МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ

Український державний університет науки і технологій

Кафедра «Комп'ютерні інформаційні технології»

Пояснювальна записка

до кваліфікаційної роботи
ОС Магістр

на тему: «Дослідження продуктивності GraphQL при використанні у веб-додатках»
за освітньою програмою: «12 Інженерія програмного забезпечення»
зі спеціальності: «121 Інженерія програмного забезпечення»

Виконав: студент групи ПЗ2422: / Андрій ГРИГОРЕНКО /

Керівник: / Вадим ГОРЯЧКІН /

Нормоконтролер: / Світлана ВОЛКОВА /

Засвідчую, що у цій роботі немає запозичень з праць інших авторів без відповідних посилань.
Студент

Дніпро – 2026

MINISTRY OF EDUCATION AND SCIENCE OF UKRAINE

Ukrainian State University of Science and Technologies

Department «Computer information technology»

Explanatory Note
to Master's Thesis

on the topic: «Research on GraphQL productivity in web applications»
according to educational curriculum «12 Software engineering»
in the Speciality: «121 Software engineering»

Done by the student of the group PZ2422: / Andrii HRYHORENKO /

Scientific Supervisor: / Vadym HORIACHKIN /

Normative controller: / Svitlana VOLKOVA /

Dnipro – 2026

Міністерство освіти і науки України
Український державний університет науки і технологій

Факультет: «Комп'ютерні технології і системи»
Кафедра: «Комп'ютерні інформаційні технології»
Рівень вищої освіти: магістр
Освітня програма: «12 Інженерія програмного забезпечення»
Спеціальність: «121 Інженерія програмного забезпечення»

ЗАТВЕРДЖУЮ
Завідувач кафедри КІТ
_____ /Вадим ГОРЯЧКІН/
(підпис)

Дата _____

З А В Д А Н Н Я

на кваліфікаційну роботу Магістр

студенту Григоренко Андрію Леонідовичу

1. Тема дипломної роботи: Дослідження продуктивності GraphQL при використанні у веб-додатках

Керівник роботи: Горячкін В. М., к.т.н., доцент

затверджені наказом 1401 ст від «02» жовтня 2025 р

2. Термін подання студентом роботи: 19 січня 2026 р
3. Вихідні дані до дипломної роботи: технічне завдання на розробку системи програмного інтерфейсу для ведення блогу; офіційна специфікація мови запитів GraphQL (GraphQL Foundation); науково-технічна література з питань тестування продуктивності веб-додатків.
4. Зміст пояснювальної записки (перелік питань до розробки) складається з вступу, аналізу розвитку технологій GraphQL, розробки і тестування веб додатку, дослідження ефективності GraphQL, загальних висновків та бібліографічного списку.
5. Перелік демонстраційного матеріалу: відео-демонстрація роботи програми.

КАЛЕНДАРНИЙ ПЛАН

№ з/п	Назва етапів кваліфікаційної роботи	Строк виконання етапів роботи	Примітка
1	Вступ		
2	Аналіз сучасного стану дослідження проблеми за науковими літературними джерелами	01.09.25 – 20.09.25	
3	Аналіз сучасного стану програмно-апаратного забезпечення, яке потребує вдосконалення для вирішення проблем дослідження	21.09.25 – 09.11.25	
5	Постановка задачі, технічне завдання	10.11.25 – 16.11.25	30%
6	Розробка інструментальних засобів дослідження	17.11.25 – 07.12.25	
7	Виконання досліджень	08.12.25 – 12.12.25	60%
8	Оформлення тез доповідей	13.12.25 – 14.12.25	
9	Розробка, узгодження та затвердження програмної документації	15.12.25 – 07.12.25	
10	Розробка демонстраційних матеріалів	07.01.26 – 15.01.26	100%
11	Подання кваліфікаційної роботи до кафедри	16.01.26	
12	Захист кваліфікаційної роботи на засіданні Екзаменаційної комісії	20.01.26	

Студент _____ Григоренко А.Л.
(підпис)

Керівник роботи _____ Горячкін В.М.
(підпис)

РЕФЕРАТ

Об’єктом дослідження є серверні підсистеми веб-API.

Предметом дослідження є експлуатаційні характеристики під навантаженням у типових сценаріях (читання/запис).

Метою роботи є визначення ефективності GraphQL і REST API при використанні у веб-застосунках.

Методи дослідження: емпіричне порівняння обраних фреймворків, їх швидкодії у різних ситуаціях.

Результати та їх новизна: дослідження робить внесок у визначення практичної застосовності різних веб-API при використанні у веб-застосунку. Результати дослідження дозволяють зробити висновки щодо того в яких ситуаціях той чи інший API показує кращі результати.

Пояснювальна записка складається зі вступу, 4 розділів, висновків, бібліографічного списку та 4 додатків:

- у вступі описується сутність розробки, її актуальність. Складається із 3 сторінок;
- у першому розділі висвітлюються ключові аспекти REST та GraphQL, пояснюються проблеми, які він вирішує, і аналізуються наявні проблеми та дослідження продуктивності. Обгрунтовується вибір напряму для подальшого дослідження. Складається з 11 сторінок;
- у другому розділі надано аналіз завдань роботи, надано обґрунтування експериментального методу дослідження. Складається з 22 сторінок;)
- у третьому розділі представлене проектування й розробка аналітичного додатку для дослідження. Складається з 7 сторінок;
- у четвертому розділі описано виконані дослідження. Складається з 32 сторінкок;
- додатки містять технічне завдання, текст програми, керівництво користувача та тези з конференції.

Таблиць – 19, рисунків – 17, бібліографія – 54 джерел.

Ключові слова: веб-додаток, rest api, graphql, продуктивність, навантажувальне тестування, час відгуку, over-fetching, ruby on rails.

ЗМІСТ

ПЕРЕЛІК УМОВНИХ ПОЗНАК, СИМВОЛІВ, ОДИНИЦЬ, СКОРОЧЕНЬ І ТЕРМІНІВ

API (Application Programming Interface) – інтерфейс прикладного програмування.

CRUD (Create, Read, Update, Delete) – чотири базові функції управління даними (створення, читання, оновлення, видалення).

GraphQL (Graph Query Language) – мова запитів до API та середовище виконання цих запитів.

JSON (JavaScript Object Notation) – текстовий формат обміну даними, заснований на JavaScript.

KLOC (Kilo Lines Of Code) – тисяча рядків програмного коду (метрика оцінки обсягу ПЗ).

MVC (Model-View-Controller) – схема розділення даних додатка, інтерфейсу користувача та керуючої логіки.

ORM (Object-Relational Mapping) – технологія програмування, яка пов'язує бази даних з концепціями об'єктно-орієнтованих мов.

RAM (Random Access Memory) – оперативна пам'ять.

REST (Representational State Transfer) – архітектурний стиль взаємодії компонентів розподіленого додатка в мережі.

RSS (Resident Set Size) – частка пам'яті, яку займає процес в оперативній пам'яті (RAM).

SDL (Schema Definition Language) – мова визначення схеми, що використовується в GraphQL.

SOAP (Simple Object Access Protocol) – простий протокол доступу до об'єктів.

SPA (Single Page Application) – односторінковий веб-додаток.

SQL (Structured Query Language) – мова структурованих запитів.

URI (Uniform Resource Identifier) – уніфікований ідентифікатор ресурсу.

URL (Uniform Resource Locator) – уніфікований покажчик ресурсу (веб-адреса).

ВСТУП

Актуальність теми. В умовах висококонкурентного веб-середовища вибір стилю прикладного інтерфейсу (REST чи GraphQL) безпосередньо впливає на продуктивність, масштабованість і вартість експлуатації систем. GraphQL обіцяє усунути надлишкову і недостатню передачу даних завдяки вибірковості полів і можливості агрегувати пов’язані дані за один запит, тоді як REST вирізняється простотою, зрілою екосистемою кешування та прогнозованою поведінкою кінцевих точок. Попри велику кількість публікацій, бракує порівнянь у реалістичних умовах із одночасним урахуванням затримки, навантаження ЦП/ОП і мережевих витрат, а також з аналізом впливу «мінімальної» вибірки полів у GraphQL. Це зумовлює потребу в системному експериментальному дослідженні на репрезентативній предметній області.

Предмет і об’єкт дослідження. Об’єктом дослідження є серверні підсистеми веб-API, реалізовані у стилях REST і GraphQL на базі типової контент-орієнтованої предметної області (користувачі, публікації, коментарі) з реляційними зв’язками. Предметом дослідження виступають експлуатаційні характеристики цих підходів під навантаженням у типових сценаріях (читання переліку користувачів, читання переліку публікацій із даними автора, створення публікації) за сукупністю метрик: час відгуку (медіанне значення, 90-й перцентиль, максимальне), завантаження центрального процесора, споживання оперативної пам’яті та обсяги переданих/отриманих даних.

Мета роботи. Мета роботи полягає в емпіричному порівнянні REST і GraphQL у веб-додатках на реалістичному прикладі з відтворюваною методикою вимірювань та формулюванні інженерних рекомендацій щодо вибору API і дисципліни вибірки даних.

Задача дослідження. Задачею дослідження є проектування предметної області (користувачі, публікації, коментарі) з індексами, що відображають виробничі практики, реалізація і дослідження паритетних REST-ендпоінтів і GraphQL-схеми.

Методи дослідження. Методологія поєднує теоретичний і емпіричний компоненти.

Аналітичні методи: критичний огляд архітектурних особливостей REST і GraphQL (маршрутизація, селекція полів, кешування), формулювання гіпотез щодо впливу проєкції полів і структури даних на продуктивність.

Експериментальні методи: навантажувальне тестування, збір системних метрик процесу (завантаження процесору та оперативної пам'яті, мережі).

Наукова новизна. Вперше в узгоджених умовах для стеку Rails 8 виконано цілісне емпіричне порівняння трьох практик взаємодії клієнт-сервер – REST, GraphQL із повною відповіддю та GraphQL з мінімальною відповіддю – з одночасною оцінкою чотирьох класів метрик (затримки: медіанне значення, 90-й перцентиль, максимальне; ЦП; ОП; мережа) у паритетних сценаріях читання/запису. Набули подальшого розвитку кількісні уявлення про компроміси GraphQL: показано, що дисципліна вибірки полів (мінімальна проекція) здатна одночасно зменшити латентність, навантаження на ЦП/ОП і мережеві обсяги у сценаріях з ієрархічними даними, тоді як «повні» відповіді можуть індукувати надмірне навантаження мережі і погіршення ресурсних показників.

Практичне значення. Внесок цієї роботи полягає у формуванні узагальненої, відтворюваної методики кількісної оцінки продуктивності API та у поєднанні декількох вимірів ефективності в єдиний протокол порівняння. Запропонований підхід зміщує акцент із синтетичних бенчмарків на прикладне середовище (реальна предметна область, індекси, типові сценарії читання/запису) та системно враховує не лише часові характеристики, а й ресурсні та мережеві витрати, що дозволяє робити інженерно корисні висновки.

Особистий внесок здобувача полягає у аналізі літератури з теми дослідження, розгляді відомих веб-API, проведення аналізу продуктивності використання цих API у веб-додатках. В якості експериментальної частини атестаційної роботи було розроблено серверний додаток з можливістю використання REST-ендпоінтів і GraphQL-схеми в паритетних умовах.

Апробація результатів дослідження та публікації

Основні положення магістерської роботи доповідалися та були схвалені на XIX Міжнародної науково-практичної конференції «Сучасні інформаційні та

комунікаційні технології на транспорті, в промисловості і освіті» (Дніпро, ДНУЗТ, 2025 року) та семінарі кафедри КІТ (08.01.2026).

.

1 ОГЛЯД СУЧАСНОГО СТАНУ ВЕБ-API І ПОСТАНОВКА ПРОБЛЕМИ

1.1 Огляд підходів до вибору та використання веб API

Еволюція підходів до побудови API пройшла кілька хвиль: від сервіс-орієнтованих технологій (SOAP/WSDL) із жорсткими контрактами і складними протоколами до легковагих HTTP-інтерфейсів у стилі REST (Representational State Transfer) [1], які використовують прості формати (JSON) і семантику методів HTTP. Подальший розвиток зумовив появу підходів, що по-різному вирішують проблему ефективної доставки саме потрібних даних: RPC-підходи (JSON-RPC, gRPC), запитні (query-based) підходи, такі як GraphQL [2], а також подійні способи інтеграції (WebSockets, Server-Sent Events, webhooks). Кожен підхід пропонує власний компроміс між еволюційністю контракту, зчепленням клієнт-сервер, безпекою та продуктивністю.

З погляду продуктивності, ключові тригери відмінностей між підходами – це кількість мережевих звернень, обсяг переданих/отриманих даних, серверні накладні на формування відповіді, можливість ефективного кешування й контроль складності запитів. REST зазвичай виграє на «плоских» вибірках завдяки невеликому серверному оверхеду і простоті кешування, але програє на складних ієрархіях через надлишок даних і додаткові раунди звернень. Запитні підходи, насамперед GraphQL, мінімізують обсяг відповіді та кількість звернень, проте несуть витрати на планування і виконання запитів, а також потребують інженерних заходів проти N+1, обмеження складності й продумані практики кешування [3].

Домінуючу роль в розробці веб додатків займають REST і GraphQL підходи. REST де-факто є базовим стандартом публічних API, а GraphQL широко використовується для складних фронтендів і мобільних клієнтів (Facebook, GitHub, Shopify тощо). Вони репрезентують два полюси проектування: «ресурс, визначений сервером» (REST) проти «форма даних, визначена клієнтом» (GraphQL). Дивись рисунок 1.1.

Альтернативи свідомо не включено до експериментальної частини з методичних міркувань. Вони є менш популярними, суттєво відрізняються

транспортом і клієнтськими вимогами, що робить пряме зіставлення із браузерними сценаріями на HTTP/1.1 некоректним [4].

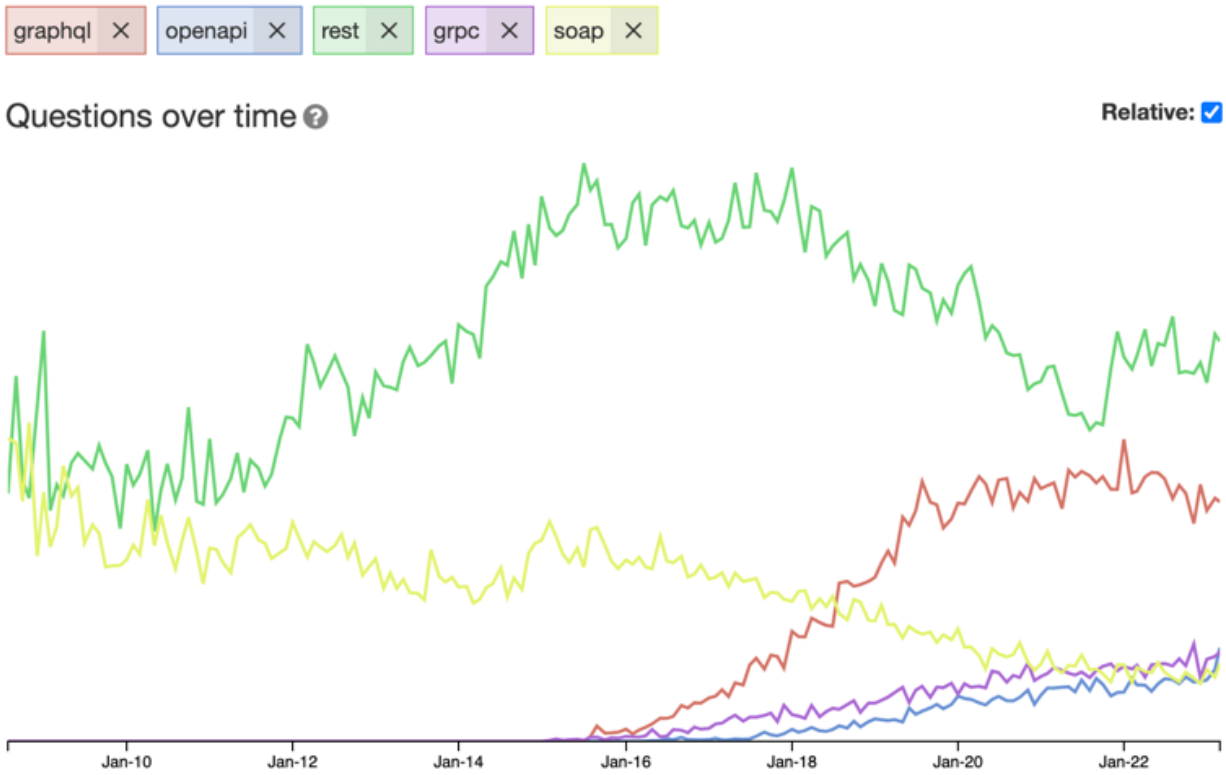

Рисунок 1.1 – Відносна кількість питань на StackOverflow

1.2 Аналіз традиційного підходу REST API

Протягом десятиліть архітектурний стиль REST був фундаментальним підходом, що домінував у розробці веб-сервісів. Запропонований у 2000 році, він здобув широке визнання завдяки своїй простоті та ефективному використанню існуючих стандартів HTTP [5]. Стратегічна важливість аналізу його основних принципів, переваг та недоліків полягає в необхідності зрозуміти еволюцію веб-API та контекст, що зумовив появу нових технологій, таких як GraphQL. Хоча REST заклав основи для взаємодії розподілених систем, зростання складності сучасних клієнтських додатків виявило його іманентні обмеження [6]. Цей підрозділ має на меті систематично деконструювати історичний контекст, ключові принципи, практичні конвенції та фундаментальні компроміси REST, щоб сформувати вичерпну теоретичну основу для подальшого порівняльного аналізу.

Важливо підкреслити, що REST – це не протокол чи стандарт, а набір архітектурних обмежень, призначених для створення масштабованих, надійних та простих у підтримці веб-сервісів. Дотримання цих обмежень – а саме: клієнт-сервер, відсутність стану, кешування, єдиний інтерфейс, багаторівнева система та код на вимогу – дозволяє системі вважатися "RESTful"(повноцінна REST архітектура) [7]. До появи REST домінуючим підходом для взаємодії веб-сервісів був SOAP, який характеризувався як надмірно складний [8]. На противагу цьому, REST запропонував кардинально інший шлях: замість створення нових протоколів він використав існуючі та добре зрозумілі стандарти HTTP. Ця простота значно знизила поріг входження для розробників та прискорила інтеграцію систем, що зробило REST де-факто стандартом для більшості веб-API на наступні два десятиліття. Цей еволюційний стрибок став можливим завдяки чітко визначеним архітектурним принципам, що є ядром філософії REST.

