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 ПРОДУКТИВНОСТІ ВЕБЗАСТОСУНКІВ ПРИ ІНТЕГРАЦІЇ СЕРВІСІВ ШТУЧНОГО ІНТЕЛЕКТУ У РЕЖИМІ РЕАЛЬНОГО ЧАСУ
(тема)

Виконав:
здобувач 2 року навчання,
групи ІНФм-24-2
Ніколаєнко В. О.
(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. Руденко Д.О.
(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 Ніколаєнко В'ячеславу Олексійовичу
(прізвище, ім'я, по батькові)

1. Тема роботи Дослідження продуктивності вебзастосунків при інтеграції сервісів штучного інтелекту у режимі реального часу

затверджена наказом університету від 14 листопада 2025 року № 1045Ст
2. Термін подання здобувачем роботи до екзаменаційної комісії 09 грудня 2025 р.
3. Вихідні дані до роботи наукова-технічна література, матеріали наукових конференцій, середовище розробки Visual Studio Code, обгортка для мови програмування TypeScript, фреймворк для фронтенду і бекенду Next.js, безсерверний бекенд Firebase.

4. Перелік питань, що потрібно опрацювати в роботі
1. Дослідження сучасних підходів до побудови вебзастосунків.
2. Аналіз літературних джерел щодо методів інтеграції AI-сервісів у вебзастосунки.
3. Визначення метрик продуктивності та побудова узагальненої моделі оцінювання швидкодії вебзастосунків з AI-компонентами.
4. Розроблення та дослідження експериментального прототипу для вимірювання часу генерації відповідей.
5. Формування рекомендацій щодо вибору архітектури та технологій для створення продуктивних AI-орієнтованих вебзастосунків.

5. Перелік графічного матеріалу із зазначенням креслеників, схем, плакатів, комп'ютерних ілюстрацій (п.5 включається до завдання за рішенням випускової кафедри) актуальність проблеми інтеграції сервісів штучного інтелекту в вебзастосунки реального часу, об'єкт та мета дослідження, постановка задачі, висновки.

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	15.10.25-22.10.25	
3	Аналіз літератури з досліджуваної проблеми	23.10.25-28.10.25	
4	Методи інтеграції AI-сервісів	29.10.25-05.11.25	
6	Програмна реалізація	16.11.25-18.11.25	
7	Обґрунтування отриманих результатів	19.11.25-20.11.25	
8	Оформлення пояснювальної записки	21.11.25-07.12.25	
9	Перевірка на нормоконтроль	07.12.25-13.12.25	
10	Перевірка на плагіат	13.12.25-15.12.25	
11	Рецензування	16.12.2025	
12	Підготовка презентації та доповіді	17.12.25-18.12.25	
13	Занесення роботи в електронний архів	19.12.2025	
14	Попередній захист кваліфікаційної роботи	22.12.2025	

Дата видачі завдання 29 вересня 2025 р.

Здобувач _____
(підпис)

Керівник роботи _____ Кобилін О. А.
(підпис) (посада, прізвище, ініціали)

РЕФЕРАТ

Пояснювальна записка до кваліфікаційної роботи: 64 с., 6 табл., 3 рисунки, 33 джерела.

AI-СЕРВІСИ, ІНТЕГРАЦІЯ, ВЕБЗАСТОСУНКИ, МІКРОСЕРВІСИ, МОНОЛІТ, ПРОДУКТИВНІСТЬ, МАСШТАБОВАНІСТЬ, РЕАЛЬНИЙ ЧАС, LATENCY, REST API, WEBSOCKET, gRPC, LLM.

Об’єктом дослідження є процес функціонування вебзастосунків, що взаємодіють з AI-сервісами у режимі реального часу.

Метою роботи є аналіз впливу архітектури застосунку, типу інтеграції та стратегії обробки на продуктивність системи під час виконання AI-обчислень.

У роботі проведено огляд сучасних моделей інтеграції, класифіковано види AI-сервісів та досліджено їхні обчислювальні характеристики. Запропоновано математичні моделі часу відповіді та впливу паралельних запитів, виконано порівняння архітектурних підходів й оцінено їх масштабованість.

Наукова новизна роботи полягає у формалізації коефіцієнта ефективності інтеграції AI-компонент та у комплексному моделюванні поведінки вебзастосунків при реальному навантаженні.

Практична значущість полягає у можливості застосування отриманих рекомендацій для проєктування систем, що працюють із LLM- моделями та іншими AI-сервісами у режимі реального часу.

У результаті створено експериментальну модель інтеграції, проведено аналіз продуктивності та сформульовано рекомендації щодо вибору архітектурних та технологічних рішень для AI-орієнтованих вебзастосунків.

ABSTRACT

Explanatory note to the qualification work: 64 pages, 6 tables, 3 figures, 33 sources.

AI SERVICES, INTEGRATION, WEB APPLICATIONS, MICROSERVICES, MONOLITH, PERFORMANCE, SCALABILITY, REAL-TIME SYSTEMS, LATENCY, REST API, WEBSOCKET, gRPC, LLM.

The object of the research is the functioning of web applications that interact with AI services in real time.

The aim of the work is to analyze how application architecture, integration type, and processing strategy influence system performance during AI-related computations.

The study reviews modern integration models, classifies AI-service types, and examines their computational characteristics. Mathematical models of response time and the influence of parallel request processing are proposed, while architectural approaches are compared and evaluated in terms of scalability.

The scientific novelty lies in the formalization of an AI-integration efficiency coefficient and in comprehensive modeling of web application behavior under real-world load.

The practical significance consists in the applicability of the recommendations for designing systems that operate with LLM models and other AI services in real-time environments.

As a result, an experimental integration model was developed, system performance was analyzed, and recommendations were formulated for selecting architectural and technological solutions for AI-oriented web applications.

ЗМІСТ

ПЕРЕЛІК УМОВНИХ ПОЗНАЧЕНЬ, СИМВОЛІВ, ОДИНИЦЬ, СКОРОЧЕНЬ І ТЕРМІНІВ

AI – Artificial Intelligence

API – Application Programming Interface

LLM – Large Language Model

REST – Representational State Transfer

gRPC – Google Remote Procedure Call

WebSocket – протокол двостороннього обміну даними в реальному часі

SSE – Server-Sent Events

WebRTC – Web Real-Time Communication

CRUD – Create, Read, Update, Delete

JSON – JavaScript Object Notation

HTTP – HyperText Transfer Protocol

Kafka – розподілена система обміну повідомленнями

RabbitMQ – брокер повідомлень для асинхронної обробки даних

Edge-AI – виконання AI-обчислень на периферійних вузлах

Gateway-AI – модель взаємодії з AI-сервісом через проміжний шлюз

Serverless – архітектура, у якій логіка застосунку виконується у вигляді незалежних функцій без постійного сервера

ВСТУП

Сучасні вебзастосунки дедалі частіше інтегрують технології штучного інтелекту для підвищення зручності користування, автоматизації процесів та розширення функціональних можливостей. Такі застосунки можуть у реальному часі виконувати аналіз текстів і зображень, генерувати рекомендації, розпізнавати об’єкти чи мову, а також реагувати на користувацькі дії з мінімальними затримками. Проте інтеграція AI-компонентів у вебзастосунки створює значні технічні виклики, пов’язані з продуктивністю, масштабованістю та стабільністю роботи системи.

Однією з головних проблем є високе навантаження на обчислювальні ресурси, адже генерація відповідей моделями машинного навчання потребує великих обсягів пам’яті, потужних процесорів або графічних прискорювачів. У випадку інтеграції зовнішніх AI-сервісів через API або WebSocket-доступ, до цього додаються мережеві затримки, непередбачуваний час відповіді, а також залежність від стабільності зовнішньої інфраструктури. У результаті виникає дилема між якістю інтелектуальної обробки даних та швидкістю реакції системи, особливо у випадках, коли користувач очікує результат у режимі реального часу.

Актуальність теми зумовлена тим, що сучасний ринок веброзробки переживає перехід від традиційних CRUD-застосунків до інтерактивних, адаптивних систем, які використовують моделі штучного інтелекту у своїй основній логіці. Компанії інтегрують сервіси OpenAI, Google Vertex AI, Hugging Face або власні моделі для аналізу запитів користувачів, генерації текстів чи обробки зображень. Проте більшість таких інтеграцій здійснюються без урахування особливостей навантаження та вимог до продуктивності. Відсутність систематичних досліджень у цьому напрямі призводить до непередбачуваних затримок, перевитрати ресурсів і зниження якості користувацького досвіду.

1 АНАЛІЗ ПРЕДМЕТНОЇ ОБЛАСТІ

1.1 Сучасні підходи до побудови вебзастосунків

1.1.1 Архітектурні стилі вебзастосунків

Архітектура вебзастосунку визначає спосіб організації його компонентів, взаємодію між ними та підхід до розгортання й масштабування. Від вибору архітектурного стилю залежить не лише продуктивність системи, але й складність інтеграції зовнішніх сервісів, у тому числі штучного інтелекту. У сучасній практиці веброзробки найпоширенішими є три підходи: монолітна архітектура, мікросервісна архітектура та serverless-підхід. Кожен із них має власні переваги й обмеження щодо інтеграції AI-сервісів у режимі реального часу.

Монолітна архітектура є класичним підходом до побудови вебзастосунків, у якому всі компоненти – інтерфейс користувача, бізнес-логіка, обробка даних і доступ до бази – реалізовані в єдиному програмному модулі. В контексті інтеграції AI-компонентів монолітна архітектура має істотні обмеження. Зокрема, у разі використання зовнішніх моделей або API, усі запити проходять через єдиний застосунок, що може призвести до перевантаження центрального вузла. Масштабування системи вимагає розгортання повних копій усього застосунку, навіть якщо лише модуль роботи з AI є ресурсомістким. Крім того, монолітні системи менш гнучкі в оновленні – додавання нової AI-функціональності може потребувати зміни всієї програми, що ускладнює підтримку та тестування.

Мікросервісна архітектура, на відміну від монолітної, базується на ідеї розділення функціональності системи на незалежні сервіси, кожен з яких виконує окреме завдання: наприклад, авторизацію, аналітику або взаємодію з AI-моделлю. Інтеграція AI у мікросервісне середовище зазвичай здійснюється у вигляді окремого AI-gateway або AI-сервісу, що приймає запити від інших

компонентів системи. Це дає змогу ізолювати обчислювально важку логіку, застосовувати різні технології, а також масштабувати саме AI-компоненту при зростанні навантаження. Основними викликами при цьому є координація між сервісами, підтримання узгодженості даних і контроль часу відповіді, який може зростати через мережеві взаємодії.

Serverless-архітектура є новітнім напрямом, що передбачає виконання логіки застосунку у вигляді окремих функцій, які автоматично запускаються у хмарному середовищі. У контексті AI це відкриває можливість гнучкого розподілу навантаження: роботу з AI-компонентою можна виконувати у функції, що викликається на вимогу, або у спеціальному контейнері з GPU. Проте головним недоліком serverless-підходу є «cold start» – затримка при першому запуску функції, а також обмеження на обсяг пам’яті й тривалість виконання. Для систем, що працюють у режимі реального часу, ці затримки можуть бути критичними. Крім того, складно організувати довготривалі стрімінгові з’єднання з AI-моделлю через обмеження тривалості функції.

1.1.2 Вебзастосунки реального часу

Вебзастосунки реального часу – це системи, здатні миттєво реагувати на події та зміни даних без необхідності ручного оновлення сторінки користувачем. На відміну від традиційних клієнт-серверних застосунків, де кожен запит ініціюється користувачем, вебзастосунки реального часу підтримують постійний двосторонній канал зв’язку між клієнтом і сервером, що дозволяє передавати дані в обох напрямках з мінімальною затримкою. Такий підхід є критично важливим для інтеграції AI-сервісів, які виконують аналіз, прогнозування або генерацію результатів у динамічному середовищі – наприклад, під час обробки голосових запитів, генерації відповідей мовними моделями чи моніторингу поведінки користувачів у реальному часі.

Принцип роботи вебзастосунків реального часу ґрунтується на постійному

оновленні стану системи відповідно до нових подій. Замість моделі «запит-відповідь», притаманної класичним вебсервісам, використовується модель, орієнтована на події, де дані передаються одразу після їх появи. Це дає змогу забезпечити інтерактивність, зменшити затримки та досягти ефекту «живої» взаємодії з користувачем. У поєднанні з AI-модулями така модель дозволяє створювати інтелектуальні сервіси, які реагують у реальному часі – наприклад, чати з мовними моделями, системи розпізнавання обличь або інтерактивні панелі моніторингу.

