладжувати піки запитів і обробляти SSL/HTTP-запити, звільняючи Node.js-сервери від обробки статичних даних та оптимізації мережі. Кожен екземпляр Node.js-програми запускається у власному Docker-контейнері з окремим оточенням і налаштуванням (через Dockerfile та docker-compose).

Щоб динамічно змінювати кількість запущених Node.js-контейнерів залежно від навантаження, використовуються Bash-скрипти для автоскейлінгу. Один скрипт (scale.sh) періодично вимірює завантаження CPU поточних контейнерів і запускає розгортання чи зупинку додаткових реплік при перевищенні порогів; інший (regenerate-nginx.sh) формує файл конфігурації Nginx на основі фактично запущених контейнерів, а третій (daemon.sh) організовує нескінченний цикл опитування й масштабування.

3.1 Серверний застосунок на Node.js

Серверний застосунок складається з декількох етапів, а саме ініціалізація, під’єднання до бази даних, створення ендпоїнтів та запуск серверу.

3.1.1 Ініціалізація Express та імпорт модулів

Цей код створює простий HTTP API-сервер на Node.js з використанням Express та PostgreSQL, а також додаємо функціонал автоматичного парсингу даних в форматі JSON (лістинг 3.1).

Лістинг 3.1 – Виконання інструкції

```
const express = require('express');
const { Pool } = require('pg');

const app = express();
app.use(express.json());
```

3.1.2 Конфігурація пулу з'єднань PostgreSQL

Ініціалізація підключення до бази даних PostgreSQL через пул з’єднань. Налаштування беруться зі змінних середовища для зручності конфігурації в Docker (лістинг 3.2).

Лістинг 3.2 – Конфігурація PostgreSQL

```
const pool = new Pool({
    user: process.env.PGUSER,
    host: process.env.PGHOST,
    database: process.env.PGDATABASE,
    password: process.env.PGPASSWORD,
    port: process.env.PGPORT,
});
```

3.1.3 Ініціалізація бази даних

Функція створює таблицю requests, якщо вона ще не існує. У таблиці зберігаються текстові запити з міткою часу створення (лістинг 3.3).

Лістинг 3.3 – Ініціалізація бази даних

```
async function initializeDB()
{
```

```
    await pool.query(`
    CREATE TABLE IF NOT EXISTS requests (
      id SERIAL PRIMARY KEY,
      data TEXT NOT NULL,
      created_at TIMESTAMP DEFAULT NOW()
    )
  `);
}
```

3.1.4 Ендпоїнт GET /data

Обробник GET-запиту до /data ендпоїнту, а потім повертає останні 100 записів з бази даних або повертає помилку серверу (лістинг 3.4).

Лістинг 3.4 – Ендпоїнт GET /data

```
app.get('/data', async (req, res) =>
{
    try {
        const result = await pool.query(‘
          SELECT * FROM requests LIMIT 100
    ’);

        res.json(result.rows);
    } catch (err) {
        res.status(500).send('Server error');
    }
});
```

3.1.5 Ендпоїнт POST /data

Обробник POST-запиту до /data ендпоїнту, що дозволяє користувачу відправити дані до таблиці requests та повертає ці дані як відповідь або повертає помилку серверу (лістинг 3.5).

Лістинг 3.5 – Ендпоїнт POST /data

```
app.post('/data', async (req, res) => {
    const { data } = req.body;
    try {
        const result = await pool.query(
            'INSERT INTO requests (data) VALUES ($1) RETURNING
```

```
*',
            [data]
        );
        res.status(201).json(result.rows[0]);
    } catch (err) {
        res.status(500).send('Server error');
    }
});
```

3.1.6 Ендпоїнт Health-Check

Обробник маршруту /health повертає статус 200 і використовується для перевірки здоров’я сервісу (лістинг 3.6).

Лістинг 3.6 – Ендпоїнт Health-Check

```
app.get('/health', (req, res) => {
    res.status(200).send('OK');
});
```

3.1.7 Запуск сервера

Ініціалізація бази даних і запуск серверу на порту 3000 після успішного підключення до БД (лістинг 3.7).

Лістинг 3.7 – Запуск сервера

```
initializeDB().then(() => {
    app.listen(3000, () => console.log('Server running on port
3000'));
});
```

3.2 Конфігурація Nginx

Файл конфігурації Nginx налаштовує зворотній проксі-сервер для розподілу вхідного трафіку між backend-сервісами, що працюють у Docker-контейнерах. Також реалізовано механізм перевірки здоров’я сервісів через окремий маршрут /lb-health, що дозволяє використовувати Nginx як

балансувальник навантаження з простим health-check (лістинг 3.7).