Теоретичні принципи REST трансформуються в конкретні інженерні практики та загальноприйняті домовленості. Одним з ключових аспектів, що сприяли популярності REST, є філософія "домовленості замість конфігурацій" (Convention over Configuration), яка є центральною, наприклад, для фреймворку Ruby on Rails [9]. Це дозволяє розробникам зосередитись на бізнес-логіці, а не на рутинних налаштуваннях. Ключові аспекти реалізації RESTful API можна представити наступним чином [1, 5, 7]:

1. Використання методів HTTP: REST повністю покладається на семантику стандартних методів HTTP для виконання операцій над ресурсами. Це зіставлення є інтуїтивно зрозумілим і відповідає парадигмі CRUD (Create, Read, Update, Delete). Дивись таблицю 1.1.

Таблиця 1.1 – Відповідність методів HTTP та REST

Метод HTTP	Операція
GET	Read (Отримання)
POST	Create (Створення)
PUT / PATCH	Update (Оновлення)
DELETE	Delete (Видалення)

2. Структура URI: Загальноприйнятою конвенцією є використання іменників у множині для іменування колекцій ресурсів (наприклад, /articles замість /getArticles), що робить URI логічними та орієнтованими на дані, а не на дії. Ієрархічні зв'язки між ресурсами відображаються через вкладеність URI (наприклад, /articles/{id}/comments).

3. Формат даних: Хоча REST не обмежує формат представлення даних, JSON (JavaScript Object Notation) де-факто став галузевим стандартом, витіснивши XML. Причинами цього є його лаконічність, краща читабельність для людини та проста інтеграція з JavaScript, що є основою більшості сучасних клієнтських додатків [3].

4. Обробка помилок: Для інформування клієнта про результат операції використовуються стандартні коди стану HTTP (200 OK, 201 Created, 404 Not Found, 500 Internal Server Error). Це дозволяє уніфікувати обробку результатів. Ключовою практикою є повернення у тілі відповіді консистентного JSON-об'єкта з деталями помилки (наприклад, { "error": { "message": "Resource not found", "code": 404 } }), що забезпечує клієнта машиночитними даними для надійної обробки виняткових ситуацій [10].

Ці практичні аспекти забезпечили REST широке розповсюдження, що було зумовлено його численними перевагами. Незважаючи на ці значні переваги, які закріпили за REST статус галузевого стандарту, еволюція веб-додатків у бік більшої складності та інтерактивності виявила його суттєві обмеження.

Попри свою ефективність та популярність, архітектура REST стикається з серйозними викликами при роботі зі складними клієнтськими додатками, такими як

сучасні односторінкові (SPA) та мобільні застосунки. Жорстка структура ендпоінтів та природа взаємодії, орієнтованої на ресурси, породжує проблеми, які стали рушійною силою для пошуку альтернатив, зокрема GraphQL [11].

Надмірне отримання даних (Over-fetching): Ця проблема виникає, коли клієнт отримує від сервера значно більше даних, ніж йому потрібно для відображення конкретного UI-компонента, оскільки REST-ендпоінт повертає повне представлення ресурсу з фіксованим набором полів. Наприклад, для відображення лише імені персонажа "Люк Скайвокер" клієнт, що звертається до REST API, може бути змушений завантажити весь об'єкт, який містить понад 15 додаткових полів, як-от зріст, маса, колір волосся тощо. Це призводить до марної витрати мережевого трафіку та уповільнення роботи додатку, особливо на мобільних пристроях з обмеженим з'єднанням.

Недостатнє отримання даних (Under-fetching): Це зворотна проблема, коли одного запиту недостатньо для отримання всієї необхідної інформації, і клієнту доводиться робити кілька послідовних запитів до різних ендпоінтів, щоб зібрати дані для одного екрану. Це створює так званий "водоспад" мережевих запитів (network waterfall), що суттєво збільшує загальну затримку. Яскравим прикладом є платформа IRIS, де для відображення однієї таблиці проєктів клієнт був змушений виконати шість окремих запитів до API для отримання даних про проєкт, його типологію, фінансування, команду та іншу пов'язану інформацію [12].

Отже, проблеми надмірного та недостатнього отримання даних, у поєднанні з жорсткістю структури відповідей та викликами версіонування, створюють значні перешкоди для продуктивності та гнучкості сучасних додатків. Ці недоліки вказують на необхідність дослідження альтернативного підходу, який би дозволив клієнту самому визначати, які дані йому потрібні, – підходу, реалізованого в GraphQL.

1.3 Спроби вирішення проблем продуктивності за допомогою GraphQL API

GraphQL виник як пряма відповідь на фундаментальні обмеження архітектури REST, зокрема на проблеми надлишкового та недостатнього отримання даних. Цей

підхід, розроблений компанією Facebook у 2012 році та відкритий для спільноти у 2015 році, пропонує клієнтоорієнтовану модель взаємодії, де клієнт, а не сервер, визначає точну структуру та обсяг необхідних даних [13]. Замість жорстко визначених ендпоінтів, GraphQL надає єдину точку доступу та потужну мову запитів, що дозволяє клієнтським додаткам отримувати всю необхідну інформацію за один мережевий запит. Цей підрозділ детально аналізує фундаментальні принципи, архітектурні переваги та ключові виклики, пов'язані з використанням GraphQL у сучасних розподілених системах.

Для повного розуміння переваг GraphQL стратегічно важливо проаналізувати його фундаментальні архітектурні відмінності від традиційних підходів, таких як REST. Ці принципи безпосередньо впливають не лише на продуктивність та гнучкість системи, але й на досвід розробників, спрощуючи клієнтську логіку та покращуючи взаємодію між командами. Ці чотири стовпи – єдина точка доступу, строга схема, ієрархічні запити та розділення операцій – функціонують не ізольовано, а як єдина система, що переносить контроль над формою даних від сервера до клієнта.

Ключові архітектурні концепції GraphQL можна синтезувати наступним чином:

1. Декларативне отримання даних через єдину точку доступу (Single Endpoint): На відміну від імперативного підходу REST, де клієнт робить запити до множини ендпоінтів для різних ресурсів (наприклад, /users, /articles/:id), GraphQL дозволяє клієнту декларативно описати всю необхідну структуру даних в одному запиті до єдиної точки доступу, зазвичай /graphql. Клієнти надсилають на цей ендпоінт POST-запити, що містять структуру потрібних даних. Такий підхід значно спрощує клієнтську логіку, оскільки відпадає потреба керувати великою кількістю URL, та централізує керування API на стороні сервера.

2. Строго типізована схема (Schema Definition Language): Основою будь-якого GraphQL API є схема, визначена за допомогою мови SDL (Schema Definition Language). Схема виступає як формальний "контракт" між клієнтом та сервером, чітко описуючи всі доступні типи даних, поля та операції [GraphQL Foundation. (2025)]. Строга типізація не лише запобігає синтаксичним помилкам на етапі розробки, але й фундаментально підвищує надійність системи, уможливлюючи статичний аналіз

коду, автоматичну генерацію клієнтських бібліотек та валідацію запитів на рівні інфраструктури, що унеможливлює цілий клас помилок, пов'язаних з невідповідністю даних. Крім того, завдяки механізму інтроспекції, схема забезпечує самодокументування API, що дозволяє розробникам інтерактивно досліджувати його можливості.

3. Ієрархічна структура запитів та відповідей: Однією з головних переваг GraphQL є те, що структура запиту точно відповідає структурі відповіді у форматі JSON. Це дозволяє клієнтам отримувати вкладені та пов'язані дані (наприклад, користувача та всі його публікації) за один мережевий запит, природно відображаючи зв'язки між сутностями. Запит має ієрархічну форму, подібну до графа, що є інтуїтивно зрозумілим для представлення складних доменних моделей.

4. Розділення операцій (Queries, Mutations, Subscriptions): GraphQL чітко розмежовує три основні типи операцій для взаємодії з даними. Запити (Queries) використовуються виключно для читання даних. Мутації (Mutations) призначені для зміни даних: створення, оновлення або видалення (CRUD). Таке чітке розділення покращує передбачуваність та керованість операцій, що модифікують стан системи. Підписки (Subscriptions) дозволяють клієнтам підписуватися на події та отримувати оновлення в реальному часі від сервера, що є потужним інструментом для побудови інтерактивних додатків.

Гнучкість GraphQL не є абстрактною перевагою; вона є прямим інструментом для оптимізації мережевої взаємодії шляхом вирішення двох хронічних проблем REST API: надлишкового та недостатнього отримання даних. Архітектурні принципи, розглянуті вище, дозволяють клієнту точно контролювати обсяг та структуру інформації, що передається мережею.

Клієнт у своєму запиті чітко вказує лише ті поля, які йому необхідні. Замість отримання повної інформації про користувача, клієнт може надіслати запит { user(id: 1) { name, email } } і отримати у відповідь лаконічний JSON, що містить виключно ці два поля. Дослідження показують, що такий підхід може кардинально зменшити обсяг переданих даних. Наприклад, у сценаріях вибіркового отримання полів розмір відповіді GraphQL може бути меншим на 94% порівняно з еквівалентним запитом до

REST API [14]. Ієрархічна природа запитів GraphQL дозволяє отримувати дані з кількох пов'язаних ресурсів в рамках одного мережевого циклу "запит-відповідь". Клієнт може сформувати складний запит, який одночасно отримує проект, його типологію та повний склад команди. Це не лише зменшує мережеву затримку (latency), але й кардинально спрощує управління станом на клієнті. Замість оркестрації кількох асинхронних запитів та подальшого об'єднання їх результатів, клієнтський додаток оперує єдиним циклом 'запит-відповідь', що знижує складність коду та ймовірність виникнення помилок синхронізації.

Незважаючи на очевидні переваги GraphQL в оптимізації мережевих запитів, його продуктивність є багатогранним питанням, що вимагає розгляду специфічних викликів, особливо на стороні сервера. Гнучкість, яку GraphQL надає клієнтам, переносить складність на бекенд, де необхідно ефективно обробляти потенційно складні запити. Одним із найяскравіших проявів цієї перенесеної складності є проблема продуктивності 'N+1', яка є прямим наслідком гнучкої природи резолверів GraphQL.

Це одна з найпоширеніших проблем продуктивності в GraphQL, яка виникає при наївній реалізації резолверів. Наприклад, при запиті списку з N авторів та їхніх книг, сервер спочатку виконує один запит до бази даних для отримання списку авторів, а потім для кожного з N авторів виконує окремий запит для отримання списку його книг. Це призводить до N+1 запитів до бази даних, що суттєво погіршує продуктивність. Ця проблема є особливо актуальною для фреймворків, таких як Ruby on Rails, через особливості їхніх ORM [15].

1.4 Проблематика порівняння продуктивності API

Не зважаючи на більш сучасний підхід GraphQL, вибір між цими двома підходами не є однозначним і залежить від багатьох факторів, включаючи архітектуру системи, вимоги до продуктивності та досвід команди. Це вимагає глибокого аналізу та наукових досліджень для визначення їхньої ефективності в конкретних умовах.

Одним із ключових показників продуктивності API є час відповіді (response time), який суттєво залежить від складності запитів. Аналіз наукових праць демонструє чітку залежність ефективності кожного з підходів від типу операції. Для простих запитів до одного ресурсу перевагу має REST, що зумовлено меншими накладними витратами на обробку запиту на стороні сервера. Дослідження, проведене на базі фреймворку NestJS, показало, що при високих навантаженнях (24 000 одночасних користувачів) REST був у середньому на 50% швидшим, ніж повна реалізація GraphQL, і на 36% швидшим, ніж оптимізована версія GraphQL, що повертала лише вибрані поля [14].

Навпаки, GraphQL демонструє суттєву перевагу при виконанні складних запитів, які вимагають агрегації даних з кількох пов'язаних ресурсів [16]. Його ефективність у таких сценаріях полягає в оптимізації мережевої взаємодії: замість множинних послідовних звернень до сервера, характерних для REST, GraphQL дозволяє отримати всі необхідні дані за один мережевий раунд (round trip) [17]. Це підтверджується експериментальними даними, згідно з якими час відповіді GraphQL може бути на 55-61% меншим порівняно з REST у випадках, коли останній вимагає кількох запитів для отримання повного набору даних [14]. Водночас для простих операцій запису даних (CREATE, UPDATE, DELETE) різниця у часі відповіді між двома підходами може бути мінімальною, що робить їх практично рівноцінними для таких завдань [18].

Однією з найвідоміших проблем продуктивності при реалізації GraphQL є проблема "N+1 запиту", розглянута вищє. Ця проблема є особливо актуальною для фреймворків з об'єктно-реляційним відображенням (ORM), таких як Ruby on Rails з його Active Record [19]. Для її вирішення застосовується стратегічний підхід "лінивого завантаження" (lazy-loading), який дозволяє відкласти завантаження пов'язаних даних до моменту, коли вони дійсно потрібні. Для реалізації цього патерну існують спеціалізовані інструменти, наприклад бібліотека graphql-batch, яка оптимізує завантаження даних шляхом їх групування (батчингу) в один запит [20].

Залежність продуктивності від технологічного стека та наявність специфічних проблем, як-от "N+1", вказують на існування дослідницької ніші для вивчення ефективності GraphQL у середовищах, які ще не були комплексно досліджені.

ВИСНОВКИ ПО РОЗДІЛУ 1

Проведений аналіз наукових праць та індустріальних практик показав, що, попри велику кількість досліджень, присвячених порівнянню REST та GraphQL, залишається невирішеним питання продуктивності GraphQL у специфічному, але поширеному середовищі. Це створює прогалину в знаннях, яку дана магістерська робота має на меті заповнити.

Актуальність та необхідність проведення даного дослідження обґрунтовується наступними факторами:

- Виявлена дослідницька ніша: Більшість існуючих порівняльних досліджень продуктивності REST та GraphQL проводяться на стеках Node.js (з використанням NestJS) [14] або Java (з використанням Spring) [21]. Аналіз літератури виявив відсутність комплексних досліджень, що порівнюють ці два підходи в однакових контрольованих умовах на базі популярного фреймворку Ruby on Rails.

- Актуальність проблеми "N+1 запиту": Ruby on Rails, як один з провідних фреймворків для веб-розробки, має вбудовані механізми ORM (Active Record). Проблема "N+1 запиту" стає особливо гострою, коли резолвери для пов'язаних об'єктів наївно реалізуються таким чином, що кожен з них ініціює окремий запит до бази даних для кожного елемента з батьківського списку. Це робить дослідження продуктивності в даному середовищі нетривіальним і практично значущим [19].

- Потреба в практичних рекомендаціях: Розробники, які працюють з Ruby on Rails, потребують чітких, емпірично підтверджених даних про те, який підхід до створення API є більш продуктивним для їхніх конкретних завдань. Результати такого дослідження нададуть їм змогу приймати обґрунтовані архітектурні рішення, обираючи між гнучкістю GraphQL та простотою REST на основі кількісних показників продуктивності.

Таким чином, дана магістерська робота спрямована на заповнення вказаної прогалини шляхом проведення контрольованого експерименту з порівняння продуктивності GraphQL та REST API, реалізованих в рамках єдиного веб-додатку на Ruby on Rails. Це дозволить отримати об'єктивні та релевантні для даного технологічного стека результати, які матимуть як наукову новизну, так і практичну цінність для спільноти розробників.

2 ОБГРУНТУВАННЯ ТА МЕТОДИКА ДОСЛІДЖЕННЯ ПРОДУКТИВНОСТІ REST І GRAPHQL API У ВЕБ-ДОДАТКУ

2.1 Аналіз завдань дослідження та вибір напряму експериментів

У першому розділі було показано, що вибір архітектурного стилю веб-API (REST чи GraphQL) безпосередньо впливає на експлуатаційні характеристики веб-системи: час відгуку, можливості масштабування, навантаження на апаратні ресурси та вартість супроводу. При цьому результати наявних публікацій є неоднорідними, а значна частина робіт або зосереджується на окремих аспектах (наприклад, лише на часі відповіді), або виконується на відмінних технологічних стеках і спрощених синтетичних сценаріях.

Для забезпечення об'єктивності, відтворюваності та наукової обґрунтованості висновків, це дослідження базується на систематичному науковому підході. Обрана методологія дозволяє послідовно пройти шлях від спостереження за проблемою до підтвердження або спростування висунутих гіпотез на основі емпіричних даних, що гарантує надійність отриманих результатів.

В основі експерименту лежить емпіричний цикл дослідження, що складається з п'яти ключових етапів, які забезпечують логічну послідовність та строгість наукового процесу:

- спостереження (observation): збір та аналіз даних про поточний стан проблеми. На цьому етапі було проведено аналіз сучасних підходів та досліджень продусктивності REST і GraphQL API, виявлено проблеми надлишкового завантаження даних та необхідності виконання множинних запитів;

- індукція (induction): формулювання загальної гіпотези на основі зібраних спостережень. На цьому етапі були висунуті припущення, що технологія GraphQL дозволить покращити продуктивність та гнучкість додатку, зменшити навантаження на мережу;

- дедукція (deduction): виведення з загальної гіпотези конкретних, перевірюваних наслідків. Як то, якщо гіпотеза вірна, то реалізація API на GraphQL

повинна показати менший час відгуку та менший розмір відповіді порівняно з REST для комплексних запитів;

- тестування (testing): проведення експерименту для перевірки гіпотези. Цей етап включає порівняльний аналіз між REST API GraphQL за визначеними метриками в однакових умовах;

- оцінка (evaluation): аналіз отриманих експериментальних даних, на основі якого гіпотеза підтверджується або спростовується. На цьому етапі формулюються остаточні висновки дослідження.

У межах даної роботи основна увага приділяється трьом класам сценаріїв, характерним для контент-орієнтованих веб-застосунків: читанню переліку користувачів, читанню переліку публікацій разом з даними автора та створенню нової публікації. Для кожного сценарію реалізовано три варіанти API: REST, GraphQL з повною відповіддю та GraphQL із мінімальною вибіркою полів. Такий дизайн дозволяє, по-перше, порівняти два стилі API за однакової інформаційної насиченості відповіді, а по-друге, проаналізувати окремий внесок дисципліни вибірки даних у загальну продуктивність GraphQL.

На основі теоретичного аналізу припускається, що для плоских вибірок REST демонструватиме меншу затримку, тоді як у складно зв’язаних запитах GraphQL із мінімальною вибіркою переважатиме за рахунок усунення overfetch. Остаточний висновок ухвалюватиметься шляхом зіставлення часових, ресурсних та мережевих показників із урахуванням статистичної значущості різниці.

2.2 Система показників ефективності та критерії порівняння

Обґрунтування експериментального методу неможливе без чіткого визначення системи показників, за якими оцінюється «ефективність» того чи іншого підходу. У загальному випадку продуктивність веб-API описується не однією, а сукупністю взаємопов’язаних метрик, які відображають як якість сервісу для кінцевого користувача, так і вартість забезпечення цього сервісу на стороні сервера.