Основними технологіями, які забезпечують роботу вебзастосунків у реальному часі, є WebSocket, Server-Sent Events (SSE) та WebRTC.

WebSocket забезпечує постійне двостороннє з’єднання між клієнтом і сервером, що дозволяє надсилати дані без необхідності повторного відкриття HTTP-з’єднання. Для інтеграції AI-сервісів цей протокол особливо корисний у сценаріях стрімінгової генерації, наприклад, коли модель поступово формує текстову відповідь або надсилає часткові результати.

SSE є простішою технологією, яка дозволяє серверу надсилати дані клієнту в односторонньому режимі. Вона часто використовується для оновлення даних у реальному часі, наприклад у системах моніторингу або аналітики, коли дані надходять стабільними потоками.

WebRTC орієнтований на обмін мультимедійними потоками (аудіо, відео) та використовується для створення інтерактивних систем відеозв’язку або розпізнавання мови на льоту. У контексті AI це дає змогу поєднувати моделі розпізнавання голосу, перекладу чи емоційного аналізу з реальним користувацьким потоком.

Важливу роль у побудові таких систем відіграють технології асинхронної обробки подій. Більшість сучасних фреймворків підтримують асинхронну модель виконання, що дозволяє обробляти тисячі одночасних підключень без значного зниження продуктивності. Це критично важливо при використанні зовнішніх AI-сервісів, оскільки запити до них часто є блокуючими через тривалість обчислень. Асинхронна архітектура дозволяє не блокувати основний

потік виконання, очікуючи на відповідь від моделі, а паралельно обробляти інші запити.

Для масштабування вебзастосунків реального часу використовуються брокери повідомлень (наприклад, RabbitMQ, Kafka, NATS) або pub/sub-системи, які дозволяють розподіляти події між кількома серверами чи мікросервісами. Такий підхід важливий при інтеграції AI-сервісів, коли різні компоненти системи виконують окремі частини обчислень – наприклад, один сервіс збирає вхідні дані, інший виконує обробку за допомогою AI, а третій формує кінцеву відповідь.

Не менш важливим аспектом є оптимізація затримки. Для систем, які працюють у режимі реального часу, навіть невелика затримка у 200 –300 мс може бути помітною користувачу. Основними джерелами такої затримки є передусім час, необхідний на безпосереднє виконання AI-запиту моделлю, а також тривалість передачі даних мережею. Крім того, на швидкість реакції суттєво впливають процеси серіалізації та десеріалізації великих об’єктів і конкуренція ресурсів, що виникає при обробці великої кількості одночасних з’єднань.

Для зменшення затримок використовуються техніки кешування відповідей, розподілення навантаження через балансувальники, а також локальне розміщення часто використовуваних моделей на периферійних серверах.

1.2 Інтеграція AI-сервісів у вебзастосунки

1.2.1 Типи AI-сервісів

У сучасній інженерній практиці під AI-сервісом розуміють програмний компонент або хмарне API, що виконує завдання машинного навчання або глибинного аналізу даних і повертає результат у зручному форматі – найчастіше через REST-інтерфейс, gRPC або WebSocket-потік. Тип AI-сервісу визначає особливості його інтеграції, обчислювальні вимоги та вплив на продуктивність вебзастосунку.

Найбільш стрімкий розвиток останніх років спостерігається у сфері великих мовних моделей, таких як GPT, Claude, Gemini, тощо. Ці системи здатні аналізувати, узагальнювати й генерувати тексти природною мовою, виконувати семантичний пошук, класифікацію запитів, а також взаємодіяти з користувачем у діалоговому форматі. У вебзастосунках LLM-сервіси використовуються для побудови чат-ботів, систем підтримки користувачів, генераторів контенту, інструментів перекладу або резюмування документів.

Особливість LLM-моделей полягає у високій складності обчислень під час етапу отримання відповіді від моделі. Один запит може вимагати десятків гігабайт пам’яті й сотень мілісекунд або навіть секунд часу, залежно від довжини контексту. Для вебзастосунків реального часу це створює ряд додаткових проблем. По-перше, затримка відповіді безпосередньо впливає на користувацький досвід, роблячи інтерфейс менш чуйним. По-друге, значна варіативність часу відповіді унеможливлює точне прогнозування навантаження на систему. Зрештою, асинхронність отримання результату вимагає обов’язкової підтримки потокової передачі даних через протоколи WebSocket або SSE для забезпечення плавної взаємодії.

У результаті LLM-моделі зазвичай інтегруються як зовнішні сервіси (OpenAI API, Hugging Face Inference API, Anthropic API), а не як локальні компоненти системи. Це спрощує впровадження, але створює залежність від мережевих затримок і тарифікації використання.

Другий ключовий клас AI-сервісів охоплює моделі комп’ютерного зору, призначені для аналізу зображень і відео. Вони виконують розпізнавання об’єктів, класифікацію сцен, виявлення руху, сегментацію зображень і відстеження об’єктів у часі. Такі сервіси широко застосовуються у вебзастосунках для вирішення різноманітних завдань, зокрема для контролю безпеки та доступу через розпізнавання облич, а також у роботі інтелектуальних систем відеоспостереження. Окрім цього, комп'ютерний зір використовується для візуального пошуку товарів, високоточної обробки медичних зображень та автоматичної модерації контенту на платформах.

У контексті вебінтеграцій комп’ютерний зір має специфічні технічні вимоги. Передача зображень або відеопотоків через мережу створює значне навантаження на канал зв’язку, тому потрібна попередня обробка на клієнтській стороні – зменшення роздільної здатності, конвертація у base64 або формати з компресією без втрат. Додатковим викликом є обчислювальна інтенсивність: обробка відеопотоку з частотою 30 кадрів за секунду вимагає значних GPU-ресурсів або спеціальних edge-сервісів. Тому у багатьох випадках для комп’ютерного зору використовується гібридний підхід – попередня обробка даних на клієнті або edge-сервері та передача результатів, а не всього відео, до центрального AI-сервісу.

Для реалізації таких систем застосовуються як хмарні рішення (Google Vision API, AWS Rekognition, Azure Computer Vision), так і open-source-фреймворки (OpenCV, YOLO, MediaPipe, Detectron2). Вибір залежить від вимог до продуктивності, приватності даних і складності інтеграції.

Третю групу становлять аналітичні та прогностичні AI-сервіси, орієнтовані на аналіз великих масивів даних, виявлення закономірностей і прогнозування майбутніх подій. Такі сервіси базуються на класичних методах машинного навчання, як-от регресія, дерева рішень, кластеризація та нейронні мережі. Їх практичне застосування охоплює прогнозування продажів, попиту чи потенційних ризиків, а також виявлення аномалій у потоках даних у реальному часі. Крім того, ці моделі є незамінними для впровадження динамічного ціноутворення та комплексної оптимізації складних бізнес-процесів.

На відміну від LLM-моделей, аналітичні сервіси часто виконуються локально на сервері вебзастосунку або у вигляді мікросервісу, що приймає дані у форматі JSON, обробляє їх і повертає аналітичний результат. У системах реального часу ці сервіси часто працюють у зв’язці з брокерами повідомлень (Kafka, RabbitMQ) або потоковими системами (Apache Flink, Spark Streaming), що дозволяє виконувати потокову обробку – безперервну обробку даних у момент їх надходження. Виклик інтеграції таких систем полягає у забезпеченні стабільної пропускної здатності і мінімальної затримки при високому обсязі

подій, а також у правильній балансировці між швидкістю обчислень і точністю прогнозу.

Окремий тп надзвичайно важливий тип AI-сервісів – це рекомендаційні системи, які аналізують поведінку користувачів, їхні уподобання й історію дій для персоналізації контенту. Рекомендаційні модулі є ключовим елементом сучасних платформ – від соціальних мереж і відеосервісів до онлайн-магазинів і стрімінгових систем.

З технічного погляду, такі сервіси забезпечують свою ефективність завдяки використанню комбінації методів: контентно-орієнтованого аналізу, що спирається на властивості об’єктів, та колаборативної фільтрації, яка базується на схожості поведінки різних користувачів. Найсучасніші системи впроваджують гібридні підходи, що майстерно поєднують обидва згадані методи із залученням алгоритмів глибинного навчання для досягнення максимальної точності прогнозів.

В умовах роботи у реальному часі головною задачею таких систем є швидке оновлення профілю користувача та генерація рекомендацій при мінімальній затримці. Для цього застосовуються механізми кешування результатів і потокова синхронізація між сервісами.

З точки зору інтеграції у вебзастосунок, рекомендаційні сервіси можуть бути як зовнішніми API, так і власними моделями, розгорнутими локально. У другому випадку виникає потреба у постійному оновленні даних користувачів і масштабуванні при пікових навантаженнях, що прямо впливає на продуктивність системи.

1.2.2 Механізми інтеграції

У сучасній практиці розробки застосовуються чотири основні підходи до інтеграції: REST API, WebSocket та gRPC. Кожен із них має власну сферу ефективності та обмеження, які необхідно враховувати при роботі з AI-

сервісами.

REST залишається найпоширенішим способом інтеграції вебзастосунків з AI-сервісами. Його головними перевагами є простота, універсальність і сумісність – практично всі сучасні AI-провайдери підтримують REST-інтерфейси. Комунікація здійснюється через стандартні HTTP-запити (переважно POST або GET) з передачою даних у форматі JSON.

Проте REST має низку обмежень у контексті реального часу. Кожен запит відкриває нове HTTP-з'єднання, що збільшує накладні витрати при великій кількості викликів. Крім того, REST реалізує синхронну модель взаємодії: клієнт очікує повної відповіді перед обробкою результату. Це не завжди прийнятно при роботі з великими мовними моделями або сервісами генерації, які можуть формувати результат поступово.

WebSocket забезпечує постійне двостороннє з'єднання між клієнтом і сервером, що дозволяє обмінюватися даними без повторного відкриття з'єднання. Це робить його ідеальним механізмом для систем реального часу – зокрема, чатів, стрімінгової генерації відповідей від AI-моделей, інтерактивних інтерфейсів аналітики або візуалізації результатів обробки моделі.

Перевагою WebSocket є можливість стрімінг-взаємодії, коли AI-сервіс може надсилати часткові результати у вигляді подій або токенів. Клієнт отримує дані поступово, що суттєво зменшує суб'єктивну затримку та покращує користувацьке сприйняття. Крім того, WebSocket дозволяє обробляти кілька паралельних потоків даних в одному з'єднанні, що важливо при інтеграції декількох AI-сервісів одночасно.

Основним викликом при використанні WebSocket є керування станом з'єднань. На відміну від REST, де кожен запит ізольований, WebSocket-з'єднання постійно активне й потребує відстеження підключень, контролю черг повідомлень і запобігання перевантаженню сервера.

При інтеграції з AI-модулями WebSocket часто виступає посередником між користувачем і зовнішнім AI-API, надаючи більш гнучкий контроль над потоком даних і дозволяючи використовувати реактивну архітектуру.

gRPC – це сучасний фреймворк від Google, побудований поверх протоколу HTTP/2, який забезпечує високу швидкодію, двосторонній стрімінг і ефективну серіалізацію даних за допомогою формату Protocol Buffers. У контексті AI інтеграцій gRPC часто використовується у мікросервісних архітектурах або всередині кластерів, де важливі низькі затримки та передача великих обсягів структурованих даних.

Порівняно з REST, gRPC має менші накладні витрати на транспорт, оскільки використовує бінарний формат і зберігає з'єднання відкритим. Це дає значне підвищення продуктивності, особливо при інтенсивних викликах між AI-модулями. Двосторонній стрімінг дозволяє реалізовувати обробку даних у реальному часі, коли клієнт і сервер одночасно надсилають і приймають дані – наприклад, при обробці відеопотоку або безперервному генеруванні AI-відповіді для великої кількості запитів.

Проте впровадження gRPC у публічні вебзастосунки має певні обмеження. Через використання нестандартного протоколу для браузерів його важче реалізувати без проміжного REST-проксі або WebSocket-шлюзу. Тому найчастіше gRPC використовується у внутрішніх системах, що взаємодіють між собою у межах мікросервісної інфраструктури або між AI-компонентами одного застосунку.

В таблиці 1.1 наведено порівняльну характеристику розглянутих механізмів інтеграції.