3.2.1 Секція events

Секція events визначає основні параметри обробки подій у Nginx. У цьому випадку вказано, що кожен процес може обробляти до 64 одночасних з’єднань (лістинг 3.8).

Лістинг 3.8 – Секція events

```
events { worker_connections 64 }
```

3.2.2 Секція http > upstream

Секція http оголошує групу серверів під назвою backend_servers, яка використовується для балансування навантаження. Кожен запит буде надсилатися до контейнера backend, що слухає порт 3000. Якщо контейнер тричі поспіль не відповість на запити, то він тимчасово виключається з обробки запитів на 30 секунд (лістинг 3.9).

Лістинг 3.9 – Секція http > upstream

```
http {
    upstream backend_servers {
        server backend:3000 max_fails=3 fail_timeout=30s;
    }
```

3.2.3 Секція location /

Секція location / відповідає за обробку всіх звичайних запитів. Всі такі запити перенаправляються на upstream-групу backend_servers, яка відповідає за їх обробку. Використовується HTTP/1.1 для підтримки keep-alive-з’єднань. Заголовок Connection скидається, щоб уникнути конфліктів. У разі помилок або таймаутів, таких як 502, 503 чи 504, запит автоматично буде повторно

відправлений на інший сервер у пулі. Таймаут на встановлення з'єднання з бекендом обмежено однією секундою, а очікування відповіді – п'ятьма секундами (лістинг 3.10).

Лістинг 3.10 – Секція location /

```
location / {
    proxy_pass          http://backend_servers;

    proxy_http_version 1.1;
    proxy_set_header   Connection "";

    proxy_next_upstream error timeout http_502 http_503
http_504;

    proxy_connect_timeout 1s;
    proxy_read_timeout    5s;
}
```

3.2.4 Секція location = /lb-health

Секція location = /lb-health визначає спеціальний маршрут, який використовується системою для перевірки доступності backend-контейнерів. Запит надсилається на /health одного з серверів у пулі. Для цього запиту встановлені більш короткі таймаути: одна секунда на з'єднання та дві секунди на отримання відповіді, що дозволяє швидко виявити непрацездатний інстанс (лістинг 3.11).

Лістинг 3.11 – Секція location = /lb-health

```
location = /lb-health {
    proxy_pass          http://backend_servers/health;
    proxy_http_version  1.1;
    proxy_set_header    Connection "";
    proxy_connect_timeout 1s;
    proxy_read_timeout    2s;
}
```

3.3 Конфігурація Docker-compose

Файл docker-compose.yml описує інфраструктуру з трьох сервісів: бекенда на Node.js, бази даних PostgreSQL та балансувальника навантаження Nginx. Кожен компонент конфігуровано для взаємодії в спільній мережі, а також для обмеження ресурсів, моніторингу стану та збереження даних бази.

3.3.1 Сервіс backend

Сервіс backend будується з Dockerfile, розташованого у папці ./backend. Він запускається лише після того, як буде готовий сервіс postgres, що вказано через директиву depends_on. Зовнішній порт не проброшено (використовується лише всередині мережі), а середовище містить змінні для підключення до бази даних (лістинг 3.12).

Лістинг 3.12 – Сервіс backend

```
backend:
    build:
      context: ./backend
    depends_on:
      - postgres

    ports:
      - "3000"
    environment:
      - PGHOST=postgres
      - PGUSER=postgres
      - PGPASSWORD=examplepw
      - PGDATABASE=testdb
      - PGPORT=5432

    networks:
      - appnet
```

3.3.2 Сервіс postgres

Сервіс postgres використовує офіційний образ PostgreSQL версії 16.

Йому задаються параметри створення бази даних та користувача через змінні середовища. Дані бази зберігаються у томі postgres-data, щоб не втрачались після перезапуску контейнера. Він також підключений до спільної мережі appnet (лістинг 3.13).