Найбільш безпосередньо на сприйняття користувачем впливає час відгуку запиту. Через стохастичний характер навантаження і мережевих затримок

використання лише середнього арифметичного може бути оманливим, оскільки не відображає поведінку так званих «хвостів» розподілу. Тому для кожного сценарію аналізуються як медіанне значення часу відповіді, яке характеризує типову затримку, так і високі перцентилі (зокрема 90-й), що відображають поведінку системи в умовах пікового навантаження. Додатково фіксується максимальне значення, яке є індикатором поодиноких, але потенційно критичних провалів у продуктивності.

Другу групу становлять ресурсні показники. Завантаження центрального процесора дозволяє оцінити, наскільки обчислювально «важкими» є різні стилі API для сервера додатка, особливо з огляду на додаткові рівні обробки запиту в GraphQL (парсинг, валідація, побудова плану виконання, виклик резолверів). Споживання оперативної пам’яті характеризує можливість масштабування системи в ширину: чим більший об’єм пам’яті потребує окремий процес сервера, тим менше екземплярів можна розгорнути в рамках одного фізичного чи віртуального вузла.

Третю групу утворюють мережеві показники, які відображають ефективність використання каналу зв’язку між клієнтом і сервером. Для кожного тесту оцінюються середні обсяги переданих та отриманих даних. Саме в цьому вимірі GraphQL традиційно декларує свою перевагу за рахунок уникнення надлишкового отримання даних. У той же час REST у типовій реалізації має фіксовані структури відповідей, що потенційно призводить до передачі зайвої інформації.

Порівняння REST і GraphQL проводиться за принципом «за інших рівних умов». Це означає, що обидва підходи працюють над однаковою предметною областю та базою даних, використовують ідентичну апаратну платформу й однаковий профіль навантаження. У сценаріях читання REST-відповідь порівнюється переважно з повною проєкцією GraphQL, тоді як мінімальна проєкція розглядається як окремий дослідницький випадок, який дозволяє кількісно оцінити вплив дисципліни вибірки полів на всі вказані вище групи метрик.

2.3 Обґрунтування експериментальної моделі предметної області та API

Щоб перевірити робочі гіпотези щодо порівняльної продуктивності REST- та GraphQL-підходів, необхідно створити експериментальні умови, які, з одного боку,

будуть достатньо простими для формального контролю, а з іншого – репрезентуватимуть типову задачу сучасних веб-систем. Аналіз практичних кейсів показав, що характерною рисою більшості інформаційних сервісів є наявність ієрархічних зв'язків «користувач → публікації → коментарі». Така структура поєднує дві принципові ситуації:

- плоскі вибірки (перелік користувачів) – критичні для перевірки базових накладних витрат;
- агреговані запити (публікації разом з авторами та вкладеними об'єктами) – показові щодо проблеми overfetch/underfetch.

Отже, у якості об'єкта дослідження доцільно використати спрощену модель системи управління контентом, що містить три сутності: користувачі, публікації, коментарі. Таке моделювання забезпечує баланс між аналітичною керованістю та прикладною значущістю результатів.

Для усунення стороннього впливу та підвищення валідності експерименту виділено дві групи змінних:

- архітектурний фактор – стиль API (REST, GraphQL-full, GraphQL-min), що є основною незалежною змінною;
- структурний фактор – тип запиту (плоский, ієрархічний, модифікувальний), який дозволяє перевірити робочі гіпотези у різних предметних контекстах.

Контрольними змінними виступають: програмно-апаратне середовище, обсяг і структура даних, схема БД, механізми кешування та профіль навантаження. Фіксація цих параметрів гарантує, що зафіксована різниця у метриках є наслідком саме архітектурного вибору, а не побічних чинників.

REST- та GraphQL-реалізації мають віддзеркалювати одну й ту саму предметну модель, забезпечуючи логічну еквівалентність переданого змісту. Для REST це досягається шляхом визначення ресурсів, що відповідають сутностям моделі, та використанням конвенцій HTTP. Для GraphQL, навпаки, формується єдина схематична точка доступу, у межах якої описуються типи та їх взаємозв'язки.

Щоб зняти ефект «N+1 запиту» й інших ортогональних оптимізацій, як у REST-контролерах, так і в GraphQL-резолверах закладається однаковий рівень

попереднього завантаження зв’язаних даних. Таким чином, експеримент вимірює саме вартість обраного протоколу взаємодії, а не якість конкретної імплементації.

Обсяг і структура тестових даних підбираються таким чином, щоб, з одного боку, виключити повні сканування таблиць і, з іншого, не вводити штучну деградацію, пов’язану з браком індексів або надлишковим розміром БД. Для цього використано нормативні правила нормалізації та створено індекси на зовнішніх ключах і полях сортування. Усі варіанти API працюють із тотожною БД, що унеможливлює спотворення порівняння через різні плани виконання SQL.

2.4 Методика проведення експериментів

2.4.1 Сценарії та режими навантаження

Методика проведення експериментів побудована таким чином, щоб, з одного боку, відтворювати типові профілі навантаження на веб-додаток, а з іншого – забезпечувати статистично достатній обсяг даних для аналізу. Для всіх сценаріїв використано єдиний профіль: десять віртуальних користувачів одночасно виконують запити протягом тридцяти секунд, між окремими ітераціями передбачається невелика пауза, яка імітує користувацькі дії в інтерфейсі. Такий режим дозволяє сформувати стабільний потік запитів без штучного перевантаження сервера.

Кожен тестовий прогін складається з двох послідовних запусків одного й того самого сценарію. Перший запуск виконує роль фази розігріву: за цей час ініціалізуються з’єднання з базою даних, заповнюються кеші, завантажується необхідний код, стабілізуються механізми JIT-компіляції[22] та збирача сміття. Другий запуск розглядається як основний експериментальний прогін, результати якого підлягають подальшому аналізу.

2.4.2 Процедура вимірювання та збір даних

Методика збору емпіричного матеріалу передбачає поетапне підвищення складності: спершу оцінюються плоскі запити, далі – ієрархічні, і лише після цього – запити на модифікацію. Така послідовність дозволяє розмежувати базові накладні витрати протоколів і специфічні накладні, що проявляються лише у складних сценаріях.

Безпосереднє зняття показників здійснюється в автоматизованому режимі. Перед початком кожного тесту запускається сервер Rails, очікуючи на його готовність до обробки HTTP-запитів. Після цього у фоновому режимі активується моніторинг за допомогою pidstat[23], який з фіксованою періодичністю вимірює частку часу процесора, зайняту процесом сервера, та обсяг використаної ним оперативної пам’яті. На наступному етапі виконується навантажувальний сценарій k6[24], який фіксує для кожного запиту час відгуку та сумарні обсяги переданих і отриманих даних.

Після завершення навантаження всі допоміжні процеси коректно зупиняються, а зібрані журнали перетворюються у табличну форму.

2.4.3 Статистична обробка результатів

Зібрані дані обробляються у декілька послідовних кроків. Спочатку з результатів k6 для кожного сценарію та кожного варіанта API обчислюються статистичні характеристики часу відгуку: медіанне значення, 90-й перцентиль і максимальне значення. Також визначаються середні обсяги даних, що передаються та приймаються за одиницю часу. Інструмент тестування навантаження k6 дозволяє порахувати ці данні автоматично. На основі вихідних CSV-файлів із pidstat обчислюються середні значення завантаження процесора і використання оперативної пам’яті.

Такий підхід дозволяє зробити порівняння більш наочним і менш чутливим до змін абсолютних величин, що можуть виникати при незначних варіаціях середовища чи фонових процесів. Особлива увага приділяється узгодженості результатів між повторними запусками та відсутності аномалій, що могли б свідчити про збої в роботі інфраструктури.

2.5 Аналіз обмежень та загроз валідності

Будь-яке емпіричне дослідження має враховувати обмеження, що можуть впливати на інтерпретацію результатів. У даному випадку передусім слід відзначити використання синтетичного навантаження з фіксованою кількістю віртуальних користувачів і сталою інтенсивністю запитів. Реальні виробничі системи, як правило,

демонструють змінні профілі трафіку з піками та провалами, що може по-іншому впливати на механізми масштабування та кешування.

Другою важливою особливістю є розгортання сервера застосунку та інструментів навантаження. Відмінності центральних процесорів, підсистем пам’яті, систем віртуалізації, тощо можуть впливати на швидкодію систем та результати вимірювань. Усе це означає, що отримані абсолютні значення часу відгуку і ресурсних витрат не слід безпосередньо переносити на інші конфігурації без додаткової валідації.

Нарешті, результати цієї роботи безпосередньо стосуються стеку Ruby on Rails 8 у поєднанні з PostgreSQL і graphql-ruby. В інших мовах програмування та фреймворках (Node.js, Java, Go тощо) як абсолютні, так і відносні показники можуть змінюватися через інші моделі виконання, реалізації ORM та оптимізації середовищ виконання. Водночас логіка впливу надлишкового та недостатнього отримання даних, а також дисципліни вибірки полів загалом зберігається, що дозволяє очікувати подібних якісних тенденцій.

ВИСНОВКИ ДО РОЗДІЛУ 2

У цьому розділі обґрунтовано вибір експериментального підходу як основного методу дослідження продуктивності REST і GraphQL у контексті реалістичного веб-додатка. Сформовано систему показників, яка включає часові, ресурсні та мережеві характеристики, визначено критерії коректного порівняння двох стилів API за умов єдиної предметної області та ідентичного профілю навантаження. Детально описано програмно-апаратне середовище, модель даних, варіанти реалізації REST- та GraphQL-інтерфейсів, а також методику проведення експериментів і статистичної обробки результатів з урахуванням обмежень та загроз валідності.

Таким чином, розділ формує концептуальну основу майбутнього експерименту, визначаючи логіку вибору моделі, керовані змінні, контрольні параметри та правила інтерпретації результатів.

3 ПРОЕКТУВАННЯ ТА РОЗРОБКА ІНСТРУМЕНТАЛЬНОГО ЗАБЕЗПЕЧЕННЯ ДЛЯ ДОСЛІДЖЕННЯ ПРОДУКТИВНОСТІ GRAPHQL У ВЕБ-ДОДАТКАХ

3.1 Формалізація задачі та зовнішнє проектування

Мета розробки – створити відтворюваний програмний стенд для дослідження продуктивності GraphQL у веб-додатках та порівняння цього підходу з традиційним REST API в однакових умовах. Відтворюваність забезпечується тим, що:

- обидва API-підходи реалізовані в одній кодовій базі;
- GraphQL та REST працюють над спільною доменною моделлю Active Record;
- використовується спільне сховище даних PostgreSQL;
- сценарії навантажувального тестування та збір метрик автоматизовані.

Для коректного порівняння реалізуються дві паралельні «проекції» над одними й тими ж сутностями: GraphQL дозволяє клієнту визначати набір полів у відповіді, а REST повертає фіксовані структури, типові для ресурсного підходу. Це створює умови для дослідження двох характерних аспектів продуктивності GraphQL, таких як вплив вкладених вибірок на кількість SQL-запитів (проблема N+1) та вплив «over-fetching/under-fetching» на час обробки запитів та використання ресурсів.

Як предметну область обрано спрощену модель блогу, що є репрезентативною для вкладених структур і водночас достатньо простою для інтерпретації експериментів. Домен включає три базові сутності: користувачі (`User`), пости (`Post`) і коментарі (`Comment`) з типовими зв’язками 1-до-багатьох.

3.2 Базова архітектура системи

Для реалізації стенда обрано архітектурний підхід Model–View–Controller (MVC), який є базовим для Ruby on Rails. В межах MVC:

- модель (Model) відповідає за доступ до даних і доменну логіку (Active Record моделі `User`, `Post`, `Comment`);

- контролер (Controller) відповідає за обробку HTTP-запитів і формування відповіді (GraphQL контролер `GraphqlController` та REST-контролери `Api::V1::*`);

- вигляд (View) у цьому проєкті має допоміжний характер і використовується переважно через GraphiQL як інструмент ручної перевірки GraphQL-схеми у development-середовищі.

Архітектурно система складається з таких контурів:

- GraphQL API: єдиний ендпоінт `/graphql` (контролер `GraphqlController`), схема `GraphqlTestSchema` з типами та мутаціями (`app/graphql/*`);
- REST API: версіоновані ендпоінти `/api/v1/*`, реалізовані контролерами `Api::V1::UsersController`, `Api::V1::PostsController`, `Api::V1::CommentsController`;
- сховище даних: PostgreSQL (таблиці `users`, `posts`, `comments`);
- інструментальний контур: сценарії k6 (`k6-tests/*.js`) та скрипт `test.sh`, який автоматизує запуск сервера, моніторинг і формування файлу з результатами.

В якості технологічної платформи для реалізації прототипу обрано фреймворк Ruby on Rails та систему управління базами даних PostgreSQL. Rails забезпечує високу швидкість розробки завдяки усталеним конвенціям, багатому набору інструментів для роботи з моделями даних і вбудованим механізмам міграцій і наповнення бази, що дозволяє створити єдину кодову базу з реалізаціями REST і GraphQL з мінімальною додатковою інфраструктурою. PostgreSQL було вибрано як надійну та функціонально багату СУБД, яка підтримує UUID-ідентифікатори, індекси, типи JSONB і розширення для моніторингу; ці можливості роблять її придатною для моделювання реалістичних навантажень і оцінки впливу доступу до БД на продуктивність API. Поєднання Rails і PostgreSQL забезпечує баланс між зручністю реалізації, достовірністю емпіричних вимірювань і відтворюваністю експериментів.

Мову Ruby обрано з огляду на наявність зрілої екосистеми для веб-розробки та API, підтримку GraphQL у вигляді бібліотеки graphql-ruby, можливість швидкого проектування доменної моделі.

Використаний технологічний стек:

- Ruby 3.4.2 – мова програмування.
- Ruby on Rails 8.0.3 – MVC-каркас, міграції, маршрутизація, обробка запитів.

- PostgreSQL – реляційна база даних.
- `graphql` (~> 2.0) – реалізація GraphQL (типи, мутації, валідація, виконання запитів).
- `GraphQL::Dataloader` – механізм батчингу асоціацій для мінімізації N+1 (використовується в схемі та типах).
- `graphiql-rails` – GraphiQL інтерфейс для development (`/graphiql`).
- Puma – HTTP-сервер Rails-застосунку.
- RSpec (`rspec-rails`) – модульні та інтеграційні тести.
- k6 – навантажувальні тести (файли `k6-tests/*.js`).
- `pidstat` + `awk` – збір та агрегація метрик CPU/RAM у `test.sh`.

3.3 Логічна модель даних

Логічна модель даних побудована навколо сутностей блогу й призначена для дослідження поведінки GraphQL у вкладених вибірках. База даних побудована за класичною моделлю «користувач – пости – коментарі» та містить три основні таблиці: Users, Posts і Comments (дивись рисунок 3.1).

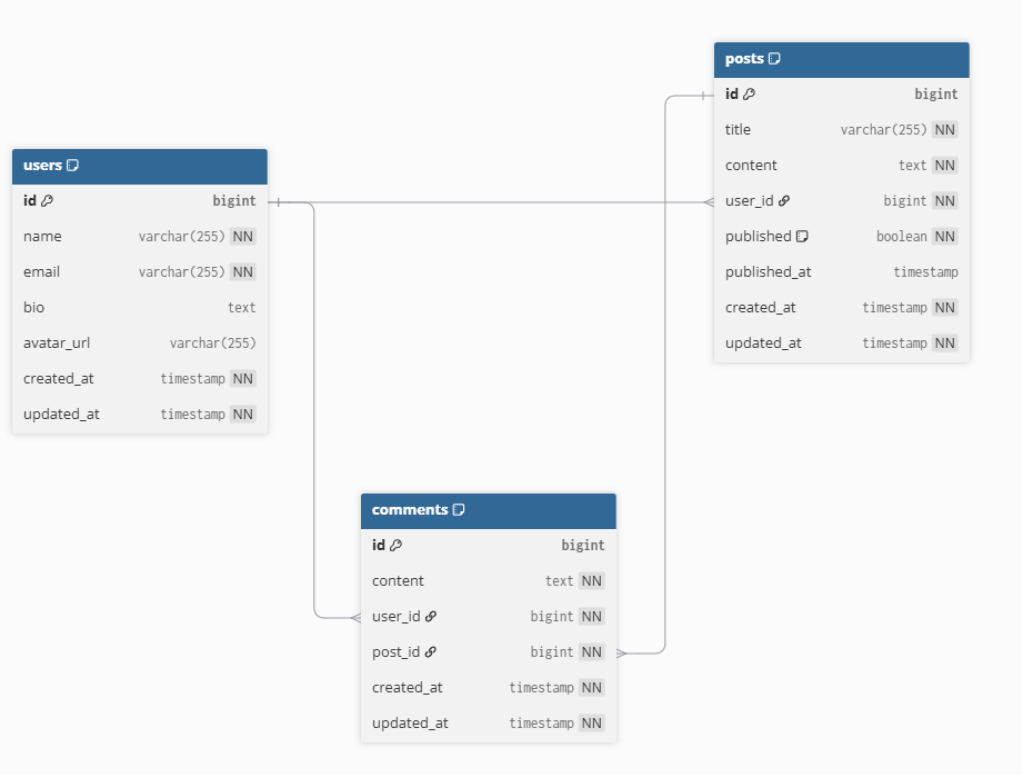

Рисунок 3.1 – Структура бази даних

Таблиця Users зберігає інформацію про користувачів системи. Кожен запис містить ідентифікатор (первинний ключ), ім’я, унікальну електронну адресу та додаткові атрибути (наприклад, біографію чи аватар). Користувач може створювати пости та залишати коментарі.

Таблиця Posts призначена для збереження публікацій користувачів. Кожен пост має унікальний ідентифікатор, заголовок, текстовий вміст, ознаку опублікованого стану, а також посилання на автора через зовнішній ключ user_id. Крім цього, пост може мати дату публікації та пов’язується з багатьма коментарями.

Таблиця Comments використовується для збереження коментарів до постів. Коментар має унікальний ідентифікатор, текстовий вміст та два зовнішні ключі: user_id (ідентифікує користувача, який залишив коментар) і post_id (ідентифікує пост, до якого цей коментар належить). Таким чином, кожен коментар належить як одному посту, так і одному користувачеві.

Зв’язки між таблицями реалізують ієрархічну структуру: один користувач може створити багато постів, один пост може містити багато коментарів, а один користувач може залишити багато коментарів до різних постів. Обрана модель є типовою для веб-додатків, що реалізують функціональність блогу чи соціальної мережі, і забезпечує простоту виконання запитів та гнучкість у подальшому масштабуванні.