Таблиця 1.1 – Порівняльна характеристика механізмів інтеграції

Механізм	**Тип зв'язку**	**Підтримка стрімінгу**	**Затримка**	**Складність впровадження**	**Типові сценарії**
REST API	Одноразовий запит- відповідь	Ні	Середня	Низька	Простий доступ до AI API, CRUD-операції
WebSocket	Постійне двостороннє	Так	Низька	Середня	Реальний час, чат-боти, стрімінг результатів
gRPC	Двосторонній потоковий	Так (full duplex)	Дуже низька	Висока	Мікросервіси, внутрішні AI-вузли

1.3 Продуктивність вебзастосунків з AI-компонентами

Продуктивність вебзастосунків, що інтегрують компоненти штучного інтелекту, є одним із ключових показників їх ефективності. У випадку систем, які працюють у режимі реального часу, від швидкодії безпосередньо залежить якість користувацького досвіду, стабільність роботи та економічна доцільність використання інфраструктури. На відміну від традиційних вебзастосунків, де затримка у декілька сотень мілісекунд є прийнятною, інтеграція AI-компонентів створює додаткові виклики через складність обчислень, потребу в паралельній обробці даних та залежність від зовнішніх сервісів.

Далі буде наведено основні метрики продуктивності вебзастосунків.

Швидкодія – це середній час між моментом надходження запиту від користувача і моментом отримання повної відповіді від системи. Цей показник є основною метрикою продуктивності для вебзастосунків реального часу, особливо при взаємодії з AI-сервісами.

Пропускна здатність – характеризує кількість запитів, які система здатна обробити за одиницю часу. У системах із AI-компонентами пропускна здатність тісно пов’язана з доступними обчислювальними ресурсами – зокрема, кількістю GPU або CPU-ядер, паралельністю запитів до моделі та ефективністю черги запитів.

При інтеграції AI через зовнішні API пропускна здатність часто обмежується політиками тарифікації або технічними лімітами постачальника. Тому при дослідженні продуктивності важливо оцінювати середню кількість успішно виконаних запитів і відсоток відхилених через перевищення ліміту.

Масштабованість – це здатність системи підтримувати зростання навантаження без істотного погіршення продуктивності. Для AI-застосунків цей параметр має особливе значення, оскільки отримання результату їх роботи є ресурсомістким процесом, що вимагає значних обчислювальних потужностей.

У технічній практиці масштабування може бути реалізоване двома основними шляхами. По-перше, через вертикальне масштабування, яке

передбачає безпосереднє збільшення ресурсів окремого вузла, як-от нарощування потужності CPU чи GPU та обсягу оперативної пам'яті (RAM). По-друге, через горизонтальне масштабування, що полягає в додаванні нових робочих вузлів до системи для рівномірного розподілу вхідних запитів між ними.

При інтеграції AI-сервісів доцільно застосовувати горизонтальне масштабування, коли кожна сутність моделі обробляє частину запитів. Це дозволяє підтримувати стабільну швидкодію навіть при різкому зростанні трафіку.

Надійність системи під час роботи з AI-сервісами визначається її здатністю стабільно обробляти запити навіть в умовах критичних навантажень та мережевих затримок. Основними показниками, що характеризують цей стан, є Error Rate, який відображає відсоток запитів, що завершилися з помилкою, та Timeout Rate, що вказує на частку запитів, час відповіді на які перевищив встановлені допустимі межі. Також ключове значення має показник Uptime, який демонструє загальний відсоток часу, протягом якого система перебуває в повноцінному робочому стані та доступна для користувачів.

Для вебзастосунків з AI-компонентами допустимий Error Rate не має перевищувати 1–2%, тоді як середній час простою повинен бути меншим за 0,1% від загального часу роботи. Висока надійність досягається за рахунок резервування серверів, ретраїв запитів і децентралізованої архітектури.

Хоча продуктивність часто оцінюють технічними метриками, у системах з AI значну роль відіграє якість сприйняття користувачем. Навіть якщо швидкодія незначно перевищує норму, але користувач бачить поступовий потік результату (наприклад, токенів у чаті), загальне враження залишається позитивним.

1.4 Аналіз існуючих рішень і технологічних стеків

Сучасні технологічні стеки для реалізації вебзастосунків із компонентами штучного інтелекту можна умовно поділити на два ключові рівні. Перший рівень

охоплює платформи AI-сервісів, які безпосередньо виконують інтелектуальну обробку даних, забезпечуючи роботу моделей. Другий рівень складають системи інфраструктури та моніторингу, чиїм завданням є забезпечення стабільної роботи всієї архітектури та оцінювання продуктивності системи в режимі реального часу.

На рівні реалізації AI-функціональності сьогодні переважають хмарні сервіси, які надають готові API для інтеграції моделей у будь-який вебзастосунок. Найпоширенішим рішенням є OpenAI API – універсальний сервіс для роботи з великими мовними моделями, що підтримує текстову, мультимодальну та стрімінгову генерацію відповідей. Його головною перевагою є висока якість результатів і стабільна швидкодія при значному навантаженні, проте він має і недоліки, як-от залежність від зовнішньої інфраструктури та мережевої затримки.

Альтернативою є Google Vertex AI – платформа, орієнтована на побудову повного циклу машинного навчання, включаючи тренування, розгортання, моніторинг і масштабування моделей. Вона дозволяє розробникам комбінувати власні моделі з готовими API для обробки тексту, зображень і мови. Своєю чергою, Hugging Face Inference API представляє екосистему відкритих моделей, що дає можливість швидко інтегрувати NLP, CV та мультимодальні рішення через REST або gRPC, вирізняючись широким вибором моделей і прозорою інтеграцією через SDK.

Замикають цей ряд комплексні платформи корпоративного рівня – AWS SageMaker та Azure AI Services. Вони надають потужні інструменти для автоматизованого розгортання моделей, їхньої оркестрації та безшовного масштабування безпосередньо у хмарі.

Для випадків, коли AI-компоненти необхідно розгорнути локально, використовуються спеціалізовані фреймворки TorchServe, TensorFlow Serving або NVIDIA Triton Inference Server, які дозволяють налаштовувати середовище виконання моделей на GPU або CPU. Такі рішення доцільні у випадках, коли система потребує низької затримки та мінімальної залежності від зовнішніх

сервісів, але вимагають складнішої підтримки інфраструктури.

1.5 Постановка задачі дослідження

Об’єкт дослідження – процес функціонування вебзастосунків, що взаємодіють із сервісами штучного інтелекту у режимі реального часу.

Предмет дослідження – закономірності зміни продуктивності вебзастосунків залежно від типу інтеграції AI-сервісів, архітектури системи та способу обміну даними.

Мета дослідження – виявити та формалізувати фактори, які впливають на швидкодію вебзастосунків при інтеграції AI-компонентів, а також розробити рекомендації щодо підвищення ефективності їх роботи в умовах реального часу.

Для досягнення поставленої мети визначено наступні основні завдання дослідження: на першому етапі передбачається проаналізувати сучасні підходи до побудови вебзастосунків та існуючі методи інтеграції AI-сервісів, наступним кроком є визначення основних метрик, за якими здійснюється оцінка продуктивності систем із AI-компонентами.

Важливою частиною роботи є побудова математичної моделі оцінювання продуктивності, яка б враховувала затримки, пропускну здатність та параметри масштабованості. Далі планується провести експериментальне дослідження для виявлення впливу різних архітектурних і комунікаційних рішень на час загальної відповіді системи. Підсумковим завданням є формулювання практичних рекомендацій щодо вибору технологічного стеку та стратегій інтеграції для досягнення максимально стабільної роботи застосунків у режимі реального часу.

Наукова новизна роботи полягає у комплексному підході до дослідження впливу інтеграції AI-сервісів на продуктивність вебзастосунків у режимі реального часу. У роботі узагальнено теоретичні принципи та практичні методи оцінювання швидкодії систем із зовнішніми інтелектуальними компонентами, а також запропоновано модель аналізу продуктивності, що враховує мережеві

затримки, особливості обчислень на стороні AI-сервісу та архітектурні параметри застосунку.

Новизна також полягає у формалізації взаємозв’язку між типом інтеграції і ключовими показниками ефективності, такими як швидкодія, пропускна здатність та масштабованість.

Практична значущість дослідження полягає у можливості використання отриманих результатів для проектування, оптимізації та моніторингу вебзастосунків, що працюють з AI-компонентами. Запропоновані підходи дозволяють зменшити затримку обробки запитів, підвищити стабільність системи при високих навантаженнях та ефективно розподіляти обчислювальні ресурси.

Результати можуть бути використані при розробці прототипів, корпоративних рішень або навчальних проєктів, де необхідна інтеграція інтелектуальних сервісів із вимогами до роботи у реальному часі.

2 МОДЕЛЮВАННЯ ПРОДУКТИВНОСТІ ВЕБЗАСТОСУНКІВ ПРИ ІНТЕГРАЦІЇ AI-СЕРВІСІВ

2.1 Теоретичні основи оцінювання продуктивності

2.1.1 Моделі часу відгуку систем із зовнішніми API

Одним із ключових аспектів оцінювання продуктивності вебзастосунків, що взаємодіють з AI-сервісами, є час відгуку системи. Цей показник визначає, наскільки швидко користувач отримує результат після надсилання запиту.

У випадку інтеграції із зовнішнім API загальний час відгуку формується як сукупність кількох критичних складових. Він включає час, витрачений на передачу запиту мережею до сервера, а також тривалість обробки запиту безпосередньо всередині вебзастосунку. Найвагомішою частиною часто є час виконання обчислень моделлю штучного інтелекту, до якого на завершальному етапі додається час передачі сформованої відповіді назад клієнту.

Загальний час відгуку можна представити у вигляді суми компонентів:

$$T_{response} = T_{network} + T_{server} + T_{model} + T_{return}, \quad (2.1)$$

де $T_{network}$ – середній час передачі запиту мережею;

T_{server} – час обробки запиту на сервері (включно з валідацією, маршрутизацією, логуванням, тощо);

T_{model} – час, який витрачається на обробку даних моделлю (AI-компонентою);

T_{return} – час зворотної передачі результату користувачу.

Для спрощення аналізу можна об’єднати перші та останні компоненти ($T_{network}$ і T_{return}) у загальний показник мережевої затримки (Round-Trip Time):

$$T_{RTT} = T_{network} + T_{return}. \quad (2.2)$$

Тоді загальний час відгуку системи можна подати як:

$$T_{response} = T_{RTT} + T_{server} + T_{model}. \quad (2.3)$$

Оскільки більшість інтеграцій із AI-сервісами відбуваються через HTTP або WebSocket, значення T_{RTT} визначається затримками маршрутизації та пропускною здатністю каналу. Для типових вебзастосунків реального часу цей параметр коливається у межах 50–200 мс, але при використанні зовнішніх API може сягати 500–700 мс, що вже істотно впливає на загальне сприйняття швидкодії.

Оскільки час обробки запитів є випадковою величиною, що залежить від навантаження системи, розмірів вхідних даних та характеристик моделі, доцільно розглядати $T_{response}$ як випадкову величину з певним розподілом імовірностей $f(t)$.

У найпростішому випадку (за умов експоненційного розподілу) середній час відгуку можна подати як:

$$E[T_{response}] = \frac{1}{\mu - \lambda}, \quad (2.4)$$

де λ – інтенсивність надходження запитів (запитів/сек);

μ – середня швидкість обробки запитів (запитів/сек), за умови $\lambda < \mu$.

Якщо запит потребує обробки кількома сервісами (наприклад, вебсервер – AI API – база даних), загальний час відгуку можна розглядати як суму середніх часів кожного етапу:

$$E[T_{response}] = \sum_{i=1}^{n} \frac{1}{\mu_i - \lambda_i}, \quad (2.5)$$

де n – кількість послідовно з’єднаних компонент системи.

Для реальних систем важливим є не лише середній час відгуку, а й його

варіативність (розкид значень). Для її оцінки застосовують коефіцієнт варіації:

$$C_v = \frac{\sigma_T}{E[T_{response}]}, \tag{2.6}$$

де σ_T – стандартне відхилення часу відгуку.

Чим більший C_v, тим менш стабільна система: час обробки запитів може різко коливатися залежно від стану AI-сервісу, мережі чи черги завдань.

У таблиці 2.1 наведено приклад типових значень складових часу відгуку для різних архітектурних сценаріїв інтеграції.