Лістинг 3.13 – Сервіс postgres

```
postgres:
    image: postgres:16
    environment:
      - POSTGRES_USER=postgres
      - POSTGRES_PASSWORD=examplepw
      - POSTGRES_DB=testdb

    volumes:
      - postgres-data:/var/lib/postgresql/data
    networks:
      - appnet
```

3.3.3 Сервіс nginx

Сервіс nginx працює як балансувальник навантаження. Його контейнер названо явно як nginx-lb. Після запуску він відкриває порт 80 для зовнішнього доступу. Використовується конфігурація з локального файлу nginx.conf, який монтується в контейнер лише для читання. Healthcheck перевіряє доступність шляху /lb-health, щоб переконатися, що backend працює (лістинг 3.14).

Лістинг 3.14 – Сервіс nginx

```
nginx:
    image: nginx:latest
    container_name: nginx-lb
    depends_on:
      - backend
    ports:
      - "80:80"
    healthcheck:
      test: ["CMD", "curl", "-f", "http://localhost/lb-health"]
      interval: 15s
      timeout:  5s
      retries:  3
```

```
volumes:
  - ./nginx/nginx.conf:/etc/nginx/nginx.conf:ro
networks:
  - appnet
```

4 РЕЗУЛЬТАТИ ВИКОРИСТАННЯ ПАРАЛЕЛІЗАЦІЇ

Під час реалізації експериментального веб-сервісу було протестовано вплив горизонтального масштабування на продуктивність системи. Основна задача полягала у виявленні вузьких місць при одночасному обслуговуванні великої кількості клієнтських запитів. З цією метою було проведено серію навантажувальних тестів із використанням інструментів wrk та hey. [9]

4.1 Тестування за допомогою wrk

Для моделювання навантаження використовувалося поєднання 10 потоків та 100 з’єднань з тривалістю 30 секунд. У конфігурації без масштабування спостерігалося значне зростання латентності вже після кількох тисяч одночасних запитів (таблиця 4.1).

Таблиця 4.1 – Результати тестування з wrk

Параметр	Без масштабування	З масштабуванням	Зміна (%)
Кількість запитів (за 30 сек)	358 315	703 270	+96.3%
Запити/сек	~11 944	~23 442	+96.3%
Середня латентність	25.45 мс	7.53 мс	-70.4%
Відсоток Non-2xx відповідей	~70%	0%	-70%
Об’єм переданих даних	54.2 МБ	106.1 МБ	+95.7%

Відповіді надходили швидше, без втрат, а пропускна здатність зросла майже вдвічі. Більшість відповідей мали статуси помилок через таймаути та перевантаження сокетів. Після масштабування системи до трьох екземплярів backend-сервісу, які обслуговувались nginx, значна частина цих проблем зникла. Висновок для цієї частини. Після масштабування система стала здатна обробляти вдвічі більше трафіку при одночасному скороченні латентності майже втричі. Це чітко демонструє ефективність кластерної обробки на рівні інфраструктури.

4.2 Тестування за допомогою hey

Тестування проводилося з частотою 10 запитів на з'єднання та 100 одночасними клієнтами протягом 30 секунд. Без масштабування сервіс обробив лише 32 000 запитів, демонструючи значні затримки, а частина відповідей мала некоректний статус. Після масштабування кількість оброблених запитів зросла більш ніж утричі, а середній час відповіді скоротився вдвічі (таблиця 4.2).

Таблиця 4.2 – Результати тестування з hey

Параметр	Без масштабування	З масштабуванням	Зміна (%)
Кількість запитів (за 30 сек)	32 025	108 308	+238%
Запити/сек	~1067	~3610	+238%
Середній час	265 мс	84 мс	-68.3%
Максимальна стабільна кількість клієнтів	~35	~110	+214%
Відсоток Non-2xx відповідей	~100%	0%	-100%

Висновок для цієї частини. Під час імітації сталої клієнтської активності сервіс з масштабуванням втричі перевершив продуктивність оригінальної конфігурації. Результати hey підтверджують стабільність та придатність сервісу до роботи в умовах реального навантаження.

4.3 Загальний підсумок

Результати тестування із використанням двох незалежних інструментів (wrk та hey) підтверджують, що горизонтальне масштабування сервісу шляхом запуску кількох екземплярів backend-компонентів суттєво підвищує загальну продуктивність. Усі ключові показники – середній RPS, латентність, стабільність – продемонстрували помітне покращення. Окрім того, усунення Non-2xx відповідей свідчить про те, що система здатна надійно функціонувати навіть у пікових умовах без втрат.

Цей підхід не потребує внесення змін до логіки застосунку, а лише коригування інфраструктури, що робить його зручним для масштабування в умовах зростання навантаження. Застосування паралельної обробки запитів на рівні контейнерів, підкріплене балансуванням, є ефективним способом досягнення високої продуктивності в сучасних веб-системах.