Вибір саме трьох сутностей – користувачі, пости та коментарі – обумовлений кількома факторами:

- простота та логічність моделі: обрана структура відображає природні взаємозв’язки у системі: користувач створює пости, а інші користувачі можуть залишати до них коментарі. Така модель легко інтерпретується та відповідає предметній області;
- нормалізація даних: винесення постів і коментарів у окремі таблиці усуває дублювання інформації, спрощує модифікацію даних та забезпечує узгодженість;
- масштабованість: збереження чітких зв’язків «один-до-багатьох» між сутностями (користувач - пости, пост - коментарі) дозволяє ефективно

працювати як із невеликими, так і з великими обсягами даних. При зростанні кількості користувачів чи постів модель залишається стійкою;

- розширюваність: така структура легко доповнюється новими сутностями (наприклад, «лайки», «теги», «категорії»), не порушуючи базової логіки взаємозв'язків. Це дозволяє розвивати систему поступово, без радикальної перебудови бази;
- універсальність для веб-застосунків: обрана модель є типовою та перевіреною практикою у веб-розробці. Вона дозволяє реалізувати як базові можливості блогу, так і більш складні сценарії взаємодії користувачів.

Для відтворюваності експериментів реалізовано генерацію тестових даних у `db/seeds.rb`: створюється набір користувачів, для кожного – випадкова кількість постів (частина з них публікується), а також випадкова кількість коментарів для опублікованих постів. Це дозволяє швидко отримати датасет з реалістичною кількістю зв'язків для тестування вкладених GraphQL-запитів.

3.4 Структура API

Розроблена програма для дослідження продуктивності GraphQL реалізує дві паралельні архітектури API для порівняння їх ефективності. Програма працює з доменною моделлю блогу, що включає три основні сутності: користувачі (Users), пости (Posts) та коментарі (Comments).

3.4.1 Структура GraphQL API

GraphQL API організовано за принципами схеми GraphQL з використанням gem graphql-ruby. Основні компоненти розташовані в директорії app/graphql/ (див. рис. 3.2).

```
app/graphql/
├── graphql_test_schema.rb      # Головна схема GraphQL
├── types/                      # Типи даних
│   ├── base_object.rb          # Базовий клас для всіх типів
│   ├── query_type.rb           # Кореневий тип для запитів
│   ├── mutation_type.rb        # Кореневий тип для мутацій
```

```
│   ├── user_type.rb          # Тип користувача
│   ├── post_type.rb          # Тип поста
│   └── comment_type.rb       # Тип коментаря
├── mutations/                # Мутації для зміни даних
│   ├── base_mutation.rb      # Базовий клас для мутацій
│   ├── create_user.rb        # Створення користувача
│   ├── create_post.rb        # Створення поста
│   └── create_comment.rb     # Створення коментаря
└── resolvers/                # Резолвери (якщо потрібні)
```

Рисунок 3.2 - Головна схема (GraphqlTestSchema)

Центральним елементом є клас GraphqlTestSchema, який визначає кореневі типи для запитів (QueryType) та мутацій (MutationType), встановлює обмеження на розмір запитів (5000 токенів) та описує основні типи даних (Types). UserType представляє користувача, PostType представляє публікації, а CommentType представляє коментарі.

Запити (Queries) використовуються для отримання інформації та представлені наступним:

- users - отримання всіх користувачів;
- user(id: ID!) - отримання користувача за ID;
- user_with_posts_and_comments(id: ID!) - користувач з усіма пов'язаними даними;
- posts(published_only: Boolean, limit: Integer, order_by: String) - отримання постів з фільтрацією;
- post(id: ID!) - отримання поста за ID;
- popular_posts(limit: Integer) - популярні пости;
- recent_posts(limit: Integer, published_only: Boolean) - останні пости;
- comments(post_id: ID, user_id: ID, limit: Integer) - отримання коментарів з фільтрацією;
- comment(id: ID!) - отримання коментаря за ID.

Мутації (Mutations) надають можливості для зміни даних та представлені наступним:

- CreateUser(name!, email!, bio, avatar_url);
- CreatePost(title!, content!, user_id!, published);
- CreateComment(content!, user_id!, post_id!, parent_id).

3.4.2 Структура REST API

REST API організовано за принципами RESTful архітектури з використанням Rails conventions. API версіоновано (v1) та розташовано в namespace Api::V1(див. рис. 3.3)

```
app/controllers/api/v1/
├── base_controller.rb          # Базовий контролер для API
├── users_controller.rb         # Контролер користувачів
├── posts_controller.rb         # Контролер постів
└── comments_controller.rb      # Контролер коментарів
```

Рисунок 3.3 Базовий контролер (BaseController)

Api::V1::BaseController надає базову спільну функціональність, яка включає управління CSRF захистом, встановлення формату за замовчуванням та надає методи для стандартизованих відповідей (render_success, render_error).

REST API використовує стандартні RESTful маршрути з додатковими ендпоінтами та наступними контролерами ресурсів:

- UsersController - реалізує повний CRUD для користувачів;
- PostsController - реалізує повний CRUD для постів;
- CommentsController - реалізує повний CRUD для коментарів.

Кожен контролер містить методи серіалізації для перетворення моделей у JSON, а також перевірки переданих параметрів, яка відбувається спільно з рівніем моделей, а помилки повертаються через стандартизовані методи render_validation_errors.

Така архітектура дозволяє проводити об'єктивне порівняння продуктивності двох підходів в ідентичних умовах.

3.5 Використані принципи та шаблони проектування

Під час проектування внутрішньої структури застосовано такі принципи та підходи:

- MVC як базова архітектурна модель Rails з розмежуванням відповідальностей між моделями, контролерами та GraphiQL API;
- SOLID: винесення повторюваної логіки у базові класи GraphQL (Types::BaseObject, Types::BaseField, Mutations::BaseMutation), локалізація доменної логіки у моделях, мінімізація дублювання;
- Патерн «Команда» для мутацій GraphQL: кожна мутація інкапсулює операцію створення даних і повертає структурований результат (об’єкт + помилки);
- Патерн «Data Loader»: GraphQL::Dataloader та Sources::ActiveRecordAssociation використовуються для батчингу та усунення N+1.

3.6 Тестування та налагодження

Функціональна коректність підтверджується автоматизованими тестами RSpec:

- інтеграційні тести GraphQL-запитів розміщено у `spec/graphql/queries/*` і виконують HTTP-виклики до `/graphql` з перевіркою структури відповіді, фільтрації та лімітів;
- тести контролерів (controller-spec) для REST API розміщено у `spec/controllers/api/v1/*` і покривають CRUD-операції, помилки валідації, додаткові ендпоінти та уніфікований формат JSON-відповіді.

Застосовано поєднання методів «чорної скриньки» (перевірка контрактів/статусів) та елементів «білої скриньки» (перевірка гілок логіки фільтрації й серіалізації).

ВИСНОВКИ ДО РОЗДІЛУ 3

Розроблено інструментальний стенд для дослідження продуктивності GraphQL у веб-додатках на базі Ruby on Rails та PostgreSQL. Реалізовано два паралельні API-контури (GraphQL і REST) над спільною доменною моделлю блогу (Users/Posts/Comments), що забезпечує порівнянність результатів. Для мінімізації N+1

у GraphQL інтегровано `GraphQL::Dataloader` та власне джерело батчингу асоціацій. Реалізований набір RSpec-тестів підтверджує функціональну коректність API. Отримане рішення є відтворюваною основою для проведення експериментальних досліджень продуктивності, які розглядаються у наступному розділі.

4 ДОСЛІДЖЕННЯ ПРОДУКТИВНОСТІ GRAPHQL У ВЕБ-ДОДАТКАХ

4.1 Підготовка до експерименту

У розділі наведено хід та результати експериментального дослідження продуктивності GraphQL у порівнянні з традиційним REST API на інструментальному стенді, розробленому у розділі 3. Дослідження спрямоване на емпіричну оцінку впливу типу API та обсягу вибірки полів у GraphQL на часові характеристики відповіді, мережевий трафік, а також ресурсне навантаження сервера (ЦП та оперативна пам’ять).

Для досягнення мети було сформульовано такі задачі дослідження:

- визначити репрезентативні сценарії взаємодії клієнта із веб-додатком, що охоплюють операції читання «плоских» даних, читання вкладених (ієрархічних) даних та операції запису;
- реалізувати для кожного сценарію три функціонально порівнянні варіанти доступу до даних: REST, GraphQL із повним набором полів (GraphQL(full)) та GraphQL із мінімально необхідним набором полів (GraphQL(min));
- виконати навантажувальні запуски за єдиним профілем та отримати порівнювані метрики затримки, мережевої активності та використання ресурсів;
- проаналізувати отримані результати, оцінити достовірність висновків, порівняти їх з відомими спостереженнями щодо GraphQL/REST і визначити напрями подальших досліджень.

4.1.1 Опис використаного програмно-апаратного середовища

Експерименти виконувалися у контрольованому середовищі з локальним розгортанням сервера застосунку та навантажувального генератора на одній машині (виклики здійснювалися до `http://localhost:3000`). Це дозволяє мінімізувати вплив зовнішніх мережевих факторів і сфокусувати вимірювання на серверних накладних витратах обробки REST та GraphQL.

Апаратні характеристики середовища:

- процесор: AMD Ryzen 7 7800X3D (8 фізичних ядер / 16 потоків);
- оперативна пам’ять: 32 GiB RAM.

Програмні характеристики середовища:

- ОС: Linux 6.6.87.2-microsoft-standard-WSL2 (x86_64);
- Ruby: 3.4.2 (через RVM gemset `graphql_test`);
- Ruby on Rails: 8.0.3;
- бібліотека GraphQL: gem `graphql` 2.5.13;
- СУБД: PostgreSQL 17.4;
- інструмент навантаження: k6;
- інструмент моніторингу: sysstat/pidstat 12.7.8.

4.1.2 Мінімізація впливу «холодного старту» та побічних факторів на результати вимірювань

Під час вимірювань продуктивності суттєвий вплив мають ефекти «холодного старту»: завантаження класів та залежностей, ініціалізація пулів з’єднань, заповнення кешів ORM/СУБД, а також наповнення сторінкових кешів ОС. Для нівелювання цих факторів кожний тест виконувався двічі поспіль. Перший запуск розглядався як прогрів і у фінальний аналіз не включався, другий запуск вважався звітним і використовувався для побудови таблиць та висновків. Такий підхід підвищує відтворюваність, оскільки зіставлення REST та GraphQL відбувається у більш стабільному режимі роботи застосунку.

На результат вимірювань також можуть впливати побічні флуктуації навантаження ОС (перемикання контекстів, фонові процеси), поведінка PostgreSQL і планувальника запитів. Для мінімізації таких ефектів застосовано однаковий профіль навантаження k6 для всіх порівнюваних реалізацій, а усі запуски виконувалися локально на одній машині, що зменшує вплив зовнішніх мережевих затримок.

4.1.3 Використання інструментів вимірювання для визначення часу відповіді та супутніх метрик

Часова ефективність оцінювалася за метрикою k6 [24] `http_req_duration`, яка відображає тривалість виконання HTTP-запиту на клієнтській стороні (локальний

виклик до сервера). Для підвищення інформативності аналізувалися не лише середні значення, а й квантилі розподілу затримок: медіана (med) як показник «типової» затримки, 90-й перцентиль (p90) як характеристика «хвоста» розподілу, максимальне значення (max) як індикатор поодиноких піків.

Мережеві характеристики визначалися за k6-метриками `data_received` і `data_sent`, нормованими у перерахунку на КБ/с. Оскільки GraphQL передає структуру запиту (selection set) у тілі HTTP-запиту, очікуваним є збільшення `data_sent` для GraphQL у порівнянні з REST, тоді як `data_received` залежить від обсягу відповіді (кількості полів, що матеріалізуються у результаті).

Ресурсні характеристики збиралися утилітою `pidstat` [23] з частотою 1 Гц для процесу Rails-сервера. Для пам’яті використовувався показник RSS (КБ). Для CPU у скрипті збору даних фіксувалася частка процесорного часу у user-space (%usr), яка є репрезентативною для оцінки накладних витрат виконання коду застосунку. Для подальшого порівняння обчислювалися середні значення CPU та RSS на інтервалі вимірювання.

Для запуску та оркестрування було створено окремий скрипт `test.sh`, який реалізує відтворюваний протокол експерименту:

- запускає Rails-сервер у фоні (`rails s`) та очікує готовності через перевірку HTTP-відповіді;
- вмикає моніторинг `pidstat -p <PID> -u -r 1` із частотою 1 Гц та записує лог;
- запускає навантажувальний сценарій `k6 run <k6-script>`;
- по завершенні коректно зупиняє процеси сервера та моніторингу;
- агрегує дані `pidstat` у CSV `performance_results.csv` з колонками `Elapsed_Time(s),CPU_Usage(%),RAM_Usage(KB)`.

В кожному k6-скрипті задано однакові параметри та проводиться перевірка успішності статус-кодів. test.sh приймає шлях до k6-скрипту і запускає його в уніфікованому середовищі, що забезпечує єдині умови збору метрик ЦП/ОП.

4.1.4 Вибір кінцевих точок API та сценаріїв для експериментів

У межах дослідження сформовано парні сценарії для REST та GraphQL, що відображають типові операції читання колекцій і створення ресурсу. Для GraphQL кожен сценарій має дві версії – “full” (повний набір полів) і “min” (мінімально достатні поля), що дозволяє експериментально оцінити вплив overfetch/underfetch на час відгуку, навантаження ЦП/ОП і мережеві витрати. Усі тести автоматизовані окремими k6-скриптами й оркестровано скриптом test.sh; кожен сценарій запускався двічі поспіль (перший запуск – розігрів, другий – звітний).

Параметри навантаження у k6-скриптах сталі для всіх сценаріїв: vus: 10, duration: '30s' (із sleep(1) між ітераціями), що забезпечує порівнюваність замірів.

У контрольних сценаріях REST досліджувалися три операції. По-перше, отримання переліку користувачів реалізовано запитом GET до шляху /api/v1/users; запит не містить тіла, використовуються стандартні заголовки HTTP/1.1 без авторизації, а коректність обслуговування фіксується кодом відповіді 200 OK. По-друге, отримання переліку публікацій із даними автора виконується запитом GET до /api/v1/posts; критерієм успіху є 200 OK. По-третє, створення публікації здійснюється запитом POST до /api/v1/posts із MIME-типом application/json; тіло запиту відповідає стилю strong parameters у Rails і включає вкладений об’єкт post із полями заголовка, змісту, ідентифікатора автора та ознаки публікації.

Аналогічні функціональні сценарії були сформульовані для GraphQL з двома варіантами кожного запиту – повним (далі GraphQL full або просто GraphQL) і мінімальним (далі GraphQL min), що дозволяє оцінити вплив обсягу вибірки даних на продуктивність. Для переліку користувачів у версії full повертається детальний профіль (ідентифікатор, ім’я, електронна адреса, біографія, аватар, похідні та службові поля), тоді як у версії min обмежено лише ідентифікатором та ім’ям. Для переліку публікацій із даними автора запит у версії full містить повний набір полів публікації разом із вкладеним блоком автора, тоді як версія min повертає лише ідентифікатор і заголовок публікації. Для сценарію створення публікації застосовано мутацію з передачею змінних; у варіанті full у відповідь повертається повний набір полів новоствореного запису, а у варіанті min – лише його ідентифікатор.

Для відтворюваності експериментів кожен сценарій було однозначно прив’язано до окремого k6-скрипта. Сценарій «Користувачі» реалізовано трьома варіантами: REST через k6-tests/rest_users_full.js, GraphQL (повний набір полів) через k6-tests/graphql_users_full.js та GraphQL (мінімальний набір полів) через k6-tests/graphql_users_id_name.js. Сценарій «Публікації з авторами» також має три відповідники: REST через k6-tests/rest_posts.js, GraphQL(full) через k6-tests/graphql_posts_full.js, GraphQL(min) через k6-tests/graphql_posts_min.js. Створення публікації реалізовано як REST POST у k6-tests/rest_create_post.js, GraphQL-мутація з повним набором повернених полів у k6-tests/graphql_create_post.js та мінімальним – у k6-tests/graphql_create_post_min.js.

Усі запуски координувалися скриптом test.sh, який автоматично піднімав rails s, перевіряв готовність сервера, вмикав моніторинг pidstat (ЦП та RSS з частотою 1 Гц), виконував відповідний k6-скрипт, коректно завершував усі процеси та формував CSV із часовими рядами метрик. Щоб нівелювати вплив холодного старту застосунку, ORM і кешів, кожен тест виконувався двічі підряд; для аналізу використовувалися результати другого прогона. Критерії коректності відповідей перевірялися у k6: для запитів читання (GET) – статус 200 OK; для REST-створення – статус 201 Created і наявність {"success": true} у тілі відповіді; для GraphQL – статус 200 та відсутність помилок як на верхньому рівні (errors), так і в доменно-специфічних полях відповіді мутації.

4.2 Проведення експерименту

4.2.1 Експеримент 1 - перелік користувачів

У сценарії «перелік користувачів» послідовно виконувалися дві серії для кожного з трьох тестів: rest_users_full, graphql_users_full та graphql_users_id_name (мінімальна виборка). На кожній ітерації скрипт test.sh приймав відповідний k6-файл як вхідний параметр, забезпечував стабільний стан сервера та збір системних метрик, після чого фіксував зведення k6 і результати моніторингу (дивись рис. 4.1). Для аналізу використовувалися лише дані другого запуску в межах кожного тесту.