Таблиця 2.1 – Орієнтовна структура часу відгуку вебзастосунку з AI-компонентою

Компонент	**REST API (зовнішній сервіс)**	**WebSocket (стрімінг)**	**Локальна модель**
Мережеві затримки T_{RTT} мс	300 – 600	100 – 250	20 – 50
Обробка сервером T_{server} мс	50 – 150	50 – 100	50 – 100
Обробка моделлю T_{model} мс	500 – 1200	400 – 800	100 – 400
Сумарно $T_{response}$ мс	850 – 1950	550 – 1150	170 – 550

2.1.2 Моделювання паралельних запитів і мережевих затримок

У реальних умовах роботи вебзастосунку більшість запитів до AI-сервісів надходять паралельно, особливо якщо система обслуговує багато користувачів або виконує кілька інтелектуальних завдань одночасно. Тому при оцінюванні продуктивності важливо враховувати не лише середній час одного запиту, а й поведінку системи під навантаженням – тобто залежність загального часу

обробки від кількості одночасних клієнтів і пропускної здатності мережі.

Нехай n – кількість паралельних запитів, що одночасно надходять до системи, а p – кількість потоків або серверів, які можуть обробляти ці запити незалежно. Тоді ефективний час обробки одного запиту в умовах паралельного навантаження можна оцінити як:

$$T_{effective} = T_{response} * \left(1 + \alpha * \frac{n}{p}\right), \quad (2.7)$$

де $T_{response}$ – середній час відгуку для одного запиту;

α – коефіцієнт взаємного впливу ($0 \leq \alpha \leq 1$), який враховує конкуренцію за ресурси процесора, пам'яті та мережі.

Якщо $\alpha = 0$, запити обробляються незалежно (ідеальний паралелізм). При $\alpha \to 1$ зростання кількості одночасних запитів лінійно збільшує затримку через перевантаження системи.

Мережевий час відгуку залежить від затримки в каналі (L) і доступної пропускної здатності (B). Для передачі пакету обсягом D байт середня затримка може бути подана як:

$$T_{RTT} = L + \frac{D}{B}. \quad (2.8)$$

У випадку інтеграції із зовнішніми AI-сервісами затримка L часто домінує, оскільки дані передаються через глобальні мережі. Наприклад, при $L = 150$ мс і $B = 10$ МБ/с передача JSON-запиту розміром 1 МБ дає:

$$T_{RTT} = 0{,}15 + \frac{1}{10} = 0{,}25 \text{ с}. \quad (2.9)$$

Тобто, навіть без обробки на стороні моделі загальний мінімальний час відповіді становитиме близько 250 мс. Для систем реального часу це критично, тому важливими стають оптимізація форматів даних, стиснення та локалізація

серверів.

Для спрощеної оцінки можна ввести коефіцієнт конкурентного завантаження:

$$K = \frac{n}{p_{effective}}, \tag{2.10}$$

де $p_{effective}$ – ефективна кількість потоків або серверів, здатних обробляти запити без втрати продуктивності.

При $K > 1$ система перевантажується, і середній час відповіді зростає експоненційно:

$$T_{server}(K) = T_{server}(1) * e^{\beta(K-1)}, \tag{2.11}$$

де β – емпіричний коефіцієнт чутливості до навантаження (зазвичай у межах 0,3–0,6).

2.1.3 Коефіцієнт ефективності інтеграції AI-компоненти

Для кількісного оцінювання впливу інтеграції AI-компоненти на загальну продуктивність вебзастосунку доцільно ввести коефіцієнт ефективності інтеграції $K_{effective}$, який характеризує, наскільки система зберігає свою швидкодію за умов взаємодії з інтелектуальним сервісом у реальному часі.

Базове визначення:

$$K_{effective} = \frac{P_{AI}}{P_{base}} = \frac{\frac{n}{T_{response,AI}}}{\frac{n}{T_{response,base}}} = \frac{T_{response,base}}{T_{response,AI}}, \tag{2.12}$$

де P_{AI} – продуктивність системи з інтегрованим AI-сервісом (кількість

оброблених запитів за одиницю часу);

P_{base} – продуктивність базової системи без AI-компоненти;

$T_{response,AI}$ – середній час відгуку системи з AI;

$T_{response,base}$ – середній час відгуку без AI.

Відповідно, $K_{effective} < 1$ свідчить про погіршення швидкодії після інтеграції AI-компоненти, а $K_{effective} \to 1$ – про ефективну, оптимізовану інтеграцію з мінімальними втратами продуктивності.

Для більш комплексного урахування факторів навантаження та затримок формулу можна узагальнити:

$$K_{effective} = \frac{T_{response,base}}{T_{server}+T_{model}+T_{RTT}(1+\frac{n}{p}\alpha)}, \quad (2.13)$$

де параметри T_{server}, T_{model}, T_{RTT}, n, p, α визначено раніше.

Таке подання дозволяє аналітично оцінити, як змінюється ефективність інтеграції при збільшенні кількості користувачів, зміні мережевих умов або складності моделі.

2.2 Моделювання впливу AI-обчислень на продуктивність.

2.2.1 Визначення параметрів навантаження

Для оцінювання впливу AI-обчислень на продуктивність вебзастосунку необхідно формалізувати основні параметри навантаження, які визначають поведінку системи при роботі в реальному часі. Ключове значення тут має кількість одночасних користувачів, що створюють паралельні запити до системи, а також розмір самого запиту, який зазвичай вимірюється у байтах або токенах. Крім того, критичним параметром є тип AI-моделі, оскільки саме він визначає складність необхідних обчислень і, як наслідок, підсумковий час виконання завдання.

Потік користувачів можна описати через інтенсивність запитів λ (запитів за секунду). Для системи, що підтримує одночасно n користувачів із середнім часом відповіді $T_{response}$, виконується співвідношення:

$$n = \lambda * T_{\text{відг}}. \tag{2.14}$$

Ця залежність дозволяє оцінити реальну пропускну здатність системи при фіксованій затримці відповіді.

Розмір даних, що передаються на обробку AI-сервісу, безпосередньо впливає на затримку передачі та тривалість обчислення.

Загальний обсяг переданих даних при роботі n користувачів за інтервал часу t:

$$V = n * D * \frac{t}{T_{response}}. \tag{2.15}$$

У реальних умовах параметр D залежить від типу запиту: короткий текст (≈ 1–2 кБ), зображення (≈ 100–500 кБ), або складна мультимодальна взаємодія (> 1 МБ). При цьому збільшення D пропорційно підвищує час мережевої передачі та обробки.

Тип AI-компоненти визначає характер навантаження на обчислювальні ресурси. Для спрощення можна ввести коефіцієнт складності моделі γ, який виражає відношення часу її виконання до часу базового запиту без AI:

$$T_{model} = \gamma * T_{server}. \tag{2.16}$$

Типові значення γ коливаються від 1,5–2 для простих мовних моделей до > 5 для мультимодальних або генеративних моделей з великими параметрами.

Сумарне навантаження системи за одиницю часу можна подати як:

$$L = n * D * \gamma. \quad (2.17)$$

Цей параметр використовується для оцінки рівня завантаженості системи й визначення порогових умов, за яких продуктивність починає погіршуватися. У подальших підпунктах розглядатиметься, як співвідносяться ці параметри із загальним часом відповіді та які стратегії обробки дозволяють підтримувати стабільну роботу системи при зростанні L.

2.2.2 Співвідношення часу AI-запиту до часу відповіді

Для оцінювання впливу AI-компоненти на загальну швидкодію системи необхідно визначити частку часу, яку займає етап обчислення запиту моделлю відносно повного циклу відповіді.

Визначимо коефіцієнт впливу AI-обчислень на час відповіді:

$$k_{AI} = \frac{T_{model}}{T_{response}}. \quad (2.18)$$

Цей коефіцієнт показує, яку частину загального часу займає виконання моделі.

Типові значення k_{AI} становлять приблизно $0{,}3 - 0{,}4$ для легких мовних моделей, та $0{,}6 - 0{,}8$ для генеративних або мультимодальних систем і $k_{AI} \to 1$ для повністю AI-залежних сервісів.

Якщо припустити, що мережеві затримки й серверна обробка становлять постійну частину c від загального часу ($c = T_{RTT} + T_{server}$), тоді можна виразити відносне зростання повного часу після додавання AI-компоненти:

$$\frac{T_{response}}{T_{response,base}} = 1 + \frac{T_{model}}{c}. \quad (2.19)$$

Тоді при фіксованій інфраструктурі продуктивність системи знижується пропорційно збільшенню частки AI-обчислень у загальному циклі.

2.2.3 Моделювання різних стратегій обробки

Поведінка системи при інтеграції AI-компоненти значною мірою залежить від обраної стратегії обробки запитів. До основних підходів, що застосовуються у сучасній розробці, належать асинхронна обробка, яка дозволяє системі не блокуватись під час очікування результату, пакетна обробка для групування запитів з метою оптимізації ресурсів, а також стрімінгова обробка, що забезпечує передачу даних частинами в міру їх генерації. Кожна з цих стратегій має різний вплив на середній час відповіді та рівень навантаження на інфраструктуру, що вимагає ретельного вибору залежно від вимог до конкретного застосунку.

Асинхронна стратегія передбачає, що запити не блокують основний потік, а виконуються паралельно. Ефективний час відповіді у цьому випадку можна подати як:

$$T_{effective} = \frac{T_{response}}{1+\delta n}, \tag{2.20}$$

де n – кількість одночасних запитів;

δ – коефіцієнт ефективності асинхронності ($0 \leq \delta \leq 1$).

При малій кількості запитів вплив мінімальний, але при зростанні навантаження асинхронна модель значно зменшує затримки, розподіляючи обробку між потоками.

Пакетна стратегія виконує кілька запитів одночасно як один великий блок. Загальний час виконання пакету обсягом b запитів можна оцінити як:

$$T_{package} = T_{init} + \frac{b*T_{model}}{\eta_b}, \tag{2.21}$$

де T_{init} – час ініціалізації пакету;

η_b – коефіцієнт ефективності обробки (залежить від оптимізації обчислень).

Зазвичай $\eta_b > 1$, тому пакетна стратегія ефективна для однотипних запитів, але не підходить для режиму реального часу через необхідність накопичення даних перед виконанням.

У стрімінговій моделі результат надсилається користувачу поступово, у міру його формування. Цей підхід мінімізує сприйняту затримку. Середній час до появи першої частини відповіді:

$$T_{stream} = T_{init} + \frac{T_{model}}{q}, \tag{2.22}$$

де q – коефіцієнт розбиття результату (кількість часткових блоків, що передаються в потоці).

Фактичний час повної відповіді не скорочується, але користувач сприймає взаємодію як значно швидшу.

У таблиці 2.2 наведено порівняльну характеристику різних стратегій обробки запитів.

Таблиця 2.2 – Порівняльна характеристика стратегій обробки запитів

Стратегія	**Середній час відповіді**	**Придатність до реального часу**	**Висновок**
Асинхронна	Зменшується при рості (n)	Висока	Оптимальна для API-запитів
Пакетна	Залежить від (b), може бути більшою	Низька	Ефективна при масовій обробці
Стрімінгова	Суб’єктивно коротша	Дуже висока	Підходить для інтерактивних AI-застосунків

2.3 Оцінювання ефективності архітектурних підходів

2.3.1 Порівняльний аналіз моделей інтеграції

Розподіл місця виконання AI-обчислень безпосередньо визначає швидкодію та ефективність вебзастосунку, що функціонує у режимі реального часу. Сучасні підходи до інтеграції можна звести до трьох базових архітектурних моделей: локальне генерування відповіді**,** gateway-AI та edge-AI. Вони відрізняються розташуванням обчислювальних ресурсів, структурою мережевих взаємодій і можливістю масштабування. Математичне моделювання таких систем доцільно проводити через порівняння часу відповіді, пропускної здатності та коефіцієнтів ефективності при зміні параметрів навантаження.

У моделі локальної інтеграції всі обчислення виконуються безпосередньо на сервері вебзастосунку або на окремому апаратному вузлі, який є його частиною. Мережеві затримки при цьому мінімальні, оскільки дані не передаються до зовнішнього середовища.

Загальний час відповіді такої системи можна подати як:

$$T_{local} = T_{server} + T_{model,local}, \tag{2.23}$$

де $T_{model,local}$ – час виконання моделі на локальних обчислювальних ресурсах.

У цьому випадку середній час відповіді залишається стабільним, поки кількість запитів не перевищує межі паралельної обробки p. Після цього спостерігається зростання затримки за майже лінійною залежністю, що характеризує обмежену масштабованість локального підходу:

$$T_{local}(n) = T_{local}(1) * \left(1 + \frac{n}{p}\right). \tag{2.24}$$

Основними перевагами цієї архітектури є низьке значення T_{local} (0,1 −

$0{,}4\ c$) і висока стабільність, а основним недоліком – залежність продуктивності від обчислювальних потужностей одного вузла.