4.4 Порівняння з конкурентними рішеннями

Для об’єктивного аналізу отриманих результатів доцільно порівняти обрану реалізацію балансувальника навантаження на базі Nginx з іншими популярними рішеннями, такими як HAProxy, Traefik та Envoy Proxy. Усі ці системи мають власні архітектурні особливості, що безпосередньо впливають на масштабованість, підтримку паралелізації, гнучкість конфігурації та швидкодію (таблиця 4.3). HAProxy є класичним рішенням корпоративного рівня. Його ключовою перевагою є максимальна продуктивність у високонавантажених середовищах завдяки оптимізації на рівні ядра системи.

Він підтримує повноцінну багатопотоковість і дозволяє адміністраторам точно контролювати розподіл запитів між ядрами CPU. Проте цей інструмент вимагає глибоких знань мережевої архітектури, має складну конфігурацію і не інтегрується автоматично з контейнерними середовищами.

Traefik, навпаки, пропонує простоту й автоматизацію, що ідеально підходить для хмарних платформ та DevOps-практик. Його архітектура базується на мові Go, що дозволяє асинхронно обробляти запити та автоматично виявляти сервіси у кластері. Але внаслідок високого рівня абстракції Traefik поступається в продуктивності при інтенсивному трафіку або складній маршрутизації.

Envoy Proxy вважається одним із найбільш функціональних сучасних проксі для мікросервісної архітектури. Він підтримує паралельну обробку, трасування запитів, TLS-термінацію, вбудовані фільтри, а також має повну підтримку HTTP/2 та gRPC. Однак складність конфігурації та необхідність зовнішнього контролера робить його менш зручним для невеликих проєктів.

Рішення на основі Nginx, яке було обране у нашому проєкті, забезпечує баланс між простотою впровадження, продуктивністю та зрозумілістю. Nginx не вимагає складних конфігурацій, підтримує масштабування через просту зміну кількості backend-контейнерів та дозволяє адаптувати логіку маршрутизації до поточного навантаження. У той же час, Nginx не надає такої гнучкості в алгоритмах балансування або глибини телеметрії, як Envoy, але цілком придатний для задач малого і середнього рівня складності.

Таким чином, обране рішення на базі Nginx добре підходить для середніх за складністю проектів, де важлива швидка інтеграція, стабільність і контрольована продуктивність. Для великих кластеризованих систем із вимогами до трасування, зворотного зв’язку з бекендами та адаптивного балансування доцільно розглядати Envoy або HAProxy. Traefik, своєю чергою, залишається кращим вибором для DevOps-команд, орієнтованих на автоматизацію розгортання.

Таблиця 4.3 – Порівняння з конкурентами

Характеристика	HAProxy	Traefik	Envoy Proxy	Наш проект
Простота налаштування	Низька	Висока	Низька	Висока
Продуктивність	Висока	Середня	Висока	Висока
Підтримка паралелізації	Так (CPU-рівень)	Так (Go-threads)	Так (I/O threads)	Частково (через масштабування)
Інтеграція з Docker	Часткова	Повна	Часткова	Повна
Моніторинг і логування	Базовий	Обмежений	Розширений	Базовий
Гнучкість конфігурації	Висока	Середня	Дуже висока	Середня

ВИСНОВКИ

У результаті виконання дипломної роботи було проведено комплексне дослідження можливостей використання паралельних обчислень для підвищення продуктивності прикладних веб-застосунків, зокрема через реалізацію балансувальника навантаження на основі Nginx та масштабування серверної частини через мультиконтейнерну інфраструктуру. Основна увага приділялася практичному вивченню того, як впровадження механізмів паралельної обробки запитів може вплинути на швидкодію, стабільність і масштабованість системи.

У процесі розробки використовувалися сучасні інструменти: серверна логіка була реалізована на платформі Node.js з використанням фреймворку Express, а засоби контейнеризації Docker дозволили забезпечити ізольованість середовищ та спростити процес розгортання і тестування.

Під час тестування системи на навантаження застосовувалися два незалежних інструменти: wrk – високопродуктивний генератор навантаження на основі багатопоточності, що дозволяє проводити глибокий аналіз затримок, та hey – простіший, але також інформативний клієнтський засіб для оцінки HTTP-продуктивності. Це дало змогу отримати комплексну картину реальної поведінки системи в умовах стресового навантаження. На етапі базового тестування, коли система запускалася з одним екземпляром бекенду, спостерігалися значні затримки у відповідях і зниження кількості успішно оброблених запитів.