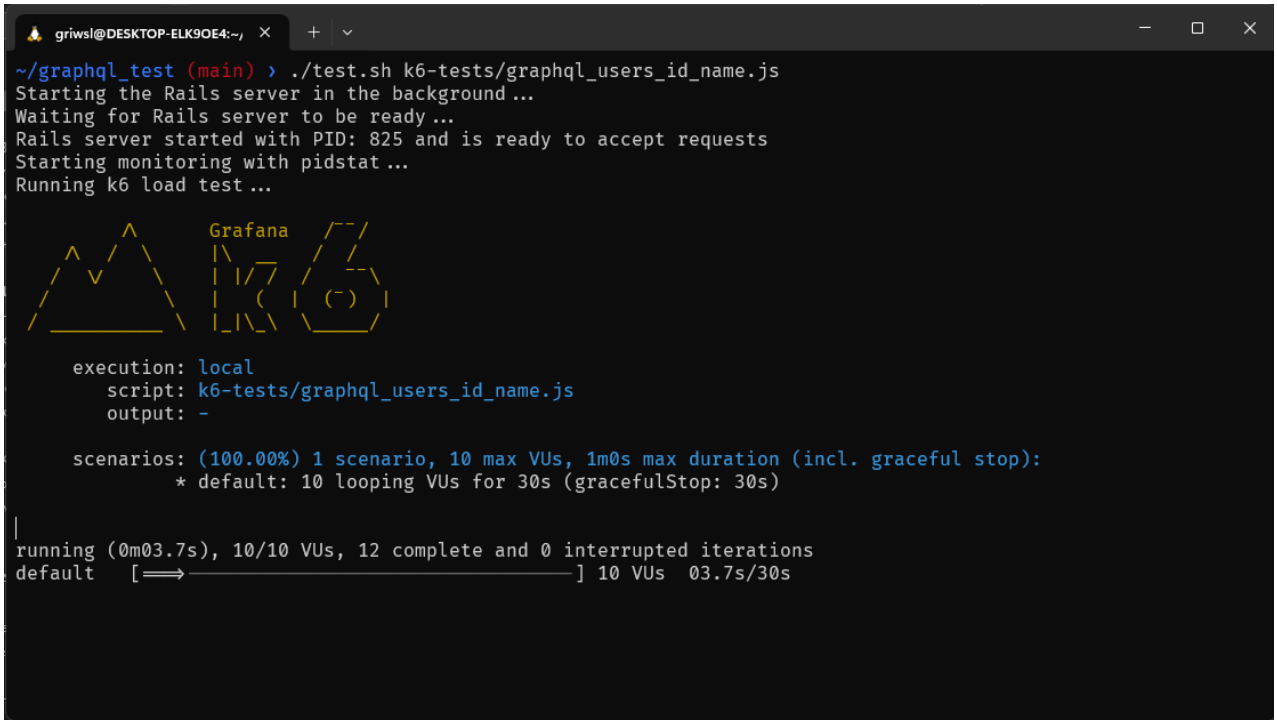

Рисунок 4.1 – процес проведення експерименту

Результати вимірювань REST API за допомогою k6 наступні:

checks_total.....: 175 5.563779/s

checks_succeeded...: 100.00% 175 out of 175

checks_failed.....: 0.00% 0 out of 175

✓ status was 200

HTTP

http_req_duration.....: avg=768.8ms min=540.33ms med=590.47ms max=2.26s p(90)=1.07s p(95)=1.16s

{ expected_response:true }...: avg=768.8ms min=540.33ms med=590.47ms max=2.26s p(90)=1.07s p(95)=1.16s

http_req_failed.....: 0.00% 0 out of 175

http_reqs.....: 175 5.563779/s

EXECUTION

iteration_duration.....: avg=1.76s min=1.54s med=1.59s max=3.26s p(90)=2.07s p(95)=2.16s

iterations.....: 175 5.563779/s

vus.....: 4 min=4 max=10

vus_max.....: 10 min=10 max=10

NETWORK

data_received.....: 681 kB 22 kB/s

data_sent.....: 14 kB 456 B/s

Виміри завантаженості процесора та оперативної пам’яті зведено у таблиці 4.1

Таблиця 4.1 - Завантаженість процесора та оперативної пам’яті REST

Elapsed_Time(s)	CPU_Usage(%)	RAM_Usage(KB)
0	16	114756
1	19	118340
2	17	121156
3	15	123076
4	18	123716
5	22	126916
6	12	127044
7	17	127300
8	13	127556
9	16	127556
10	18	127684
11	14	127684
12	16	127684
13	16	127684
14	16	127684
15	14	127684
16	17	127812
17	17	127812
18	14	127812
19	11	127812
20	13	127812
21	13	127812

22	17	127812
23	16	127812
24	14	127812
25	19	127812
26	16	127812
27	19	127812
28	12	127812
29	12	127812
30	7	127940

Результати вимірювань GraphQL з мінімальною вибіркою полів за допомогою k6 наступні:

checks_total.....: 169 5.373408/s

checks_succeeded...: 100.00% 169 out of 169

checks_failed.....: 0.00% 0 out of 169

✓ status was 200

HTTP

http_req_duration.....: avg=830.64ms min=583.42ms med=640.68ms max=2.55s p(90)=1.12s p(95)=1.2s

{ expected_response:true }...: avg=830.64ms min=583.42ms med=640.68ms max=2.55s p(90)=1.12s p(95)=1.2s

http_req_failed.....: 0.00% 0 out of 169

http_reqs.....: 169 5.373408/s

EXECUTION

iteration_duration.....: avg=1.83s min=1.58s med=1.64s max=3.56s p(90)=2.12s p(95)=2.2s

iterations.....: 169 5.373408/s

vus.....: 4 min=4 max=10

vus_max.....: 10 min=10 max=10

NETWORK

data_received.....: 158 kB 5.0 kB/s

data_sent.....: 37 kB 1.2 kB/s

Виміри завантаженості процесора та оперативної пам’яті для мінімальної вибірки GraphQL зведено у таблиці 4.2

Таблиця 4.2 - Завантаженість процесора та оперативної пам’яті GraphQL(min)

Elapsed_Time(s)	CPU_Usage(%)	RAM_Usage(KB)
0	33	133712
1	21	134608
2	17	135248
3	17	135504
4	18	135888
5	16	136016
6	15	136272
7	16	136784
8	17	136912
9	14	136912
10	17	136912
11	14	137680
12	18	137680
13	17	137680
14	21	137680
15	10	137808
16	17	137808
17	16	137808
18	18	137808
19	15	137808
20	16	138064
21	21	141776
22	15	142160
23	17	142160
24	18	142160
25	20	142288
26	17	142288
27	16	142288
28	14	142288

29	13	142288
30	11	142288

Результати вимірювань GraphQL з повною вибіркою полів за допомогою k6 наступні:

checks_total.....: 167 5.335491/s

checks_succeeded...: 100.00% 167 out of 167

checks_failed.....: 0.00% 0 out of 167

✓ status was 200

HTTP

http_req_duration.....: avg=839.44ms min=327.89ms med=654.08ms max=2.65s p(90)=1.15s p(95)=1.21

{ expected_response:true }...: avg=839.44ms min=327.89ms med=654.08ms max=2.65s p(90)=1.15s p(95)=1.21

http_req_failed.....: 0.00% 0 out of 167

http_reqs.....: 167 5.335491/s

EXECUTION

iteration_duration.....: avg=1.84s min=1.32s med=1.65s max=3.65s p(90)=2.15s p(95)=2.21

iterations.....: 167 5.335491/s

vus.....: 5 min=5 max=10

vus_max.....: 10 min=10 max=10

NETWORK

data_received.....: 629 kB 20 kB/s

data_sent.....: 54 kB 1.7 kB/s

Виміри завантаженості процесора та оперативної пам’яті для повної вибірки GraphQL зведено у таблиці 4.3

Таблиця 4.3 - Завантаженість процесора та оперативної пам’яті GraphQL

Elapsed_Time(s)	CPU_Usage(%)	RAM_Usage(KB)
0	20	127688
1	26	135240
2	17.82	136008
3	19	136136
4	15	136392
5	17	136520
6	14	136904
7	16	136904
8	17	137160
9	14	137160
10	15	137288
11	17	137672
12	16	137928
13	20	138056
14	20	138184
15	17	138312
16	17	141256
17	19	143944
18	15	144072
19	12	144328
20	15	144328
21	16	145480
22	20	146632
23	15	146888
24	15	146888
25	18	146888
26	17	146888
27	16	146888
28	16	146888
29	18	146888
30	7	146888

Зібравши разом результати експерименту, виділивши найважливіші дані для дослідження за результатів вимірів к6, побудовано порівняльні таблиці 4.4 та 4.5. Середнє навантаження центрального процесора та оперативної пам’яті зведено у таблиці 4.6

Таблиця 4.4 – Порівняння часу виконання запитів у секундах

Сценарій	Медіанне значення (med)	90й перцентиль (p90)	Максимальне значення (max)
REST	0.59	1.07	2.26
GraphQL(min)	0.641	1.12	2.55
GraphQL	0.654	1.15	2.65

Таблиця 4.5 – Порівняння завантаженості мережі (КБ/сек)

Сценарій	Відправлено	Отримано
REST	0.456	22
GraphQL(min)	1.2	5
GraphQL	1.7	20

Таблиця 4.6 – Порівняння завантаженості процесора та пам’яті

Сценарій	Процесор (%)	Оперативна пам’ять (КБ)
REST	15.35484	126478.3
GraphQL(min)	16.93548	138534.7
GraphQL	16.67161	140925.7

На базі таблиць побудовано графіки та діаграми для порівняння час виконання запиту (див. рис. 4.2), завантаженості процесора (див. рис. 4.3), оперативної пам’яті (див. рис. 4.4) та мережі (див. рис. 4.5)

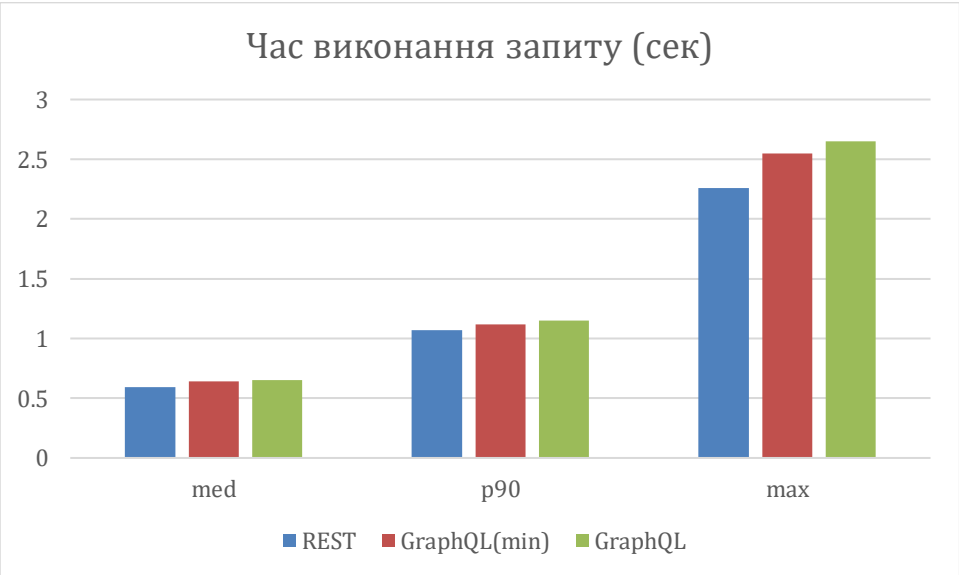

Рисунок 4.2 – час виконання запиту

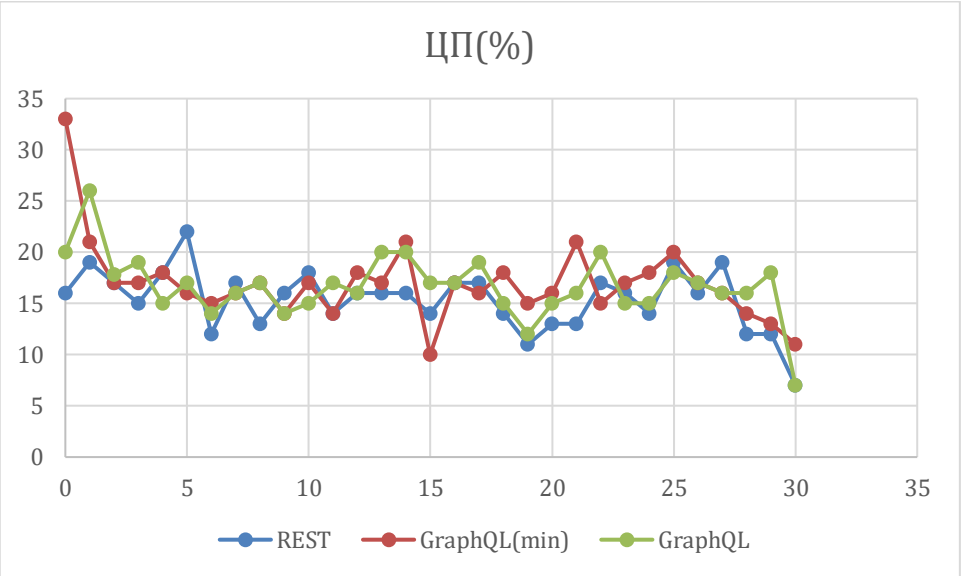

Рисунок 4.3 – завантаженість процесора

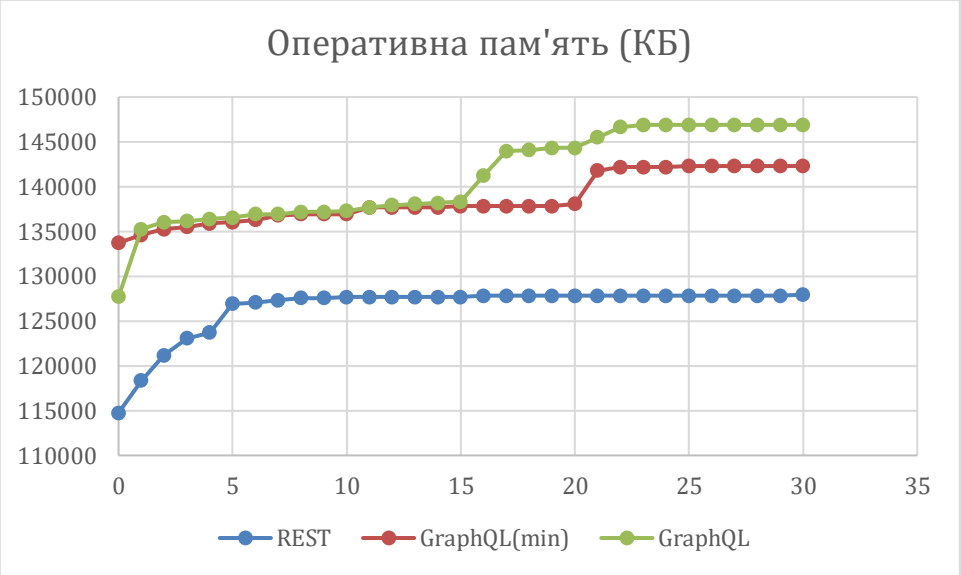

Рисунок 4.4 – завантаженість оперативної пам’яті

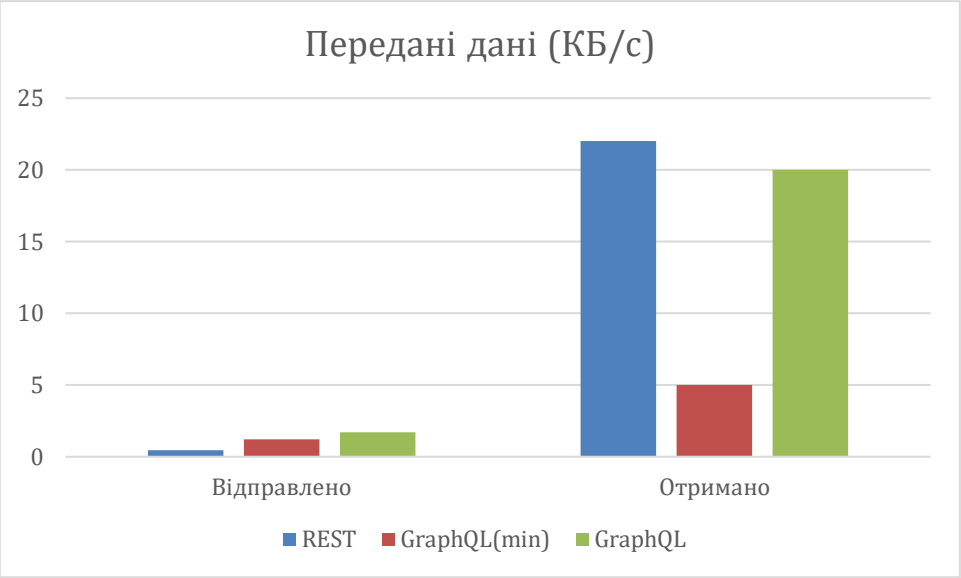

Рисунок 4.5 – завантаженість мережі

У сценарії читання переліку користувачів REST продемонстрував найменші затримки: медіанне значення 0.590 с, p90 1.07 с, максимум 2.26 с. GraphQL(min) був повільнішим за REST за всіма показниками окрім зайнятої пам’яті (медіана +8.5%, p90 +4.7%, max +12.8%), а GraphQL(full) – найповільнішим (медіана +10.8%, p90 +7.5%, max +17.3%). Усі серії завершилися без помилок (100% успішних перевірок k6), що підтверджує функціональну коректність порівнюваних реалізацій.

Мережеві показники підтверджують очікуваний вплив обсягу вибірки: GraphQL(min) є найекономнішим за отриманими даними (5.0 КБ/с проти 22.0 КБ/с у REST, тобто −77.3%). Для GraphQL(full) обсяг отриманих даних близький до REST (20.0 КБ/с), що узгоджується з широким набором полів у відповіді. Водночас `data_sent` для GraphQL є вищим через передачу структури запиту.

За ресурсними характеристиками середні значення CPU у GraphQL(min/full) є близькими між собою (16.94% та 16.67%) і дещо вищими за REST (15.35%). Пам’ять відрізнялася помітніше: середній RSS у REST становив 126478 КБ, у GraphQL(min) – 138535 КБ (+9.5%), у GraphQL(full) – 140926 КБ (+11.4%). Це узгоджується з додатковими накладними витратами виконання GraphQL (парсинг/валідація запиту та диспетчеризація резолверів), а також з більшим обсягом матеріалізованих полів у “full”-варіанті.

Отже, у «плоскому» сценарії читання REST має стабільну перевагу за затримкою, тоді як GraphQL проявляє найбільшу практичну цінність за умови мінімізації вибірки полів (GraphQL(min)), що суттєво зменшує обсяг відповіді без пропорційного виграшу в латентності.

4.2.2 Експеримент 2 – перелік публікацій з авторами

Аналогічна процедура застосовувалася у сценарії «перелік публікацій з авторами»: дві послідовні серії виконувалися для rest_posts, graphql_posts_full та graphql_posts_min. Тут було критично забезпечити функціональну еквівалентність REST-реалізації і GraphQL(full) щодо інформаційної насиченості відповіді, тоді як GraphQL(min) свідомо обмежувався мінімальним набором полів для оцінки впливу зменшеної вибірки на затримку та використання ресурсів.