Модель gateway-AI передбачає використання віддаленого шлюзу або зовнішнього API, який приймає, маршрутизує та обробляє запити до AI-компоненти. Вебзастосунок у цьому випадку виступає лише посередником між користувачем і сервісом. Загальний час відповіді системи з gateway-інтеграцією описується рівнянням:

$$T_{gateway} = T_{RTT} + T_{server} + T_{model,gateway}, \tag{2.25}$$

де $T_{model,gateway}$ – час обчислень на стороні віддаленого сервісу.

Оскільки обчислення виконуються поза локальною інфраструктурою, масштабованість такої системи визначається швидкістю зовнішнього API та його здатністю обробляти потік запитів інтенсивністю λ. Якщо позначити середню пропускну здатність шлюзу як $\mu_{gateway}$, то умова стабільності системи задається нерівністю $\lambda < \mu_{gateway}$. У протилежному випадку відбувається накопичення черги запитів, і середній час відповіді зростає експоненційно:

$$E[T_{gateway}] = \frac{1}{\mu_{gateway} - \lambda}. \tag{2.26}$$

Таким чином, gateway-архітектура характеризується високою масштабованістю, проте має суттєві коливання латентності (до 1,5–2 с) через зовнішні мережеві чинники.

Архітектура edge-AI поєднує риси двох попередніх підходів, розміщуючи обчислення на проміжних серверах, географічно близьких до користувача. Це дозволяє суттєво скоротити час передачі даних та зменшити коливання затримки.

У загальному вигляді час відповіді edge-системи можна подати як:

$$T_{edge} = T_{RTT,local} + T_{server} + T_{model,edge}, \tag{2.27}$$

де $T_{RTT,local}$ – локальна мережна затримка (20–80 мс);

$T_{model,edge}$ – час обчислень на периферійному вузлі.

Edge-модель має вищу стабільність та здатність до горизонтального масштабування, оскільки навантаження розподіляється між кількома вузлами. Її середній час відповіді становить 0,2 – 0,7 с, що є компромісом між швидкодією локальної системи та масштабованістю gateway-рішення.

2.3.2 Масштабованість систем при збільшенні навантаження

Масштабованість системи визначається її здатністю зберігати стабільну продуктивність при зростанні кількості одночасних користувачів або інтенсивності запитів до AI-компоненти. Для вебзастосунків, що інтегрують інтелектуальні сервіси, цей показник є критичним, оскільки обчислення на стороні моделі можуть бути ресурсоємними та чутливими до перевантаження.

У загальному випадку залежність часу відповіді від кількості користувачів можна подати як функцію:

$$t_{response}(n) = t_0 \left(1 + \alpha\left(n - n_{optimal}\right)^k\right), \quad (2.28)$$

де t_0 – середній час відповіді при оптимальному навантаженні $n_{optimal}$;

α – коефіцієнт деградації продуктивності;

k – емпіричний показник нелінійності зростання часу відгуку (зазвичай 1,2 – 1,5);

$n_{optimal}$ – оптимальна кількість одночасних користувачів, яку система здатна обробляти без деградації продуктивності.

Для локальної архітектури значення $n_{optimal}$ обмежене кількістю фізичних ядер або потоків CPU/GPU, тому продуктивність зростає лінійно лише до певного рівня, після чого система переходить у режим перевантаження. У такому

стані середній час відповіді зростає експоненційно, а кількість відмовлених запитів перевищує 5 – 7%.

Edge-AI демонструє найкращу масштабованість у сценаріях із географічно розподіленими користувачами. За рахунок паралельного оброблення запитів на периферійних вузлах середній час відповіді зростає повільніше, а коефіцієнт ефективності залишається високим навіть при збільшенні кількості користувачів у 3 – 5 разів. Ключовим обмеженням edge-підходу є необхідність синхронізації між вузлами, особливо при роботі з генеративними моделями, що підтримують стан сесії.

Для практичного оцінювання масштабованості системи використовується коефіцієнт збереження продуктивності:

$$S(n) = \frac{P(n)}{P(1)} = \frac{t_1}{t_n}, \tag{2.29}$$

де $P(n)$ – пропускна здатність при n користувачах;

t_1 та t_n – середній час відповіді при одиночному та багатокористувацькому навантаженні відповідно.

Порівняльна характеристика стратегій обробки запитів представлена в таблиці 2.3.

Таблиця 2.3 – Порівняльна характеристика стратегій обробки запитів

Архітектура	**Δ середнього часу відповіді при ×10 навантаженні**	**Коеф. збереження продуктивності S(n)**	**Висновок**
Локальна	+220 – 300%	0,55 – 0,65	Обмежена апаратними ресурсами
Gateway-AI	+130 – 180%	0,70 – 0,80	Залежить від пропускної здатності API
Edge-AI	+70 – 100%	0,85 – 0,90	Найкраща масштабованість у розподілених середовищах

2.3.3 Формування критеріїв оптимізації та вибору архітектури

Формування критеріїв оптимізації архітектури вебзастосунку з інтегрованими AI-компонентами ґрунтується на поєднанні кількох взаємопов’язаних аспектів – продуктивності, масштабованості, стабільності роботи та якості користувацького досвіду. Основна мета полягає у забезпеченні мінімального часу відповіді системи при одночасному збереженні стійкості до навантаження та ефективному використанні обчислювальних ресурсів.

Передусім доцільно розглядати затримку як головний параметр, що визначає сприйняття швидкодії користувачем. Її оптимізація досягається шляхом зменшення мережевих витрат, використання постійних каналів зв’язку та розміщення вузлів обчислень у безпосередній близькості до користувача. У системах реального часу це дозволяє скоротити середній час відповіді на 30 – 40% порівняно з централізованими рішеннями.

Другим ключовим напрямом оптимізації є ефективність використання обчислювальних ресурсів. Оскільки AI-компоненти мають високу ресурсомісткість, важливим є забезпечення горизонтального масштабування сервісів і динамічного розподілу навантаження між контейнерами. У хмарних середовищах це досягається завдяки механізмам автоматичного масштабування, які адаптивно збільшують або зменшують кількість екземплярів сервісу залежно від поточного трафіку. Такий підхід дозволяє зберігати стабільність навіть за умов різкого зростання кількості запитів, не перевантажуючи GPU або CPU.

Не менш важливим є аспект стійкості до перевантажень. У цьому контексті доцільно застосовувати асинхронні черги повідомлень, кешування результатів та механізми повторних викликів при помилках. Це підвищує надійність системи й запобігає лавиноподібному зростанню часу відповіді при перевищенні порогу пропускної здатності. Для AI-застосунків така стійкість є визначальною, оскільки навіть короткочасні перевантаження можуть призвести до різкого зниження якості сервісу.

Окремим критерієм виступає якість користувацького досвіду, яка

визначається не лише реальним, а й сприйнятим часом відповіді. Використання стрімінгової генерації результатів, поступового оновлення інтерфейсу та адаптивного керування пріоритетами запитів дозволяє зменшити суб'єктивну затримку та створити відчуття безперервної взаємодії із системою.

Для комплексного оцінювання доцільності архітектурних рішень вводиться інтегральний показник ефективності, який враховує швидкодію, стабільність і масштабованість. У загальному вигляді він може бути поданий як зважена функція вигляду

$$E = \omega_1 \frac{1}{t_{response}} + \omega_2 \eta + \omega_3 S(n), \tag{2.30}$$

де $t_{response}$ – середній час відповіді;

η – коефіцієнт ефективності інтеграції;

n – коефіцієнт збереження продуктивності;

ω_i – вагові коефіцієнти, що визначають пріоритети системи.

Порівняльний аналіз отриманих результатів показує, що локальна архітектура є доцільною для малих проєктів із фіксованим навантаженням, де пріоритетом є мінімальна латентність. Gateway-підхід ефективний при використанні зовнішніх API із високим рівнем масштабованості, тоді як edge-архітектура забезпечує оптимальний баланс між швидкодією, стабільністю та гнучкістю розгортання. Саме останній варіант можна вважати найбільш перспективним для систем, що працюють у режимі реального часу та обслуговують значну кількість користувачів.

3 РОЗРОБКА ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ ДЛЯ ЕКСПЕРИМЕНТАЛЬНОГО ДОСЛІДЖЕННЯ ШВИДКОСТІ ГЕНЕРАЦІЇ ВІДПОВІДЕЙ

3.1 Призначення та загальна характеристика розробленої системи

Розроблене програмне забезпечення створене з метою експериментального порівняння швидкості генерації відповідей мовною моделлю Google Gemini за двома різними підходами: через звичайний REST- запит до серверного маршруту Next.js та через виклик хмарної функції Firebase. У межах дослідження важливо не просто отримати коректну відповідь від моделі, але й виміряти час, необхідний на її формування, і порівняти продуктивність обох методів у максимально однакових умовах. Саме для цього застосунок забезпечує можливість багаторазового повторення запитів, накопичення статистики та зручне відображення отриманих результатів.

Застосунок виконує роль інтерфейсу, який дозволяє користувачеві вручну задавати умови експерименту. Користувач обирає тип вхідних даних: введення текстового фрагмента або завантаження PDF-файлу. Далі визначається спосіб взаємодії з моделлю, тобто метод генерації: REST або Firebase Function. Після відправлення запиту система передає дані у відповідний обробник та отримує відповідь від мовної моделі, причому перед кожним викликом точно фіксується час початку та завершення обробки. Результат повертається разом із виміряною тривалістю, що дозволяє поступово формувати вибірку затримок для кожного методу. Ці значення зберігаються у контексті застосунку доти, доки користувач не оновить сторінку або не розпочне нову сесію.

Сутність практичного рішення полягає у тому, що сама мовна модель, промпт, структура запиту, вміст файлу та формат відповіді у двух методів є повністю ідентичними. Відмінність полягає виключно в інфраструктурній частині, де REST-маршрут обробляється безпосередньо API-роутом Next.js, а другий метод проходить через окрему хмарну функцію, розгорнуту у Firebase.

Завдяки цьому експеримент дозволяє оцінити вплив середовища виконання, мережевої інфраструктури та додаткових шарів обробки на загальний час відповіді.

Функціонально система складається з клієнтської частини, що відображає форму для введення даних, статистику швидкості та отриманий результат, і серверної логіки, що готує запит до моделі Gemini, обробляє файли, відправляє їх до Google API та повертає відповідь. Незалежно від методу, застосунок завжди використовує однаковий базовий промпт для нормалізації відповідей і обмежує їхній обсяг приблизно п’ятьмастами символів. Це дозволяє мінімізувати випадкові коливання, пов’язані з довжиною та структурою згенерованих відповідей, і забезпечити більш коректне порівняння часу обробки.

Особливість побудови системи полягає у наявності чітко відокремлених рівнів: інтерфейсу, логіки викликів до API, серверних обробників та окремого шляху обробки у Firebase Functions. Усі обчислення тривалості виконуються безпосередньо в тих точках, де формується запит до моделі, що дозволяє фіксувати реальний час роботи самої генерації без перекручувань, пов’язаних із рендерингом інтерфейсу або затримками у браузері.

Розроблене програмне забезпечення є інтерактивним інструментом дослідження продуктивності, що дозволяє на практиці виміряти поведінку мовної моделі в різних сценаріях виклику, порівняти два інфраструктурні підходи та зробити висновки щодо доцільності використання того чи іншого методу у реальних проектах. Система забезпечує стабільний процес збирання статистики, наочне відображення результатів і повністю відтворювані умови експерименту.

3.2 Вибір технологій, середовища розробки та архітектурних рішень

Під час створення програмного забезпечення для вимірювання швидкості генерації відповідей постало завдання побудувати гнучку, чітко структуровану

та водночас достатньо легку систему, яка дозволяла б як обробляти запити до мовної моделі, так і відстежувати продуктивність різних варіантів серверної інфраструктури. Тому важливою частиною проєктування стала оцінка доступних технологій та вибір оптимального інструментарію.

Основою системи було обрано платформу Next.js, що поєднує в собі можливості React для побудови клієнтського інтерфейсу та власні серверні маршрути для обробки запитів. Завдяки цьому стало можливим розмістити клієнтську й серверну частини в межах одного проєкту без необхідності створювати окремий бекенд. Такий підхід спрощує підтримку, пришвидшує розробку та дозволяє точно контролювати шлях обробки запиту, що є критичним для коректного вимірювання часу генерації відповідей. Використання TypeScript у всіх частинах проєкту забезпечує сувору типізацію, зменшує кількість помилок на етапі розробки та робить логіку обробки запитів більш прозорою.