Після впровадження масштабування – шляхом підняття кількох контейнерів із серверною логікою та додавання балансувальника було зафіксовано суттєве покращення всіх основних показників: знизився середній час відповіді, зросла кількість оброблених запитів у секунду, а система загалом стала більш стабільною. Отримані результати демонструють значущість навіть базових рішень з паралелізації для підвищення

ефективності веб-сервісів. Разом із тим, варто зазначити, що база даних PostgreSQL залишалася єдиною спільною точкою для всіх контейнерів, і не підлягала масштабуванню. Це стало потенційним вузьким місцем у системі, оскільки при зростанні кількості екземплярів бекенду навантаження на БД зростає експоненційно. У межах проєкту цей аспект не був вирішений, однак можливими напрямками покращення є реалізація реплікації PostgreSQL, розподілення навантаження на читання та застосування кешування результатів запитів на стороні API.

Крім архітектурних рішень, значну роль у досягненні стабільної роботи системи відіграло впровадження контейнеризації за допомогою Docker. Незважаючи на те, що запуск багатьох інстансів у контейнерах вимагає додаткових системних ресурсів, таких як пам’ять, CPU і доступ до файлової системи, витрати на контейнеризацію виявилися виправданими. Ізоляція середовищ дозволила зменшити ризики конфліктів, спростити масштабування, пришвидшити розгортання та забезпечити зручний моніторинг роботи окремих компонентів. Контейнерна структура також забезпечує високу гнучкість при експериментах із конфігурацією, дозволяючи швидко вносити зміни в архітектуру без ризику порушити роботу всієї системи.

Подальший розвиток цієї теми передбачає поглиблене дослідження більш складних стратегій маршрутизації, адаптивного балансування залежно від навантаження в реальному часі, а також інтеграції із зовнішніми моніторинговими платформами та хмарними оркестраторами.

Враховуючи отримані результати, впровадження паралельного виконання через масштабовану архітектуру може бути не лише корисним, але й необхідним кроком у розробці високонавантажених і відмовостійких веб-сервісів. Виконана робота доводить, що навіть при використанні відкритих і відносно простих інструментів можна суттєво покращити показники продуктивності, і вона може слугувати практичною основою для подальших реальних впроваджень у сфері масштабованих веб-систем.

ПЕРЕЛІК ДЖЕРЕЛ ПОСИЛАННЯ

1. Flanagan D. JavaScript: The Definitive Guide / D. Flanagan. – Sebastopol, CA: O'Reilly Media, 2020. – 1096 p. – ISBN 978-1491952023.

2. Tilkov S., Vinoski S. Node.js: The Right Way / S. Tilkov, S. Vinoski. – New York: Pragmatic Bookshelf, 2018. – 240 p. – ISBN 978-1680501959.

3. Nginx Docs Team. NGINX HTTP Server: Beginner’s Guide / Nginx Docs Team. – San Francisco: O'Reilly Media, 2019. – 300 p. – ISBN 978-1492044343.

4. Merkel D. Docker: Up & Running: Shipping Reliable Containers in Production / D. Merkel. – Sebastopol, CA: O'Reilly Media, 2018. – 320 p. – ISBN 978-1491917572.

5. Node.js Documentation, OpenJS Foundation, 2024. – https://nodejs.org/en/docs

6. NGINX Official Documentation, F5 Inc., 2024. – https://docs.nginx.com/nginx/

7. Docker Documentation, Docker Inc., 2024. – https://docs.docker.com/

8. Burns B., Grant B., Oppenheimer D. Kubernetes: Up and Running / B. Burns et al. – Sebastopol, CA: O'Reilly Media, 2022. – 360 p. – ISBN 978-1492046530.

9. Gulbrandsen M., Kalvenes J. Load Testing for DevOps / M. Gulbrandsen, J. Kalvenes. – Berkeley, CA: Apress, 2021. – 250 p. – ISBN 978-1484270642.

10. Levis A. Scalable Web Architecture and Distributed Systems / A. Levis. – ACM Queue, 2018. – Vol. 16, No. 5.

11. wrk – A HTTP benchmarking tool, GitHub, 2024. – https://github.com/wg/wrk

12. hey – HTTP load generator, Apache Benchmark replacement, GitHub, 2024. – https://github.com/rakyll/hey