Результати вимірювань REST API за допомогою k6 наступні:

checks_total.....: 139 4.368865/s

checks_succeeded...: 100.00% 139 out of 139

checks_failed.....: 0.00% 0 out of 139

✓ status was 200

HTTP

http_req_duration.....: avg=1.24s min=895.97ms med=1.03s max=3.11s p(90)=1.68s p(95)=1.72s

{ expected_response:true }...: avg=1.24s min=895.97ms med=1.03s max=3.11s p(90)=1.68s p(95)=1.72s

http_req_failed.....: 0.00% 0 out of 139

http_reqs.....: 139 4.368865/s

EXECUTION

iteration_duration.....: avg=2.24s min=1.89s med=2.03s max=4.11s p(90)=2.68s p(95)=2.73s

iterations.....: 139 4.368865/s

vus.....: 7 min=7 max=10

vus_max.....: 10 min=10 max=10

NETWORK

data_received.....: 14 MB 452 kB/s

data_sent.....: 11 kB 358 B/s

Виміри завантаженості процесора та оперативної пам’яті зведено у таблиці 4.7

Таблиця 4.7 - Завантаженість процесора та оперативної пам’яті REST

Elapsed_Time(s)	CPU_Usage(%)	RAM_Usage(KB)
0	29	130000
1	31	132176
2	21	132688
3	32	134352
4	25	134864
5	32	135888
6	21	136912
7	27	138960
8	19	139216
9	39	139728
10	22	140240
11	30	140880
12	20	141264
13	28	141648
14	23	142032
15	28	142928
16	18	143440
17	30	144208
18	24	144464
19	29	145232
20	25	145744
21	35	146640
22	29	147152
23	20	147792
24	28	148048
25	22	148816

26	27	149712
27	22	150224
28	34	151120
29	26	151504
30	28	152016

Результати вимірювань GraphQL з мінімальною вибіркою полів за допомогою k6 наступні:

checks_total.....: 169 5.387822/s

checks_succeeded...: 100.00% 169 out of 169

checks_failed.....: 0.00% 0 out of 169

✓ status was 200

HTTP

http_req_duration.....: avg=824.33ms min=581.92ms med=642.9ms max=2.44s p(90)=1.13s p(95)=1.2s

{ expected_response:true }...: avg=824.33ms min=581.92ms med=642.9ms max=2.44s p(90)=1.13s p(95)=1.2s

http_req_failed.....: 0.00% 0 out of 169

http_reqs.....: 169 5.387822/s

EXECUTION

iteration_duration.....: avg=1.82s min=1.58s med=1.64s max=3.44s p(90)=2.13s p(95)=2.2s

iterations.....: 169 5.387822/s

vus.....: 4 min=4 max=10

vus_max.....: 10 min=10 max=10

NETWORK

data_received.....: 484 kB 15 kB/s

data_sent.....: 37 kB 1.2 kB/s

Виміри завантаженості процесора та оперативної пам’яті зведено у таблиці 4.8

Таблиця 4.8 - Завантаженість процесора та оперативної пам’яті GraphQL(min)

Elapsed_Time(s)	CPU_Usage(%)	RAM_Usage(KB)
0	26	131436
1	17	133356
2	17	133740
3	20	134252
4	18	134252
5	15	134380
6	18	134508
7	16	135660
8	16	135660
9	15	135788
10	15	135788
11	15	136300
12	17	136300
13	13	136300
14	17	136428
15	15	136556
16	13	136556
17	17	136556
18	19	136556
19	15	136556
20	15	136940
21	16	140012
22	17	140268
23	11	140268
24	19	140396
25	17	140396
26	16	140396
27	16	140396
28	16	140396
29	16	140524
30	10	140524

Результати вимірювань GraphQL з повною вибіркою полів за допомогою k6 наступні:

checks_total.....: 136 4.315268/s

checks_succeeded...: 100.00% 136 out of 136

checks_failed.....: 0.00% 0 out of 136

✓ status was 200

HTTP

http_req_duration.....: avg=1.26s min=905.83ms med=1.08s max=3.37s p(90)=1.62s p(95)=1.84s

{ expected_response:true }...: avg=1.26s min=905.83ms med=1.08s max=3.37s p(90)=1.62s p(95)=1.84s

http_req_failed.....: 0.00% 0 out of 136

http_reqs.....: 136 4.315268/s

EXECUTION

iteration_duration.....: avg=2.26s min=1.9s med=2.08s max=4.37s p(90)=2.62s p(95)=2.84s

iterations.....: 136 4.315268/s

vus.....: 4 min=4 max=10

vus_max.....: 10 min=10 max=10

NETWORK

data_received.....: 14 MB 443 kB/s

data_sent.....: 70 kB 2.2 kB/s

Виміри завантаженості процесора та оперативної пам’яті зведено у таблиці 4.9

Таблиця 4.9 - Завантаженість процесора та оперативної пам’яті GraphQL(full)

Elapsed_Time(s)	CPU_Usage(%)	RAM_Usage(KB)
0	26	131436
1	17	133356

2	17	133740
3	20	134252
4	18	134252
5	15	134380
6	18	134508
7	16	135660
8	16	135660
9	15	135788
10	15	135788
11	15	136300
12	17	136300
13	13	136300
14	17	136428
15	15	136556
16	13	136556
17	17	136556
18	19	136556
19	15	136556
20	15	136940
21	16	140012
22	17	140268
23	11	140268
24	19	140396
25	17	140396
26	16	140396
27	16	140396
28	16	140396
29	16	140524
30	10	140524

Зібравши разом результати експерименту, виділивши найважливіші дані для дослідження за результатів вимірів к6, побудовано порівняльні таблиці 4.10 та 4.11. Середнє навантаження центрального процесора та оперативної пам’яті зведено у таблиці 4.12

Таблиця 4.10 – Порівняння часу виконання запитів у секундах

Сценарій	Медіанне значення (med)	90й перцентиль (p90)	Максимальне значення (max)
REST	1.03	1.68	3.11
GraphQL(min)	0.643	1.13	2.44
GraphQL	1.08	1.62	3.37

Таблиця 4.11 – Порівняння завантаженості мережі (КБ/сек)

Сценарій	Відправлено	Отримано
REST	0.358	452
GraphQL(min)	1.2	15
GraphQL	2.2	443

Таблиця 4.12 – Порівняння завантаженості процесора та пам’яті

Сценарій	Процесор (%)	Оперативна пам’ять (КБ)
REST	26.58064516	142577.0323
GraphQL(min)	16.22580645	137014.3226
GraphQL	27.64516129	159318.7097

На базі таблиць побудовано графіки та діаграми для порівняння час виконання запиту (див. рис. 4.6), завантаженості процесора (див. рис. 4.7), оперативної пам’яті (див. рис. 4.8) та мережі (див. рис. 4.9)

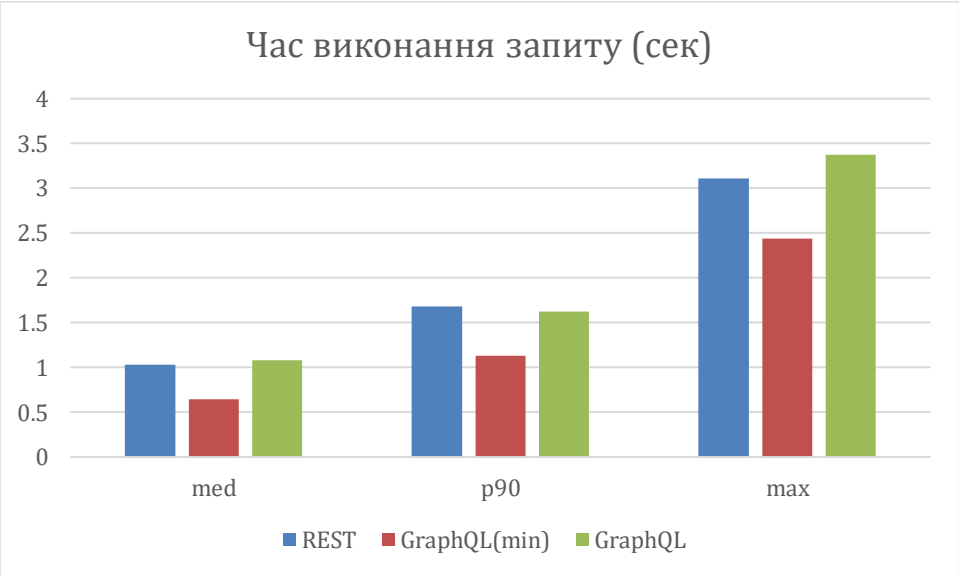

Рисунок 4.6 – час виконання запиту

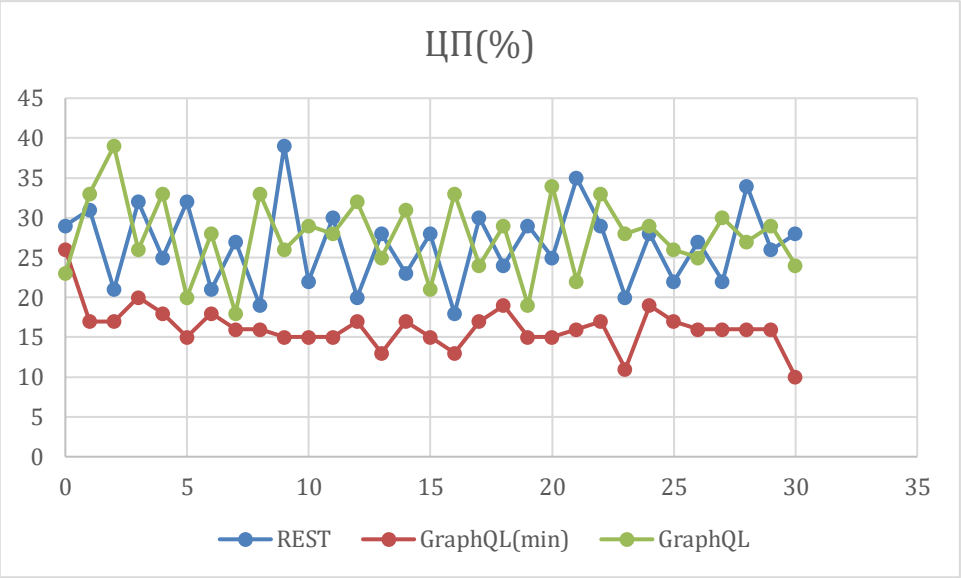

Рисунок 4.7 – завантаженість процесора

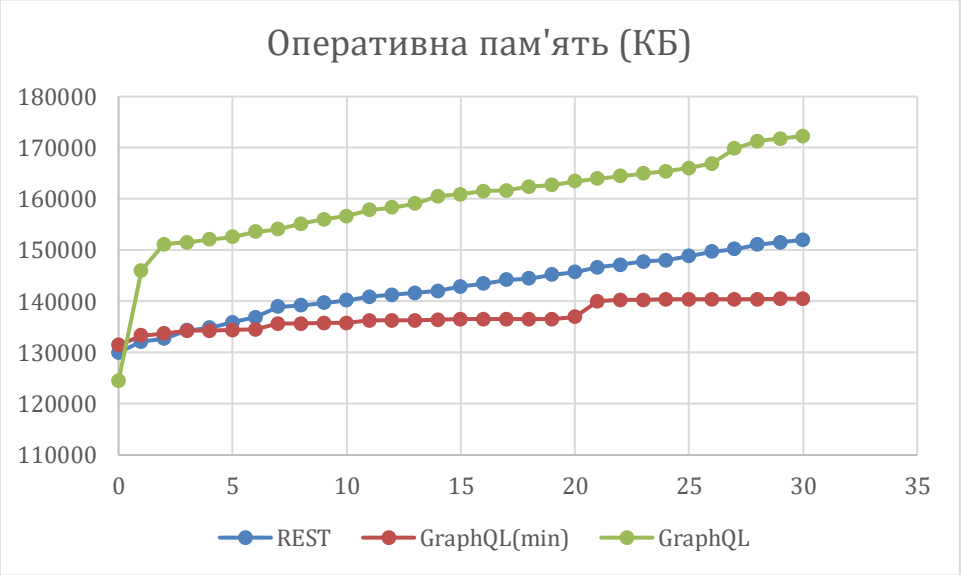

Рисунок 4.8 – завантаженість оперативної пам’яті

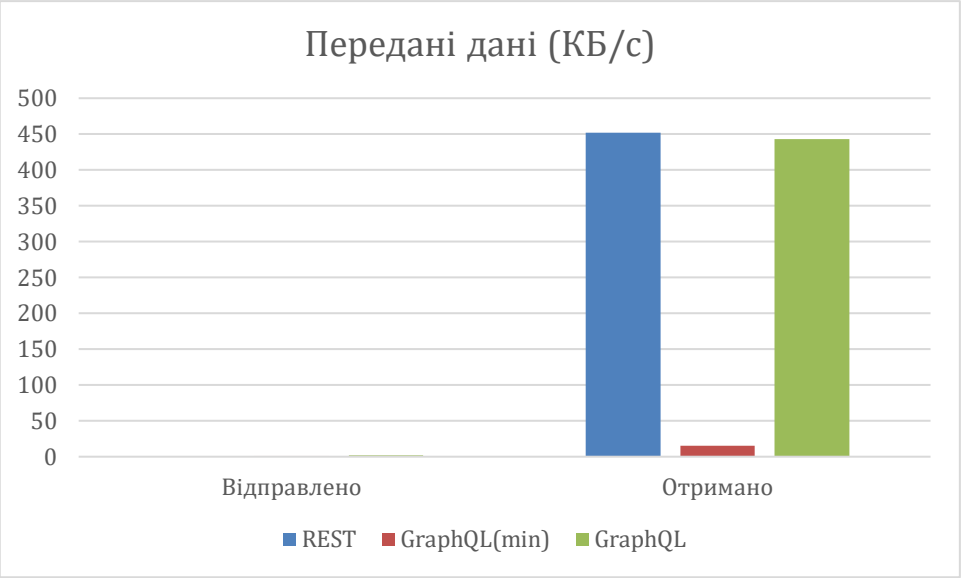

Рисунок 4.9 – завантаженість мережі

У сценарії читання переліку публікацій разом із даними автора найкращі часові характеристики продемонстрував запит GraphQL із мінімальною вибіркою полів: медіанне значення становило 0.643 с, 90-й перцентиль – 1.13 с, максимальне – 2.44 с. Для REST відповідні показники дорівнювали 1.03 с, 1.68 с і 3.11 с, тоді як для GraphQL із повною відповіддю – 1.08 с, 1.62 с і 3.37 с. Отже, порівняно з REST, GraphQL(min) зменшує медіанну затримку приблизно на 38% і p90 на ~33%, тоді як GraphQL(full) за медіаною повільніший за REST (~+5%), має трохи нижчий p90 (1.62 с проти 1.68 с), але гірший «хвіст» (max 3.37 с проти 3.11 с).

За ресурсними метриками GraphQL(min) також є найекономнішим: середнє завантаження ЦП – 16.23% (проти 26.58% у REST і 27.65% у GraphQL(full)), середній RSS – 137 014 КБ (проти 142 577 КБ у REST і 159 319 КБ у GraphQL(full)). Це свідчить, що зменшення обсягу матеріалізованих полів та роботи резолверів прямо знижує навантаження як на процесор, так і на пам’ять; навпаки, «повна» відповідь у GraphQL створює найвищий тиск на пам’ять і дещо підвищує ЦП.

Мережевий профіль підкреслює ключову перевагу вибірковості GraphQL: отримані дані для GraphQL(min) становили лише ~15 КБ/с, тоді як для REST – ~452 КБ/с, а для GraphQL(full) – ~443 КБ/с. Відправлені дані для GraphQL вищі (1.2 КБ/с у min та 2.2 КБ/с у full проти 0.358 КБ/с у REST) через передачу структури запиту, однак цей наклад є на порядки меншим за економію на обсязі отриманої відповіді. Сукупно

це означає, що саме контрольований вибір мінімально потрібних полів у GraphQL забезпечує найбільший практичний виграш: нижча затримка, менші витрати ЦП/ОП і радикально нижчий мережевий трафік. Натомість GraphQL(full) за цього сценарію не дає переваг над REST і навіть погіршує пам’ять і максимальні затримки, що узгоджується з очікуваним ефектом overfetch.

4.2.3 Експеримент 3 – створення публікацій

У сценарії «створення публікацій» дві серії запускалися для rest_create_post, graphql_create_post та graphql_create_post_min. На рівні навантаження це відповідало виконанню HTTP-запиту POST у варіанті REST та GraphQL-мутацій у двох варіантах повернення даних (повному та мінімальному). Для уникнення колізій та сторонніх ефектів кожна операція створення формувала унікальні дані (зокрема, заголовок із міткою часу), а перевірка коректності виконання здійснювалася вбудованими перевірками k6.

Результати вимірювань REST API за допомогою k6 наступні:

checks_total.....: 342 10.841972/s

checks_succeeded...: 100.00% 342 out of 342

checks_failed.....: 0.00% 0 out of 342

✓ status was 201

✓ success response

HTTP

http_req_duration.....: avg=807.44ms min=563.27ms med=644ms max=2.36s p(90)=1.1s p(95)=1.19s

{ expected_response:true }...: avg=807.44ms min=563.27ms med=644ms max=2.36s p(90)=1.1s p(95)=1.19s

http_req_failed.....: 0.00% 0 out of 171

http_reqs.....: 171 5.420986/s

EXECUTION

iteration_duration.....: avg=1.8s min=1.56s med=1.64s max=3.36s p(90)=2.11s p(95)=2.19s

iterations.....: 171 5.420986/s

vus.....: 5 min=5 max=10

vus_max.....: 10 min=10 max=10

NETWORK

data_received.....: 179 kB 5.7 kB/s

data_sent.....: 58 kB 1.8 kB/s

Виміри завантаженості процесора та оперативної пам’яті зведено у таблиці 4.13

Таблиця 4.13 - Завантаженість процесора та оперативної пам’яті REST

Elapsed_Time(s)	CPU_Usage(%)	RAM_Usage(KB)
0	19	116556
1	21	123724
2	15	127948
3	15	131532
4	18	134604
5	22	138572
6	12	138572
7	19	138572
8	16	138700
9	22	138700
10	14	138700
11	21	138700
12	18	138828
13	13	138828
14	12	138828
15	16	138828
16	14	138956
17	15.84	138956
18	19	138956
19	15	138956
20	13	138956
21	17	138956
22	17	138956
23	13	138956

24	13	138956
25	12	138956
26	15	138956
27	15	138956
28	16	138956
29	15	138956
30	8	138956

Результати вимірювань GraphQL з мінімальною вибіркою полів за допомогою k6 наступні:

checks_total.....: 338 10.722546/s

checks_succeeded...: 100.00% 338 out of 338

checks_failed.....: 0.00% 0 out of 338

✓ status was 200

✓ no errors

HTTP

http_req_duration.....: avg=834.74ms min=575.88ms med=655.5ms max=2.49s p(90)=1.15s p(95)=1.27s

{ expected_response:true }...: avg=834.74ms min=575.88ms med=655.5ms max=2.49s p(90)=1.15s p(95)=1.27s

http_req_failed.....: 0.00% 0 out of 169

http_reqs.....: 169 5.361273/s

EXECUTION

iteration_duration.....: avg=1.83s min=1.57s med=1.65s max=3.49s p(90)=2.15s p(95)=2.27s

iterations.....: 169 5.361273/s

vus.....: 4 min=4 max=10

vus_max.....: 10 min=10 max=10

NETWORK

data_received.....: 124 kB 3.9 kB/s

data_sent.....: 89 kB 2.8 kB/s

Виміри завантаженості процесора та оперативної пам’яті зведено у таблиці 4.14

Таблиця 4.14 - Завантаженість процесора та оперативної пам’яті GraphQL(min)

Elapsed_Time(s)	CPU_Usage(%)	RAM_Usage(KB)
0	27	134328
1	16	138168
2	14	138552
3	13	139832
4	18	140856
5	19	141112
6	14	141368
7	14	142008
8	21	142008
9	14	142264
10	16	142264
11	15	143288
12	15	143544
13	16	143672
14	15	143672
15	18	143800
16	17	143800
17	14	143800
18	18	143928
19	18	143928
20	13	144056
21	12	145080
22	20	148408
23	15	148408
24	12	148664
25	12	148664
26	14	148664
27	14	148664
28	19	148792
29	13	148792
30	10	148792