Фронтендова частина застосунку побудована на React, а для роботи з асинхронними операціями використовується бібліотека TanStack Query. Це рішення дозволяє чітко структурувати запити до API, контролювати стан завантаження, кешування та помилки. Обраний підхід спрощує тестування логіки та дає можливість легко оновлювати статистику тривалості виконання запиту без зайвих перерендерів інтерфейсу.

Введення контексту в застосунок дає змогу зберігати стан у межах всієї сесії, а саме останню отриману відповідь та масиви тривалостей для кожного методу генерації. Подібне рішення дозволяє працювати зі статистикою на боці клієнта без потреби в додатковому сховищі або бекенд-сервісі.

Серверна логіка всередині Next.js реалізується через обробник POST-запиту, який не лише приймає текст або файл, але й визначає обраний користувачем метод генерації відповіді. Цей обробник використовує Google Gemini API для взаємодії з мовною моделлю. Застосунок також передбачає можливість завантаження PDF-файлів до Google API, що є умовою для коректної роботи моделі при обробці документів. Окремим елементом архітектури виступає базовий промпт, збережений у окремому файлі, що гарантує

стабільність отриманих відповідей та їх однакову структуру для різних запитів. Це рішення було важливим з огляду на те, що відтворюваність результатів – одна з ключових вимог експериментальної частини дослідження.

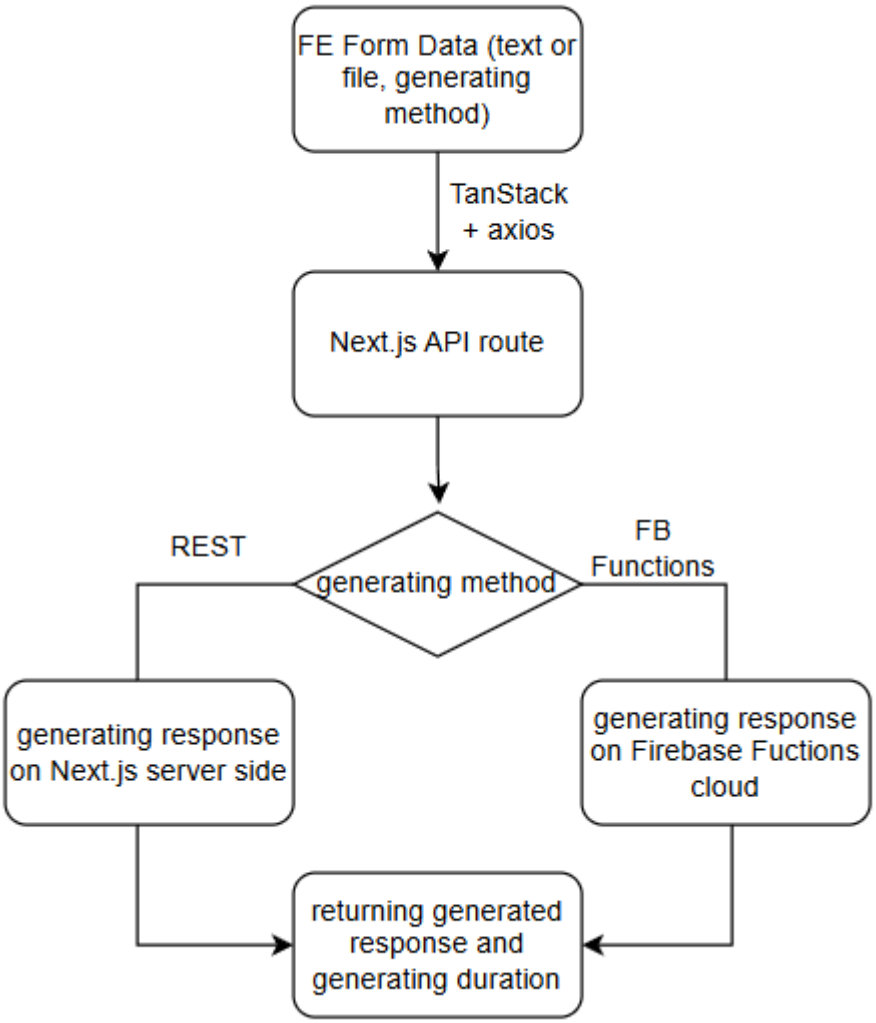

Рисунок 3.1 – Схема архітектури розробленого застосунку

Для реалізації другого методу генерації відповідей застосовано хмарні функції Firebase. Вони дозволяють винести виконання серверної логіки за межі додатку Next.js і порівнювати результат з REST-запитами в умовах іншого середовища виконання. У Firebase Function виконується та сама логіка: розпакування тексту або base64-файлу, завантаження його до Google API, виклик Gemini API та вимірювання часу роботи. Обраний підхід відкриває можливість оцінити, наскільки інфраструктурні відмінності між двома моделями виконання

впливають на сумарний час обробки. Функція деплоїться через конфігураційний файл index.ts, а для обробки HTTP-запитів використовується onRequest, що спрощує інтеграцію з клієнтом.

Застосунок оформлений як односторінковий інтерфейс, у якому користувач може послідовно обрати тип вхідних даних, метод генерації та отримати результат. Для візуалізації згенерованої відповіді використовується ReactMarkdown, що дає змогу правильно відображати форматований текст. Усі проміжні значення часу генеруються на стороні сервера через performance.now(), тому інтерфейс не впливає на точність вимірювань.

3.3 Реалізація клієнтської частини вебзастосунку

Клієнтська частина застосунку виконує роль основної точки взаємодії користувача з експериментальною системою. Вона дає змогу легко вибрати тип вхідних даних, обрати метод генерації відповіді, відправити запит та в режимі реального часу спостерігати за результатами роботи двох різних підходів. Важливо, що логіка фронтенду побудована таким чином, щоб мінімізувати вплив інтерфейсу на точність вимірювань: клієнт лише надсилає дані та відображає отримані значення, не беручи участі у процесі обчислення тривалості.

Основа клієнтської сторони – компонент Home.tsx, що формує структуру головної сторінки та керує логікою вибору параметрів експерименту. У межах цього компонента користувач може переключатися між режимами введення тексту або завантаження файлу. Обраний метод впливає на те, який елемент відображається у формі: або текстове поле для введення інформації, або компонент завантаження файлу. Одразу під цим блоком знаходиться перемикач способу генерації відповіді: REST або Firebase Function. Будь-які варіації в налаштуваннях моментально відображаються в стані застосунку.

Передбачена форма обробляється за допомогою бібліотеки react-hook-form, що дає змогу керувати введенням даних, перевіркою коректності та

передачею значень до функції відправлення запиту. На відміну від традиційних варіантів роботи з формами, ця бібліотека не створює зайвих перерендерів і працює достатньо ефективно, що є важливою умовою під час виконання великої кількості послідовних запитів.

Ключовим елементом взаємодії фронтенду з API є хук useGetAIResponse. Він побудований на основі TanStack Query та інкапсулює логіку виклику API-запиту до маршруту /ai-response. Після успішного отримання відповіді хук оновлює стан у контексті – зберігає згенерований текст та додає час виконання до відповідного масиву залежно від обраного методу. Цей підхід дає змогу відділити мережеву логіку від інтерфейсної та забезпечує стабільну структуру обробки результатів.

Стан застосунку зберігається у GeneratedResponseContext, який містить інформацію про останній отриманий результат, масиви тривалостей для REST- та Firebase-запитів і методи для оновлення цих значень. Контекст використовується на рівні всієї сторінки, тому усі компоненти мають доступ до актуальних даних без потреби передавати їх у властивостях. Він виступає центральним сховищем, що дозволяє провести сесію експериментів без втрати даних під час повторних запитів, доти доки користувач не оновить сторінку.

Після відправлення запиту користувач може побачити індикатор завантаження, а після завершення – блок із результатами. У цьому блоці наводиться середній час виконання всіх запитів певного типу, кількість спроб та остання згенерована відповідь, яка відтворюється у форматі Markdown. Завдяки такому підходу користувач отримує не лише одиничний результат, але й накопичену статистику, що дозволяє більш точно порівняти поведінку двох методів у рамках однієї сесії.

3.4 Серверна логіка в Next.js для генерації відповідей

Серверна частина Next.js у цьому проєкті виконує роль першого з двох

альтернативних механізмів взаємодії з мовною моделлю. Вона обробляє REST-запити, які надходять від клієнтської частини, готує дані у форматі, необхідному для моделі Gemini, виконує завантаження файлів за потреби, формує запит до Google AI та фіксує час генерації відповіді. Структура серверної логіки побудована таким чином, щоб мінімізувати зайву роботу та забезпечити максимально точні вимірювання продуктивності саме на етапі обробки мовною моделлю.

У маршруті POST відбувається приймання даних у вигляді formData, що дозволяє передавати як текст, так і файл. Отримані дані обробляються через стандартні методи Next.js: текстове поле зчитується у вигляді рядка, тоді як файл перетворюється у File-об’єкт, після чого – у ArrayBuffer. Якщо користувач передав файл, серверний код ініціює завантаження цього файлу до Google API, що є обов’язковим кроком перед передачею вмісту до моделі Gemini. Для цього використовується функція uploadFileToGoogleApi, яка формує Blob на основі отриманого буфера, завантажує його та очікує завершення обробки файлу сервісом Google. Тільки після того, як файл переходить у стан, придатний для використання, сервер може включити його до запиту моделі.

Основна частина обробки виконується у момент формування запиту до Gemini. Перед відправкою запиту фіксується час початку обробки, а після отримання відповіді – час завершення. Розрахована різниця і є фактичним часом генерації відповіді, без урахування роботи інтерфейсу, затримок браузера чи інших побічних факторів. Такий спосіб вимірювання дозволяє отримати чисті дані, які відображають лише роботу моделі та інфраструктури, що бере участь у виклику.

У випадку REST-методу логіка виконується всередині Next.js, без додаткових мережевих переходів. Однак, коли користувач обирає метод Firebase, сервер уже не викликає модель напряму – запит передається на зовнішню хмарну функцію, розгорнуту у Firebase. Маршрут у Next.js у такому разі перетворюється фактично на проксі, який перенаправляє дані та передає клієнту результат без додаткової обробки. Це дозволяє порівняти два підходи в однакових умовах,

адже обидва маршрути починаються на клієнті однаково, але далі розходяться у двох різних напрямках.

Загальна структура роута побудована достатньо прозоро, і кожен етап логічно ізольований: приймання даних, обробка файлів, виклик моделі, вимірювання часу, формування відповіді. Завдяки цьому підхід легко розширювати, а також зручно аналізувати під час дослідження. Сервер повертає лише необхідну інформацію – згенерований текст та duration, що робить систему компактною та дозволяє уникнути зайвих впливів на результати.

Лістинг 3.1 Роут /ai-response, який відповідає за серверну частину застосунку:

```
export const POST = async (request: NextRequest) => {
  try {
    const formData = await request.formData();

    const text = (formData.get("text") as string) || undefined;
    const file = (formData.get("file") as File) || undefined;
    const method = formData.get("method") as GeneratingMethods;

    const buffer = file ? await file.arrayBuffer() : undefined;

    switch (method) {
      case GeneratingMethods.REST: {
        const uploadedFile = buffer
          ? await uploadFileToGoogleApi(buffer)
          : undefined;

        const start = performance.now();
        const response = await getGeminiResponse(
          BASIC_PROMPT + (text ? `\nText to summarize: ${text}` : ""),
```

```
    uploadedFile
  );
  const end = performance.now();

  return NextResponse.json(
    { response, duration: end - start },
    { status: 200 }
  );
}
case GeneratingMethods.Firebase: {
  const base64File = buffer
    ? Buffer.from(buffer).toString("base64")
    : undefined;

  const request = await fetch(
    "https://generateairesponse-fz6rcejwuq-uc.a.run.app",
    {
      method: "POST",
      headers: { "Content-Type": "application/json" },
      body: JSON.stringify({
        text,
        file: base64File,
      }),
    }
  );

  const { response, duration } = (await request.json()) as {
    response: string;
    duration: number;
  };
```

```
        return NextResponse.json({ response, duration }, { status: 200 });
      }
    }
  } catch (error) {
    if (error instanceof Error) {
      return NextResponse.json({ error: error.message }, { status: 400 });
    }
  }
};
```

3.5 Реалізація генерації відповідей через Firebase Cloud Functions

Другий метод отримання відповідей від мовної моделі реалізовано шляхом винесення частини серверної логіки у хмарну інфраструктуру Firebase Cloud Functions. Такий підхід дає змогу порівняти час генерації відповідей у двох різних середовищах виконання: локальному серверному маршруті Next.js та віддаленій функції, що запускається у середовищі Google Cloud. Оскільки обидва методи використовують однакові вхідні дані, той самий промпт і той самий механізм звернення до Google Gemini API, інфраструктурні відмінності стають ключовим чинником, який впливає на результати експерименту.