Результати вимірювань GraphQL з повною вибіркою полів за допомогою k6 наступні:

checks_total.....: 332 10.629495/s

checks_succeeded...: 100.00% 332 out of 332

checks_failed.....: 0.00% 0 out of 332

✓ status was 200

✓ no errors

HTTP

http_req_duration.....: avg=850.44ms min=585.09ms med=676.27ms max=2.6s p(90)=1.18s p(95)=1.25s

{ expected_response:true }...: avg=850.44ms min=585.09ms med=676.27ms max=2.6s p(90)=1.18s p(95)=1.25s

http_req_failed.....: 0.00% 0 out of 166

http_reqs.....: 166 5.314747/s

EXECUTION

iteration_duration.....: avg=1.85s min=1.58s med=1.67s max=3.6s p(90)=2.18s p(95)=2.25s

iterations.....: 166 5.314747/s

vus.....: 4 min=4 max=10

vus_max.....: 10 min=10 max=10

NETWORK

data_received.....: 172 kB 5.5 kB/s

data_sent.....: 111 kB 3.5 kB/s

Виміри завантаженості процесора та оперативної пам’яті зведено у таблиці 4.15

Таблиця 4.15 - Завантаженість процесора та оперативної пам’яті GraphQL(full)

Elapsed_Time(s)	CPU_Usage(%)	RAM_Usage(KB)
0	22	125280
1	19	137056
2	18	140000
3	16	141408
4	16	141664
5	20	142176
6	15	142944
7	17	143200
8	16	143200
9	17	143200
10	20	143200
11	13	144096
12	15	144352
13	17	144480
14	13	144480
15	15	144480
16	18	144480
17	15	144608
18	19	144608
19	12	144864
20	19	144864
21	18	146016
22	17	150368
23	16	150624
24	15	150624
25	17	150624
26	14	150624
27	15	150624
28	18	150624
29	16	150624
30	8	150624

Зібравши разом результати експерименту, виділивши найважливіші дані для дослідження за результатів вимірів к6, побудовано порівняльні таблиці 4.16 та 4.17. Середнє навантаження центрального процесора та оперативної пам’яті зведено у таблиці 4.18

Таблиця 4.16 – Порівняння часу виконання запитів у секундах

Сценарій	Медіанне значення (med)	90й перцентиль (p90)	Максимальне значення (max)
REST	0.644	1.1	2.36
GraphQL(min)	0.655	1.15	2.49
GraphQL	0.676	1.18	2.6

Таблиця 4.17 – Порівняння завантаженості мережі (КБ/сек)

Сценарій	Відправлено	Отримано
REST	1.8	5.7
GraphQL(min)	2.8	3.9
GraphQL	3.5	5.5

Таблиця 4.18 – Порівняння завантаженості процесора та пам’яті

Сценарій	Процесор (%)	Оперативна пам’ять (КБ)
REST	15.83354839	136920.3871
GraphQL(min)	15.67741935	143973.4194
GraphQL	16.32258065	144839.2258

На базі таблиць побудовано графіки та діаграми для порівняння час виконання запиту (див. рис. 4.10), завантаженості процесора (див. рис. 4.11), оперативної пам’яті (див. рис. 4.12) та мережі (див. рис. 4.13)

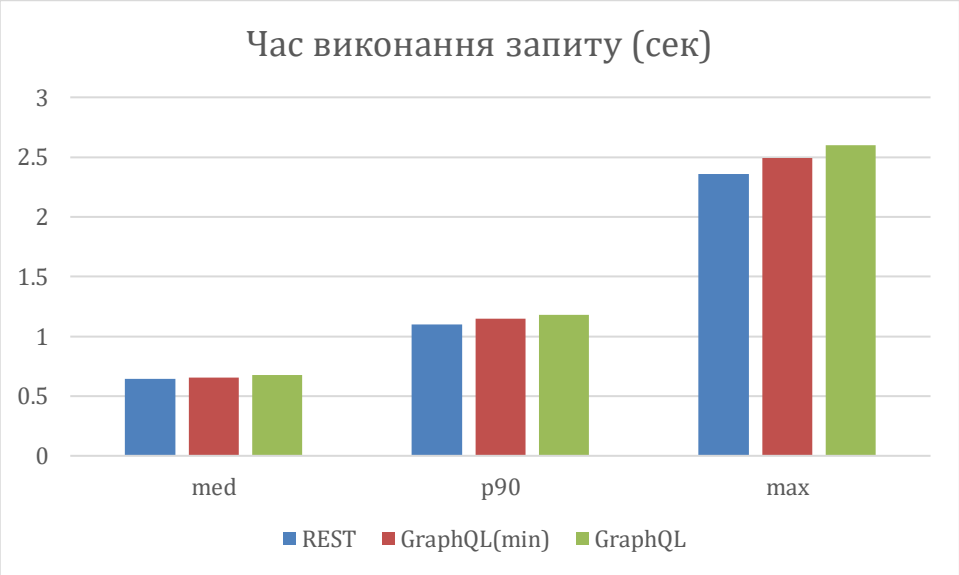

Рисунок 4.10 – час виконання запиту

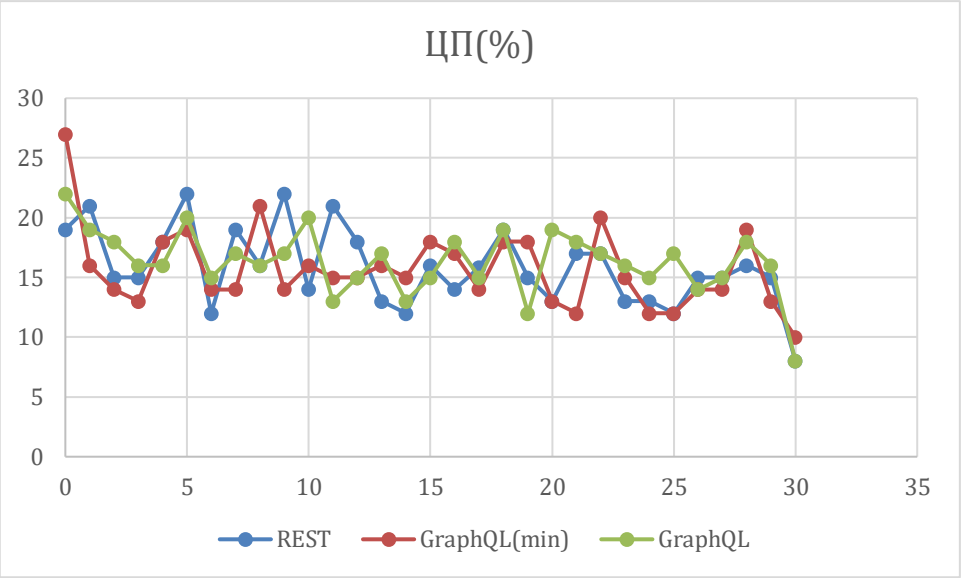

Рисунок 4.11 – завантаженість процесора

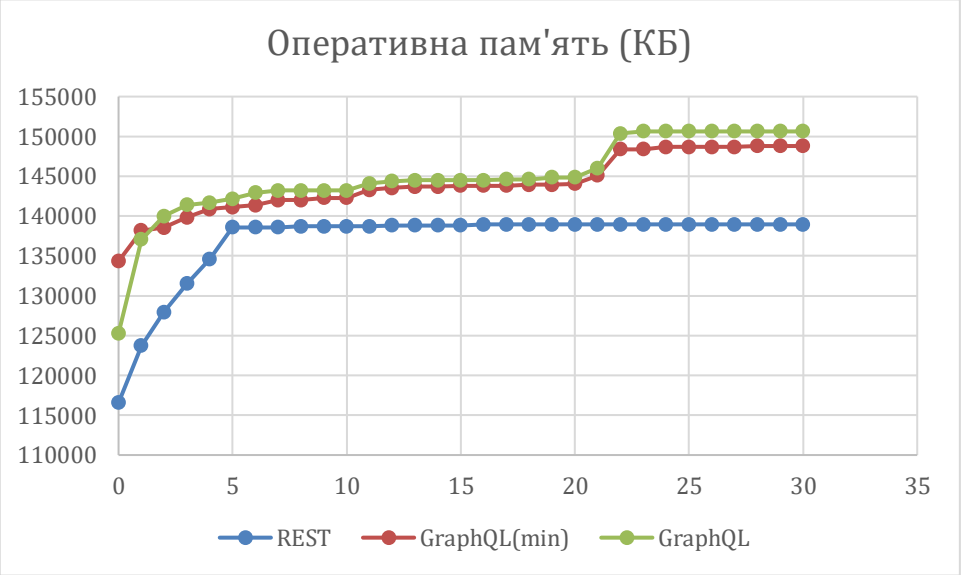

Рисунок 4.12 – завантаженість оперативної пам’яті

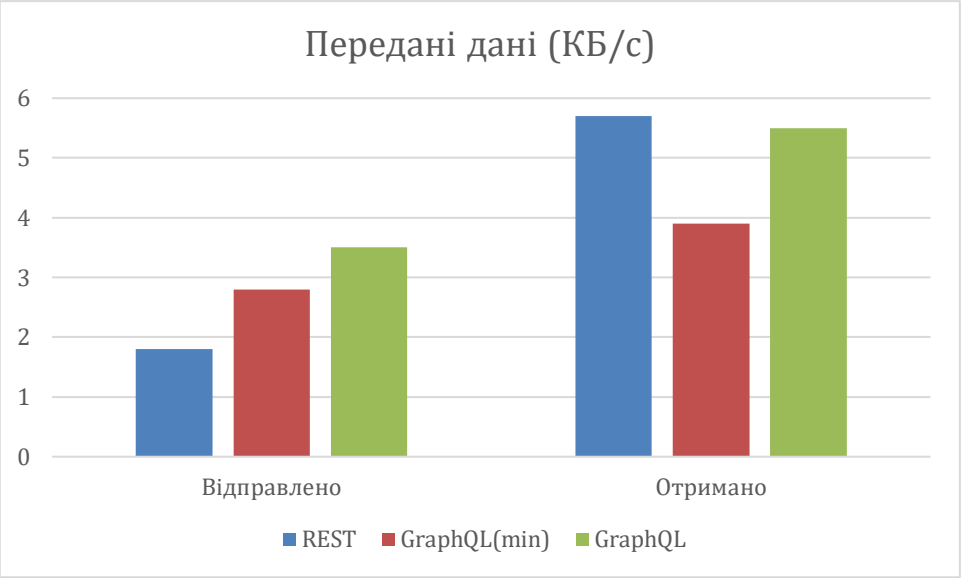

Рисунок 4.13 – завантаженість мережі

У сценарії запису (створення публікації) спостерігається близькість часових характеристик між усіма трьома підходами, з невеликою перевагою REST. Для REST медіанне значення становило 0.644 с, 90-й перцентиль – 1.10 с, максимальне – 2.36 с. Для GraphQL(min) відповідно 0.655 с (+1.7% до медіани), 1.15 с (+4.5% до p90) та 2.49 с (+5.5% до максимального). Для GraphQL(full) – 0.676 с (+5.0%), 1.18 с (+7.3%) і 2.60 с (+10.2%). Отже, при операції запису REST утримує невелику перевагу за всіма квантилями затримки; GraphQL(min) є майже рівним за медіаною, а GraphQL(full) послідовно повільніший.

Ресурсні показники підтверджують, що різниця у ЦП є мінімальною: REST – 15.83%, GraphQL(min) – 15.68% (навіть трохи нижче), GraphQL(full) – 16.32% (≈+3.1% до REST). Натомість пам’ять у GraphQL помітно вища: 136 920 КБ у REST проти 143 973 КБ (+5.15%) у GraphQL(min) та 144 839 КБ (+5.79%) у GraphQL(full). Це узгоджується з накладними витратами резолверів і формуванням відповіді, особливо у “full”-варіанті.

Мережеві характеристики демонструють типовий компроміс GraphQL: відправлені дані більші через передачу структури запиту (REST – 1.8 КБ/с; GraphQL(min) – 2.8 КБ/с; GraphQL(full) – 3.5 КБ/с), зате обсяг отриманих даних може бути нижчим за рахунок вибірковості полів (REST – 5.7 КБ/с; GraphQL(min) – 3.9 КБ/с, що на ≈31.6% менше; GraphQL(full) – 5.5 КБ/с, близько до REST). Таким чином,

у сценарії запису GraphQL(min) забезпечує мережеву економію на відповіді без істотного погіршення затримки, тоді як GraphQL(full) збільшує як мережеві, так і пам’ятні накладні.

Сумарно, для створення ресурсу REST лишається найшвидшим за медіаною та p90, різниця з GraphQL(min) невелика і може бути прийнятною, якщо пріоритет – зменшення обсягу відповіді й уніфікація доступу через GraphQL-схему. Використання GraphQL(full) у цьому сценарії доцільно обмежувати випадками, коли дійсно необхідно повертати розширений набір полів; інакше воно створює непотрібний тиск на пам’ять і дещо підвищує латентність.

ВИСНОВКИ ДО РОЗДІЛУ 4

У розділі наведено хід та результати експериментального дослідження продуктивності GraphQL у порівнянні з REST API, а також аналіз впливу обсягу вибірки даних на часові та ресурсні характеристики веб-додатку.

Виявлено такі закономірності впливу структурної складності запиту (плоскі дані проти ієрархічних) на показники затримки та швидкодії:

- при операціях читання «плоских» колекцій (без вкладеностей) REST API демонструє менші затримки порівняно з GraphQL;
- при операціях читання глибоко вкладених (ієрархічних) даних GraphQL із оптимізованою вибіркою (min) значно перевершує REST API за швидкодією;
- операції запису (мутації) мають порівнювані часові характеристики, проте REST зберігає незначну перевагу.

Так, у сценарії читання списку користувачів («плоский» запит) медіанна затримка REST склала 0,59 с, що на 8,5 % швидше за GraphQL(min) (0,641 с). Натомість у сценарії читання публікацій з авторами («ієрархічний» запит) GraphQL(min) виявився на 38 % швидшим за REST (0,643 с проти 1,03 с), оскільки REST змушений був передавати надлишкові дані зв’язаних сутностей.

Виявлено такі закономірності впливу обсягу вибірки полів (GraphQL Full vs Min) на мережеві та ресурсні метрики:

- при зменшенні кількості запитуваних полів у GraphQL (усунення overfetching) суттєво зменшується обсяг вхідного трафіку та споживання оперативної пам’яті сервера;
- при запиті повного набору полів (GraphQL Full) показники продуктивності наближаються до REST або стають гіршими через накладні витрати на обробку графа.

Так, у сценарії з публікаціями перехід від повної вибірки (GraphQL Full) до мінімальної (GraphQL Min) дозволив зменшити споживання оперативної пам’яті на 14 % (з 159 МБ до 137 МБ) та знизити навантаження на процесор з 27,6 % до 16,2 %. Обсяг отриманих даних мережею при цьому впав у десятки разів (з 443 КБ/с до 15 КБ/с).

Виявлено такі закономірності щодо ресурсних накладних витрат архітектурних стилів: – GraphQL загалом споживає більше оперативної пам’яті порівняно з REST для аналогічних операцій через витрати на парсинг запитів та валідацію схеми; – вихідний трафік (data sent) у GraphQL завжди вищий через необхідність передачі тіла запиту з переліком полів.

Так, у сценарії створення публікації споживання RAM для GraphQL було на 5–6 % вищим, ніж для REST (144 МБ проти 137 МБ), а обсяг відправлених клієнтом даних був більшим майже вдвічі (3,5 КБ/с проти 1,8 КБ/с).

Крім того, результати підтвердили, що переваги GraphQL є найбільш вираженими у сценаріях читання складних зв’язаних даних, де гнучкість вибірки дозволяє компенсувати накладні витрати виконання.

ВИСНОВКИ

У роботі вирішено науково-прикладну задачу визначення ефективності використання технологій REST та GraphQL у веб-додатках. Реалізовано інструментальне забезпечення для порівняльного аналізу, яке, на відміну від синтетичних тестів, працює з реальною реляційною моделлю даних (користувачі, публікації, коментарі) та відтворює проблему «N+1 запиту». Це дозволило отримати достовірні метрики продуктивності для трьох варіантів взаємодії: REST, GraphQL з повною вибіркою (full) та GraphQL з оптимізованою вибіркою (min).

На основі розробленого програмного стенда на базі Ruby on Rails 8 та PostgreSQL проведено комплексне порівняльне дослідження продуктивності API за показниками часової затримки, споживання системних ресурсів та навантаження на мережу.

Експериментально встановлено залежність часової ефективності API від структурної складності запиту:

у сценаріях отримання «плоских» списків (перелік користувачів) REST API демонструє найкращу швидкодію. Медіанний час відгуку REST склав 0,59 с, що на 8,5% швидше за оптимізований GraphQL (0,641 с) та на 10,8% швидше за повний GraphQL.

у сценаріях отримання ієрархічних даних (публікації разом з авторами) GraphQL із мінімальною вибіркою показав суттєву перевагу. Медіанний час відгуку скоротився на 37,6% порівняно з REST (0,643 с проти 1,03 с), а показник 90-го перцентилю зменшився на 32,7% (1,13 с проти 1,68 с). Це підтверджує гіпотезу про неефективність REST при агрегації зв'язаних сутностей через проблему надлишкової передачі даних.

Виявлено закономірності впливу дисципліни вибірки полів (Field Selection) на мережевий трафік та використання пам’яті. Використання мінімальної проєкції даних у GraphQL дозволяє радикально знизити навантаження на мережу в сценаріях читання. Обсяг отриманих даних зменшився з 452 КБ/с (REST) та 443 КБ/с (GraphQL full) до 15 КБ/с (GraphQL min), що становить скорочення трафіку на 96,6%.

Доведено, що запит надлишкових полів у GraphQL (Over-fetching) призводить до невиправданих витрат ресурсів сервера. У сценарії з публікаціями перехід від повної до мінімальної вибірки дозволив зменшити споживання оперативної пам’яті на 14% (з 159 МБ до 137 МБ) та знизити середнє навантаження на процесор з 27,6% до 16,2%.

Визначено характеристики ефективності при операціях модифікації даних (Create).

При створенні ресурсів REST API зберігає незначну перевагу в швидкодії: медіанна затримка склала 0,644 с, що на 1,7% менше за GraphQL Min та на 5% менше за GraphQL Full.

Встановлено, що GraphQL має стабільно вищі накладні витрати на оперативну пам'ять у всіх сценаріях (на 5–11% більше порівняно з REST) через витрати на парсинг запитів, валідацію схеми та роботу резолверів. Також вихідний трафік (від клієнта до сервера) у GraphQL у 2–5 разів перевищує показники REST через необхідність передачі тіла запиту.