Реалізація методу через Firebase охоплює окремий обробник HTTP-запитів, який працює незалежно від застосунку Next.js, проте реалізує ідентичну до нього логіку генерації респонсу, описану в минулому розділі.

Хмарна функція деплоїться через стандартний інтерфейс Firebase, а її вхідний точковий обробник задається через onRequest. Це дозволяє викликати її як звичайний HTTP-ресурс, що надходить від клієнтського застосунку. Таким чином формується окремий маршрут, який, на відміну від REST-шляху Next.js, виконується не у межах застосунку, а на стороні інфраструктури Firebase.

Лістинг 3.2 Функція generateAIReponse.ts, яка деплоїться в Firebase Cloud Functions:

```
interface RequestBody {
  text?: string;
  file?: string;
}

export const getAIResponse = async (request: Request, response: Response) =>
{
  try {
    const { text, file } = request.body as RequestBody;
    const arrayBuffer = file ? Buffer.from(file, "base64").buffer : undefined;
    const uploadedFile = arrayBuffer
      ? await uploadFileToGoogleApi(arrayBuffer)
      : undefined;
    const start = performance.now();
    const generatedResponse = await getGeminiResponse(
      BASIC_PROMPT + (text ? `\nText to summarize: ${text}` : ""),
      uploadedFile
    );
    const end = performance.now();
    response.status(200).json({
      response: generatedResponse,
      duration: end - start,
    });
  } catch (error) {
    return logger.error(error);
  }
};
```

3.6 Алгоритм завантаження файлів до Google API

Опрацювання файлів є важливою частиною роботи застосунку, оскільки мовна модель повинна отримувати PDF-документи у форматі, придатному для подальшого аналізу. Використання Google API забезпечує можливість завантажувати файл у хмарне сховище Google, після чого модель може отримувати до нього доступ у структурованому вигляді. У межах проєкту алгоритм обробки файлів повністю однаковий як у REST-реалізації, так і у варіанті з використанням Firebase Cloud Functions, що дозволяє отримувати коректні та порівнювані результати під час експериментального дослідження.

Процес обробки PDF-файлу розпочинається з перетворення бінарних даних, отриманих від користувача, у контейнер типу Blob. Це необхідний підготовчий етап, оскільки API завантаження у Google AI вимагає передання файлу саме у вигляді Blob-об’єкта. Після цього викликається метод googleai.files.upload, який завантажує файл на сервер. У межах запиту визначається лише базова конфігурація – наприклад, відображуване ім’я файлу, що не впливає на якість подальшої роботи моделі, але полегшує ідентифікацію файлу під час тестування. Після завантаження файлу API повертає інформацію про створений ресурс, однак цей ресурс ще не готовий до використання. Файл переходить у стан обробки, і модель не може працювати з ним до того часу, поки він не буде повністю індексований. Саме тому після завантаження ініціюється цикл очікування: функція періодично перевіряє стан файлу, повторно звертаючись до googleai.files.get з інтервалом у кілька секунд. Цей етап триває до моменту, поки файл не набуде стану, що дозволяє моделі отримати до нього доступ. У випадку переходу у стан, позначений як «FAILED», файл не може бути використаний у поточному запиті, і функція повертає нульовий результат.

Після завершення обробки Google API повертає файловий ресурс, який містить URI та MIME-тип. Ці параметри є обов’язковими для створення структурної одиниці Part, що використовується у запитах до Google Gemini. Значення URI визначає розташування файлу у хмарній інфраструктурі Google, а

MIME-тип дає змогу моделі правильно інтерпретувати структуру документа. Завдяки виклику createPartFromUri формується об’єкт, що може бути безпосередньо включений у запит до моделі разом із текстовими частинами.

Особливість роботи з Google API полягає в тому, що обробка файлів відбувається незалежно від конкретного запиту до мовної моделі. Тобто один і той самий PDF-документ може використовуватися у кількох запитах поспіль, якщо це потрібно. У межах цього проєкту файл завантажується під час кожного нового запиту, що дозволяє оцінити повний цикл роботи системи, включно з можливими затримками, пов’язаними з завантаженням. Але у масштабних системах допускається перевикористання вже завантажених файлів, що зменшує накладні витрати.

Лістинг 3.3 Функція завантаження файлу на Google API Cloud:

```
export const uploadFileToGoogleApi = async (file: ArrayBuffer) => {
  const blob = new Blob([file], { type: "application/pdf" });

  const uploadedFile = await googleai.files.upload({
    file: blob,
    config: {
      displayName: `file.pdf`,
    },
  });

  if (!uploadedFile.name) return;

  let getFile = await googleai.files.get({ name: uploadedFile.name });

  while (getFile.state === "PROCESSING") {
    await new Promise((resolve) => setTimeout(resolve, 2000));
    getFile = await googleai.files.get({ name: uploadedFile.name });
```

```
  }

  if (getFile.state === "FAILED" || !getFile.uri || !getFile.mimeType) {
    return;
  }

  return createPartFromUri(getFile.uri, getFile.mimeType);
};
```

3.7 Використання Google Gemini 2.5 Flash для генерації відповідей

Ядром обох механізмів генерації відповідей – REST-запиту та хмарної функції – є звернення до моделі Google Gemini 2.5 Flash. Вона забезпечує швидке формування узагальнених текстів та підтримує роботу з комбінованими вхідними даними, що включають текстові фрагменти та файли. Саме модель виконує основне обчислювальне навантаження в рамках застосунку і саме час її роботи порівнюється в кінцевому результаті.

Взаємодія із Gemini реалізується через офіційну бібліотеку @google/genai. За допомогою цього пакета формується клієнт GoogleGenAI, який ініціалізується з використанням API-ключа. Клієнт надає метод generateContent, що приймає модель, текстові частини запиту та, за потреби, файл. Під час формування запиту застосунок передає об’єкт contents, який складається з текстового промпту та структурованої частини uploadedFile у вигляді Part. Це дає моделі змогу отримувати доступ до завантаженого PDF-документа та обробляти його як єдиний контекст разом із текстовою інструкцією.

Важливою складовою стабільності роботи моделі є використання єдиного базового промпту BASIC_PROMPT.

Лістинг 3.4 Файл constants.ts, який експортує basicPrompt:

```
export const BASIC_PROMPT = `You are a precise and reliable text-summarization assistant.
Summarize the provided content clearly and concisely while preserving all essential information.
Follow these rules:
- Keep the summary objective and neutral.
- Maintain the original meaning without adding new ideas.
- Remove redundancies, filler text, and irrelevant details.
- Highlight key points, main arguments, important data, and conclusions.
- If the text is technical, preserve terminology but simplify explanations when possible.
- If the content contains a list or multi-step process, summarize it as a structured list.
- Do not omit critical context.
- If the text is excessively long or repetitive, compress it proportionally.

Return only the summarized text with no additional commentary. Limit your response by 500 symbols.
`;
```

Він містить набір інструкцій щодо того, яким має бути результат: узагальненість, стислість, відсутність зайвих міркувань та обмеження обсягу відповіді до п’ятисот символів. Завдяки стабільній структурі промпту результати різних запитів стають більш однорідними, що зменшує вплив варіацій у формуванні відповідей на результати вимірювання часу. В межах роботи метою є не якість формулювання, а швидкість генерації, тому обмеження довжини відповіді відіграє важливу роль у стандартизації умов.

Після відправлення запиту модель повертає структурований результат, з

якого застосунок отримує текстову частину через властивість response.text. У випадку, коли модель не може сформувати відповідь або повертає порожній результат, застосунок передає у відповідь службове повідомлення про помилку. Використання Part у contents дозволяє не лише передавати розміщений у Google API документ, але й структурувати запит таким чином, що модель отримує одночасно інструкцію та файл. Це дозволяє порівнювати як текстові запити, так і файлові, використовуючи один і той самий механізм генерації. Така гнучкість робить систему універсальною та розширюваною в майбутньому.

Лістинг 3.5 Функція генерації респонсів, час роботи якої є ключовою метрикою практичної частини:

```
export const googleai = new GoogleGenAI({ apiKey: process.env.GEMINI_API_KEY });

export const getGeminiResponse = async (
  prompt: string,
  uploadedFile?: Part
) => {
  const response = await googleai.models.generateContent({
    model: "gemini-2.5-flash",
    contents: [{ text: prompt }, ...(uploadedFile ? [uploadedFile] : [])],
  });

  return response.text || "Failed to generate response";
};
```

3.8 Тестування роботи застосунку та аналіз результатів

У межах практичної частини було проведено експериментальне

дослідження, спрямоване на вимірювання часу генерації відповідей мовною моделлю за двома різними підходами. Обидва методи використовували однакову модель, однакову структуру запиту та однаковий обсяг вхідних даних. Метою було оцінити вплив інфраструктури виконання на продуктивність, а також сформувати статистичні показники на основі багаторазових викликів.

Процес вимірювання здійснювався безпосередньо на серверному боці, що дозволило виключити затримки, пов’язані з рендерингом інтерфейсу або опрацюванням даних у браузері. Кожен запит фіксував момент початку звернення до моделі та час отримання відповіді. Результат передавався клієнтові у вигляді числового значення, що відображало лише тривалість роботи моделі.

Тестування виконувалось у двох режимах: на основі текстових запитів та на основі PDF-документів. В обох випадках вимірювався лише час, необхідний для аналізу та узагальнення короткого текстового фрагмента, час, що витрачався на завантаження файлу до Google API та інші затримки враховані не були. Усі значення тривалостей зберігались у вигляді вибірок для кожного з методів, після чого на основі цих вибірок обчислювалось середнє значення, що дозволяло усереднити ефект випадкових мережевих коливань (табл.3.1).

Таблиця 3.1 – Результати генерації 10 відповідей за текстовими вхідними даними

n ітерації	**Час генерації REST, ms**	**Час генерації FB, ms**
1	4008	6042
2	5352	4486
3	5193	4657
4	4508	4656
5	4833	4366
6	4116	5921
7	4069	6214
8	7867	13515
9	3377	5682
10	5210	3196

Представлені дані демонструють помітну варіативність часу виконання

для обох методів генерації. У межах десяти ітерацій REST-запити здебільшого мають меншу тривалість, хоча окремі значення демонструють значне збільшення часу, що свідчить про вплив мережевих або інфраструктурних факторів. Для Firebase Function характерна вища загальна розкиданість результатів, а також наявність декількох значень, що істотно перевищують середній рівень інших спроб. Водночас у певних ітераціях час роботи Firebase виявився меншим за час REST, що підкреслює несталість поведінки обох підходів у поодиноких викликах. На цьому етапі можна говорити про наявність тенденції до швидшої роботи REST-методу, однак остаточний висновок потребує врахування середнього значення за всіма десятьма спробами, яке подано далі у вигляді рисунку 3.2.

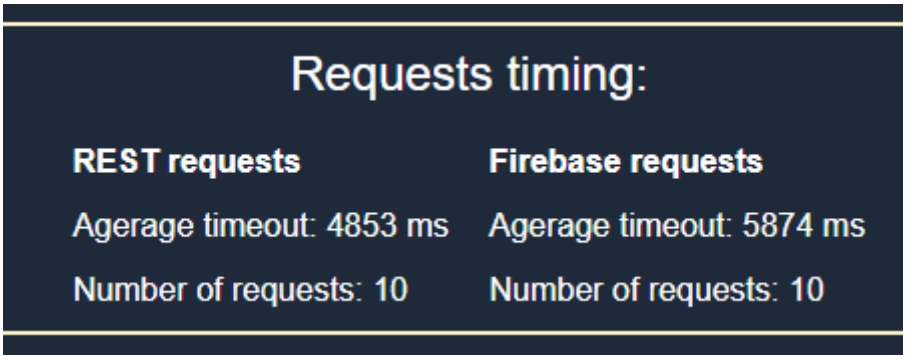

Рисунок 3.2 – Скріншот вебзастосунку з середнім значенням затримки відповіді для текстових вхідних даних

З урахуванням отриманих середніх значень часу обробки для обох методів можна сформулювати підсумковий висновок щодо експерименту з текстовими вхідними даними. Середній час генерації відповіді для REST-методу становив 4853 мс, тоді як для Firebase Function – 5874 мс. Різниця між цими показниками є сталою та зберігається незалежно від варіативності окремих ітерацій. Це свідчить про те, що за умов однакового промпту, однакової моделі та ідентичного обсягу вхідних даних REST-запити в середньому забезпечують менший час виконання.

Значення для Firebase демонструють як більшу середню тривалість, так і

ширший діапазон коливань, що може бути пов’язано з додатковими затримками, характерними для хмарних функцій, зокрема з мережевою інфраструктурою та часом активації середовища виконання. Незважаючи на наявність окремих випадків, коли Firebase працював швидше за REST, сукупні результати однозначно вказують на те, що в рамках текстових запитів REST-метод виявився продуктивнішим (табл.3.2).

Таблиця 3.2 – Результати генерації 10 відповідей за файлом в якості вхідних даних

n ітерації	**Час генерації REST, ms**	**Час генерації FB, ms**
1	6144	6122
2	6369	6305
3	4568	5518
4	9335	6866
5	7736	5837
6	5018	5949
7	11019	8207
8	5316	8069
9	5129	5842
10	4485	5322

Отримані результати демонструють значну варіативність часу обробки для обох методів у випадку роботи з PDF-файлом. У низці ітерацій час REST-запитів є суттєво більшим за час обробки у Firebase Function, а в інших – навпаки, Firebase демонструє помітно більшу тривалість. Обидва методи характеризуються нерівномірністю часу виконання, причому окремі значення для REST (наприклад, ітерації 4 та 7) виходять за межі загального діапазону, що впливає на загальну стабільність. Загалом на основі поодиноких спроб сформувати однозначну тенденцію складно, оскільки методи демонструють близькі значення у більшості ітерацій, але з окремими піковими відхиленнями.

Остаточне співвідношення між ними можна оцінити після аналізу середніх значень, наведених на рисунку 3.3.

Рисунок 3.3 – Скріншот вебзастосунку з середнім значенням затримки відповіді для файлових вхідних даних

Середній час генерування відповіді за REST-методом становив 6512 мс, тоді як для Firebase Function середнє значення дорівнювало 6404 мс. Отримані результати свідчать про практично однакову швидкість роботи обох методів у разі використання файлу як вхідних даних. Наявна різниця між середніми значеннями є мінімальною і не демонструє чіткої переваги одного з підходів. Водночас для обох методів характерні окремі значення з помітними піковими затримками, що вказує на вплив змінних мережевих умов та роботи сервісу Google API. У межах файлових запитів обидва підходи демонструють подібну загальну продуктивність, і наявна різниця не є суттєвою з практичної точки зору.

ВИСНОВКИ

Дослідження продемонструвало, що продуктивність вебзастосунків, які взаємодіють із сервісами штучного інтелекту, формується комплексом взаємопов’язаних факторів: архітектурною організацією системи, способом інтеграції зовнішніх AI-компонентів, механізмами обміну даними, параметрами навантаження та характеристиками самих моделей. Аналіз сучасних підходів до створення вебзастосунків і огляд наявних методів інтеграції дав змогу виокремити ключові відмінності між REST, WebSocket та gRPC, зокрема щодо затримки, підтримки потокового передавання, складності впровадження та типових сценаріїв використання. Порівняльні характеристики цих механізмів показали помітні відмінності у швидкодії та придатності кожного з них до роботи у режимі реального часу.

Теоретичний аналіз дозволив структурувати основні метрики оцінювання ефективності інтеграції AI-компонентів: час відповіді, пропускну здатність, параметри масштабованості та стабільність системи в умовах збільшеного навантаження. Вивчення обчислювальних властивостей моделей, механізмів паралельної обробки, черг запитів і мережевих затримок стало основою для побудови математичних моделей, які описують залежності між складністю AI-запиту, інфраструктурними характеристиками та поведінкою системи при зростанні кількості одночасних звернень.

Моделювання різних стратегій обробки запитів дозволило оцінити вплив обчислень на стороні AI-сервісу, мережевих коливань і характеру взаємодії компонентів у складних системах. Порівняння підходів локального розгортання моделей, розподілу інтелектуальних компонентів через gateway-шар і використання edge-обробки дало можливість виокремити сильні та слабкі сторони кожного варіанту. Локальні рішення гарантують мінімальні затримки, але обмежують масштабованість і ускладнюють інфраструктуру; gateway-підхід пропонує збалансовану модель керування навантаженням; edge-обробка є перспективним шляхом для систем з високими вимогами до часу відповіді та

доступністю попередньо оброблених даних.

Практична частина роботи підтвердила теоретичні положення на експериментальному прототипі вебзастосунку. Реалізовано клієнтську частину, серверну логіку на базі Next.js, механізм генерації відповідей через Firebase Cloud Functions, інтеграцію з Google AI File API та використання моделі Gemini

2.5 Flash для аналізу даних і формування відповідей. Проведені експерименти дали змогу простежити динаміку часу відповіді, стабільність системи під навантаженням, вплив архітектури на затримку та ефективність обробки файлів. Отримані результати збігаються з висновками математичного моделювання і підтверджують значущість правильно обраного механізму інтеграції, стратегії паралельної обробки та інфраструктурних рішень.

Визначено сценарії, у яких REST-підхід є достатнім, випадки, де WebSocket забезпечує найкращий баланс між швидкодією та стабільністю, та ситуації, у яких gRPC надає максимальну ефективність у мікросервісних середовищах зі значною кількістю двосторонніх потокових взаємодій. Зроблено висновки щодо застосування локальних інференс-серверів, оптимізації навантаження, використання черг, кешування та адаптивних стратегій обробки, що є критично важливим для систем зі складними AI-компонентами.

Результати роботи доводять, що поєднання теоретичного моделювання, аналізу архітектур і практичної реалізації дає всебічне бачення впливу інтеграції AI-сервісів на швидкодію вебзастосунків. Сформовані рекомендації можуть використовуватися для проєктування високопродуктивних систем, розробки корпоративних рішень, впровадження інтелектуальних сервісів у вебзастосунки та оптимізації існуючих архітектурних моделей. Створена експериментальна модель і отриманий аналітичний матеріал підтверджують досягнення поставленої мети та відкривають можливості для подальшого вдосконалення, зокрема в напрямах edge-AI, інтелектуального балансування та динамічного керування навантаженням у системах реального часу.

ПЕРЕЛІК ДЖЕРЕЛ ПОСИЛАННЯ

1. A Survey on Web Testing: On the Rise of AI and Applications in Industry, Kertusha Iva et al. (2025) *arXiv preprint*.

2. Ahmed H. Ali. (2024) Innovative Strategies for Enhancing Web Application Performance: A Contemporary Load Testing Approach, *International Journal of Engineering Trends and Technology*, 72(10), pp. 246-256.

3. Al-Mansour K., Petrescu D. (2024) Compression and Serialization Techniques to Reduce Network Overhead for AI APIs, *IEEE Communications Surveys & Tutorials*, 26(1), pp. 150–167.

4. Aluwala Aakash. (2024) Optimizing IT Operations with AI-Driven Application Performance Management, *J Comput Sci Software Dev*, 3:204.

5. An Empirical Study of AI Techniques in Mobile Applications, Li Yinghua et al. (2022) *arXiv preprint*.

6. Anatomizing Deep Learning Inference in Web Browsers, Wang Qipeng et al. (2024) *arXiv preprint*.

7. Ayyagari Aravind; Goel Punit; Renuka A. (2024) Leveraging AI and Machine Learning for Performance Optimization in Web Applications, *Darpan International Research Analysis*, 12(2), pp. 199-218.

8. Bianchi F., Korhonen J. (2024) Ensemble Caching for Multimodal AI Requests in Distributed Web Systems, *Journal of Parallel and Distributed Computing*, 182(5), pp. 71–89.

9. Choithramani D. (2024) Integration of Artificial Intelligence in Web Development, *Harrisburg University Digital Commons*.

10. Enhancing User Experience in Full Stack Web Applications by AI/ML APIs (2025) *The Academic*, Article.

11. Enhancing Web Application Performance with AI-Driven Optimization Techniques (2021) *International Journal of Scientific & Research Publications (IJSR)*, 10(2).

12. Esposito V., Novak M., Ramakrishnan S. (2023) Throughput Modeling

for AI-Backed Microservices under Bursty Traffic, *Software: Practice and Experience*, 53(11), pp. 2104–2123.

13. Fernandez G., Liu X. (2024) Edge-AI Orchestration Techniques to Reduce Latency in Interactive Web Applications, *IEEE Transactions on Cloud Computing*, 12(3), pp. 345–360.

14. Ghattas M., Mora A. M., Odeh S. (2025) A Novel Approach for Evaluating Web Page Performance Based on Machine Learning Algorithms and Optimization Algorithms, *AI*, 6(2), pp. 19.

15. Gowrigari Sudheer Kumar Reddy. (2022) AI-Driven Web Performance Optimization: Striking the Perfect Balance between Speed, Security, and W3C Standards, *Asian Journal of Multidisciplinary Research & Review*, 3(6), pp. 231-242.

16. Huang Y., O'Neill P. (2025) Benchmarks for LLM Inference Latency in Serverless Environments, *International Journal of Cloud Computing*, 14(1), pp. 33–49.

17. Integrated AI in web development: The future of smart web applications (2024) *IJRPR*, V6 (5).

18. K. Mani. (2025) Machine learning models in web applications, *ScienceDirect – Journal Article*.

19. Kim J., Alvarez P., Sokolov D. (2023) Evaluating WebSocket vs gRPC for Token-Level Streaming in Conversational AI, *ACM Transactions on Internet Technology*, 23(1), pp. 1–19.

20. Nourikhah H., Akbari M.K., Kalantari M. (2016) Modeling and predicting measured response time of cloud-based web services using long-memory time series, *arXiv preprint*.

21. Novakova T., Singh A., Müller C. (2023) Reliability Engineering for AI-Enhanced Web Services: Retries, Timeouts, and Backoff Strategies, *IEEE Software*, 40(4), pp. 44–53.

22. Oliveira R., Gupta S. (2024) Asynchronous Architectures for Scalable AI-Driven Web Services, *International Journal of Web Engineering and Technology*, 18(2), pp. 205–222.

23. Pandiya Dileep Kumar; Charankar Nilesh. (2023) Integration of Microservices and AI for Real-Time Data Processing, *International Journal of Computer Engineering and Technology (IJCET)*, 14(2), pp. 240-254.

24. Park S., Mensah K., Zhao H. (2025) Privacy-Preserving Edge Inference for Web Applications Using Federated LLMs, *Neural Networks*, 171(2), pp. 299– 317.

25. Pereira L., Novak M., Singh A. (2024) Adaptive Caching Strategies for Low-Latency AI Inference in Web Applications, *Journal of Network and Computer Applications*, 210(4), pp. 112–127.

26. Rahman M., Ivanov A. (2025) Hybrid Edge–Cloud Architectures for Multimedia AI in Real-Time Web Apps, *Sensors*, 25(7), pp. 3421–3438.

27. Rossi E., Gómez R. (2024) Effective Queueing Policies for Prioritizing Real-Time Requests to AI Backends, *Performance Evaluation*, 166(3), pp. 102–120.

28. Saito K., Brown L. (2023) Measuring Perceived Latency in Streaming LLM Interfaces, *Human–Computer Interaction Journal*, 39(6), pp. 513–532.

29. Thatikonda Kalyan Chakravarthy. (2025) Methods and Processes for Optimization of Cloud API Performance through AI-Based Machine Learning Ensemble Cache Management, *International Journal of Advanced Research in Engineering and Technology (IJARET)*, 16(1), pp. 662-672.

30. Towards a World Wide Web powered by generative AI, AlDahoul N., Hong J., Varvello M. et al. (2025) *Scientific Reports*, 15, Article 7251.

31. Unleashing the Power of AI. A Systematic Review of Cutting-Edge Techniques in AI-Enhanced Scientometrics, Webometrics, and Bibliometrics, Saeidnia H.R. et al. (2024) *arXiv preprint*.

32. Vepsäläinen Juho; Hellas Arto; Vuorimaa Petri. (2024) Overview of Web Application Performance Optimization Techniques, *arXiv/preprint*.

33. Zhao H., Martínez R. (2025) Real-Time Stream Processing for LLM-Powered Web Services, *Future Generation Computer Systems*, 148(2), pp. 58–73.