Практична цінність роботи полягає у формулюванні інженерних рекомендацій: для простих CRUD-операцій та систем з обмеженими ресурсами пам'яті доцільно використовувати REST; для клієнтів зі складною візуалізацією зв'язаних даних (мобільні додатки, дашборди) використання GraphQL дозволяє скоротити час відгуку на третину та трафік майже на порядок, але лише за умови суворого контролю вибірки полів на клієнті.

Перспективи подальших досліджень полягають у вивченні впливу механізмів кешування (HTTP-кешування для REST проти персистентних запитів для GraphQL) на загальну продуктивність системи, а також у порівняльному аналізі продуктивності підходів в умовах мікросервісної архітектури (Apollo Federation).

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Fielding R. T. Architectural Styles and the Design of Network-based Software Architectures [Electronic resource] : Dissertation / R. T. Fielding. – 2000. – Access mode : URL: https://ics.uci.edu/~fielding/pubs/dissertation/fielding_dissertation.pdf. – Date of Access: 22 November 2025.
2. GraphQL: A query language for APIs [Electronic resource] / Facebook Inc. – Access mode : URL: https://graphql.org. – Date of Access: 22 November 2025.
3. What's the Difference Between GraphQL and REST? [Electronic resource] / AWS. – Access mode : URL: https://aws.amazon.com/compare/the-difference-between-graphql-and-rest/. – Date of Access: 22 November 2025.
4. GraphQL, REST, OpenAPI Trend Analysis 2023 [Electronic resource] / WunderGraph. – Access mode : URL: https://wundergraph.com/blog/graphql_rest_openapi_trend_analysis_2023. – Date of Access: 22 November 2025.
5. History of REST APIs [Electronic resource]. – Access mode : URL: https://www.mobapi.com/history-of-rest-apis/. – Date of Access: 22 November 2025.
6. OpenAPI [Electronic resource] / Open API Initiative. – 2017. – Access mode : URL: https://www.openapis.org/. – Date of Access: 22 November 2025.
7. Best practices for RESTful web API design [Electronic resource] / Microsoft. – Access mode : URL: https://learn.microsoft.com/en-us/azure/architecture/best-practices/api-design. – Date of Access: 22 November 2025.
8. SOAP Version 1.2 Part 0: Primer (Second Edition) [Electronic resource] / W3C. – Access mode : URL: https://www.w3.org/TR/2007/REC-soap12-part0-20070427/. – Date of Access: 22 November 2025.
9. Ruby on Rails: Compress the complexity of modern web apps [Electronic resource]. – Access mode : URL: https://rubyonrails.org/. – Date of Access: 22 November 2025.
10. Introducing JSON [Electronic resource]. – Access mode : URL: https://www.json.org/json-en.html. – Date of Access: 22 November 2025.

11. Shtatnov A. Our learnings from adopting GraphQL [Electronic resource] / A. Shtatnov, R. S. Ranganathan // Netflix TechBlog. – 2018. – Access mode : URL: https://medium.com/netflix-techblog/our-learnings-from-adopting-graphql-f099de39ae5f. – Date of Access: 22 November 2025.
12. How to Create a GraphQL API With Ruby on Rails [Electronic resource] / Devot Team. – Access mode : URL: https://devot.team/blog/how-to-create-a-graphql-api-with-ruby-on-rails. – Date of Access: 22 November 2025.
13. Porcello E. Learning GraphQL: Declarative Data Fetching for Modern Web Apps [Text] / E. Porcello, A. Banks. – 1st edn. – Sebastopol : O'Reilly Media, 2018. – 206 p.
14. Stępień K. Performance evaluation of REST and GraphQL API approaches in data retrieval scenarios using NestJS [Text] / K. Stępień, M. Skublewska-Paszkowska // Journal of Computer Sciences Institute. – 2025. – Vol. 36. – P. 350–356.
15. Handling the N+1 Problem [Electronic resource] / Apollo GraphQL. – Access mode : URL: https://www.apollographql.com/docs/graphos/schema-design/guides/handling-n-plus-one. – Date of Access: 22 November 2025.
16. Puustinen O. GraphQL for building microservices [Electronic resource] / O. Puustinen. – Tampere University, 2020. – P. 30–31.
17. Brito G. REST vs GraphQL: A Controlled Experiment [Text] / G. Brito, M. T. Valente // arXiv preprint arXiv:2003.04761. – 2020.
18. Niklasson A. RESTful API vs. GraphQL a CRUD performance comparison [Text] / A. Niklasson, V. Werélius. – Linnaeus University, 2023.
19. Optimizing GraphQL N+1 query problems in Ruby on Rails [Electronic resource] / Fullstack Labs. – Access mode : URL: https://www.fullstack.com/labs/resources/blog/optimizing-graphql-n-1-query-problems-in-ruby-on-rails. – Date of Access: 22 November 2025.
20. Efficient GraphQL queries in Ruby on Rails & Postgres [Electronic resource] / Pganalyze. – Access mode : URL: https://pganalyze.com/blog/efficient-graphql-queries-in-ruby-on-rails-and-postgres. – Date of Access: 22 November 2025.
21. Переяславська С. О. Проєктування рівня маршрутизації в мікросервісних архітектурах на платформі Spring [Текст] / С. О. Переяславська, О. А. Смагіна //

Сучасні інформаційні системи. – 2023. – Т. 7, № 1. – С. 64–71. – DOI: 10.30837/ITSSI.2023.25.064.

22. YJIT - Yet Another Ruby JIT [Electronic resource]. – Access mode : URL: https://docs.ruby-lang.org/en/3.4/yjit/yjit_md.html. – Date of Access: 22 November 2025.
23. pidstat(1) – Linux manual page [Electronic resource]. – Access mode : URL: https://man7.org/linux/man-pages/man1/pidstat.1.html. – Date of Access: 22 November 2025.
24. Grafana k6: Load testing for engineering teams [Electronic resource]. – Access mode : URL: https://k6.io/. – Date of Access: 22 November 2025.
25. Landeiro M. I. F. Analysis of GraphQL performance: a case study [Text] : Dissertation / M. I. F. Landeiro. – Coimbra : University of Coimbra, 2019.
26. Newman S. Building Microservices: Designing Fine-Grained Systems [Text] / S. Newman. – 2nd Edition. – London : O'Reilly Media, 2021. – 612 p.
27. GraphQL: A systematic mapping study [Text] / A. Quiña-Mera, P. Fernández, J. M. García, A. Ruiz-Cortés // ACM Computing Surveys. – 2023. – Vol. 55, Issue 10. – P. 1–35. – DOI: 10.1145/3561818.
28. Niklasson A. RESTful API vs. GraphQL a CRUD performance comparison [Text] : Bachelor Degree Project / A. Niklasson, V. Werélius. – Linnaeus University, 2023.
29. Abdelfattah A. S. Roadmap to Reasoning in Microservice Systems: A Rapid Review [Text] / A. S. Abdelfattah, T. Cerny // Applied Sciences. – 2023. – Vol. 13, № 3. – 1838 p. – DOI: 10.3390/app13031838.
30. Khan I. A. A Comparative Analysis of REST and GraphQL APIs: Performance, Efficiency, and Developer Experience [Text] / I. A. Khan, H. Mishra, K. Choubey // International Journal of Advanced Multidisciplinary Scientific Research (IJAMSR). – 2025. – Vol. 8, Issue 4. – DOI: 10.31426/ijamsr.2025.8.4.8212.
31. Kazanavicius J. Evaluation of microservice communication while decomposing monoliths [Text] / J. Kazanavicius, D. Mazeika // Computing and Informatics. – 2023. – Vol. 42. – P. 1–36. – DOI: 10.31577/cai_2023_1_1.

32. Hartig O. Semantics and complexity of GraphQL [Text] / O. Hartig, J. Pérez // Proceedings of the 2018 World Wide Web Conference (WWW '18). – 2018. – P. 1155–1164. – DOI: 10.1145/3178876.3186014.
33. Brito G. Migrating to GraphQL: A practical assessment [Text] / G. Brito, T. Mombach, M. T. Valente // 2019 IEEE 26th International Conference on Software Analysis, Evolution and Reengineering (SANER). – 2019. – P. 140–150. – DOI: 10.1109/SANER.2019.8667986.
34. Lawi A. Evaluating GraphQL and REST API services performance in a massive and intensive accessible information system [Text] / A. Lawi, B. L. E. Panggabean, T. Yoshida // Computers. – 2021. – Vol. 10, № 11. – P. 138. – DOI: 10.3390/computers10110138.
35. Hartina D. A. Performance analysis of GraphQL and RESTful in SIM LP2M of the Hasanuddin University [Text] / D. A. Hartina, A. Lawi, B. L. E. Panggabean // 2018 2nd East Indonesia Conference on Computer and Information Technology (EIConCIT). – 2018. – P. 237–240. – DOI: 10.1109/EIConCIT.2018.8878524.
36. Lee E. Performance Measurement of GraphQL API in Home ESS Data Server [Text] / E. Lee, K. Kwon, J. Yun // 2020 International Conference on Information and Communication Technology Convergence (ICTC). – 2020. – P. 1929–1931. – DOI: 10.1109/ictc49870.2020.9289569.
37. Mikuła M. Comparison of REST and GraphQL web technology performance [Text] / M. Mikuła, M. Dzieńkowski // Journal of Computer Sciences Institute. – 2020. – Vol. 16. – P. 309–316. – DOI: 10.35784/jcsi.2077.
38. Margański P. REST and GraphQL comparative analysis [Text] / P. Margański, B. Pańczyk // Journal of Computer Sciences Institute. – 2021. – Vol. 19. – P. 89–94. – DOI: 10.35784/jcsi.2473.
39. Seabra M. REST or GraphQL? A Performance Comparative Study [Text] / M. Seabra [et al.] // Proceedings of the XIII Brazilian Symposium on Software Components, Architectures, and Reuse (SBCARS '19). – 2019. – P. 123–132.
40. Vesić M. Comparative Analysis of Web Application Performance in Case of Using REST versus GraphQL [Text] / M. Vesić, N. Kojić // Proceedings of the Fourth

International Scientific Conference on Recent Advances in Information Technology (ITEMA). – 2020.

41. Sayago Heredia J. P. Comparative Analysis Between Standards Oriented to Web Services: SOAP, REST, and GraphQL [Text] / J. P. Sayago Heredia // Communications in Computer and Information Science. – 2019. – Vol. 1141. – P. 260–275.
42. Stubailo S. GraphQL vs. REST [Electronic resource] / S. Stubailo. – 2017. – Access mode : URL: https://blog.apollographql.com/graphql-vs-rest-5d425123e34b. – Date of Access: 22 November 2025.
43. Guha S. A Comparative Study between GraphQL & Restful Services in API Management of Stateless Architectures [Text] / S. Guha, S. Majumder // International Journal on Web Service Computing. – 2020. – Vol. 11, № 2. – P. 1–16.
44. Kamiński L. Comparative review of selected Internet communication protocols [Text] / L. Kamiński [et al.] // Foundations of Computing and Decision Sciences. – 2023. – Vol. 48(1). – P. 39–56. – DOI: 10.2478/fcds-2023-0003.
45. Vogel M. Experiences on Migrating RESTful Web Services to GraphQL [Text] / M. Vogel, S. Weber, C. Zirpins // Service-Oriented Computing – ICSOC 2017 Workshops. – 2018. – P. 283–295. – DOI: 10.1007/978-3-319-91764-1_23.
46. Wittern E. Generating GraphQL-Wrappers for REST (-like) APIs [Text] / E. Wittern, A. Cha, J. A. Laredo // Web Engineering. – 2018. – P. 65–83.
47. Vohra N. Implementation of REST API vs GraphQL in microservice architecture [Text] / N. Vohra, I. B. K. Manuaba // 2022 International Conference on Information Management and Technology (ICIMTech). – 2022. – P. 45–50. – DOI: 10.1109/ICIMTech55935.2022.9915244.
48. Wittern E. An Empirical Study of GraphQL Schemas [Text] / E. Wittern [et al.] // Proceedings of the 17th International Conference on Service-Oriented Computing (ICSOC). – 2019. – Vol. 11895.
49. Cha A. A principled approach to GraphQL Query Cost Analysis [Text] / A. Cha [et al.] // Proceedings of the 28th ACM Joint Meeting on ESEC/FSE. – 2020. – P. 257–268.
50. Choosing Between REST and GraphQL for Microservices [Electronic resource] / AWS Architecture Blog. – 2021. – Access mode : URL:

https://aws.amazon.com/blogs/architecture/rest-vs-graphql-for-microservices/. – Date of Access: 22 November 2025.

51. Wycislik L. A Comparative Assessment of JVM Frameworks to Develop Microservices [Text] / L. Wycislik, L. Latusik, A. M. Kaminska // Applied Sciences. – 2023. – Vol. 13. – DOI: 10.3390/app13031343.
52. Torikian G. The GitHub GraphQL API [Electronic resource] / G. Torikian [et al.]. – 2016. – Access mode : URL: https://githubengineering.com/the-github-graphql-api/. – Date of Access: 22 November 2025.
53. Ireland S. M. GraphQL for the delivery of bioinformatics web APIs and application to ZincBind [Text] / S. M. Ireland, A. C. R. Martin // Bioinformatics Advances. – 2021. – Vol. 1(1). – DOI: 10.1093/bioadv/vbab023.
54. State of the API Industry: REST vs. GraphQL Trends [Electronic resource] / Postman API Report. – 2022. – Access mode : URL: https://www.postman.com/state-of-api/2024. – Date of Access: 22 November 2025.

ДОДАТКИ

ДОДАТОК А

Технічне завдання

ЗАТВЕРДЖУЮ
Перший проректор Українського
державного університету
науки і технологій
Анатолій РАДКЕВИЧ

«СИСТЕМА ПРОГРАМНОГО ІНТЕРФЕЙСУ ДЛЯ ВЕДЕННЯ БЛОГУ»

Технічне завдання
ЛИСТ ЗАТВЕРДЖЕННЯ
44165850.1546-01-ЛЗ

Представники
підприємства-розробника
Завідувач кафедри КІТ
Вадим ГОРЯЧКІН

Керівник розробки
Вадим ГОРЯЧКІН

Виконавець
Андрій ГРИГОРЕНКО

Нормоконтролер
Світлана ВОЛКОВА

2025

ЗАТВЕРДЖЕНО
44165850.1546-01-ЛЗ

СИСТЕМА ПРОГРАМНОГО ІНТЕРФЕЙСУ ДЛЯ ВЕДЕННЯ БЛОГУ

Технічне завдання

44165850.1546-01

Листів 21

2025

АНОТАЦІЯ

Документ 44165850.1546-01 «Система програмного інтерфейсу для ведення блогу. Технічне завдання» входить до складу програмної документації до програми, яка реалізує програмне забезпечення. У даному документі представлені призначення та область застосування програмної системи, технічні, техніко-економічні показники, що пред'являються до програми, терміни розробки проекту.

ЗМІСТ

ВСТУП

Додаток «СИСТЕМА ПРОГРАМНОГО ІНТЕРФЕЙСУ ДЛЯ ВЕДЕННЯ БЛОГУ» – серверний веб-застосунок (API), призначений для управління об’єктами домену «блог»: користувачами, публікаціями та коментарями. Мета – надати уніфікований програмний інтерфейс доступу до даних блогу двома способами: GraphQL (гнучкі запити) і REST (традиційна сумісність).

Основні завдання програми:

- керування сутностями блогу (створення, читання, оновлення, видалення);
- отримання вибірок публікацій (останні, опубліковані, популярні) з фільтрами та сортуванням;
- можливість подальшого розширення: автентифікація/авторизація, пагінація, метрики популярності, ієрархічні коментарі.

Ключові слова та терміни:

- GraphQL – мова запитів до API, що дозволяє клієнту визначати структуру відповіді;
- REST – архітектурний стиль побудови веб сервісів;
- N+1 – неефективне виконання численних додаткових SQL-запитів при вибірці зв’язків;
- публікація – пост користувача зі статусом і часовою міткою.
- API (інтерфейс програмування додатків) – це набір правил і протоколів, які дозволяють веб-додаткам та іншим програмам обмінюватися даними та функціональністю.

Причини виникнення: відсутність аналогів на ринку програмних продуктів.

Область застосування: загальна, навчальні та виробничі проекти, де потрібен гнучкий API для блогу/контент-платформи з використанням REST та GraphQL API.

1 ПІДСТАВА ДЛЯ РОЗРОБКИ

Підставою для розробки є наказ ректора Українського державного університету науки і технології Радкевич А.В. «Про затвердження тем та призначення керівників дипломних проектів» №1401ст. від 02 жовтня 2025 р.

Тема дипломної роботи – «Дослідження продуктивності GraphQL при використанні у веб-додатках».

Керівник – Вадим Миколайович Горячкін.

2 ПРИЗНАЧЕННЯ РОЗРОБКИ

2.1 Функціональне призначення:

Додаток «СИСТЕМА ПРОГРАМНОГО ІНТЕРФЕЙСУ ДЛЯ ВЕДЕННЯ БЛОГУ» дозволяє користувачам керувати сутностями блогу та будувати клієнтські застосунки (веб, мобільні, тощо).

2.2 Експлуатаційне призначення:

Додаток працює в режимі HTTP-сервісу на сервері/в контейнері, надаючи доступ через API з використанням інструментів розробника (GraphiQL, Postman) та клієнтських застосунків.

3 ВИМОГИ ДО ПРОГРАМНОГО ПРОДУКТУ

3.1 Вимоги до функціональних характеристик

Вимоги до функціональних характеристик наступні:

1) Ведення блогу:
   - програма повинна надавати API для створення, отримання, видалення та редагування користувачів, публікацій і коментарів;
   - мати перевірку вхідних аргументів на коректність;
   - необхідно забезпечити паритетність та відповідність між GraphQL та REST.
2) Зберігання стану:
   - програма повинна зберігати поточний стан в базі даних;
   - програма повинна забезпечити рівноправність при отриманні та збереженні даних незалежно від використаного API;
3) Вхідні дані:
   - програма приймає вхідні запити з використанням протоколу HTTP. Всі параметри повинні бути правильно сформовані для уникнення помилок;
   - для GraphQL: JSON із полями запиту, змінних та назвою операції;
   - для REST: JSON тіла зі схемою сутності, параметри запиту для фільтрів/лімітів;
4) Вихідні дані:
   - програма повертає дані у відповідь на вхідні запити з використанням протоколу HTTP;
   - для GraphQL: повертається поле data з об’єктами схем, поле errors за потреби;
   - для REST: повертається уніфікована структура відповіді; коди статусу HTTP відповідно до результату;
   - у разі помилки в команді або параметрах виводиться відповідне повідомлення про помилку з поясненням.

5) Організація програми:
    - програма побудована на базі архітектура модель – вид – контролер з додатковим шаром GraphQL (схема, типи, мутації).
    - програма обробляє вхідні дані за допомогою функції відповідної моделі та повертає стандартизовані відповіді;
    - вхідні команди та параметри перевіряються на коректність, і в разі відсутності обов'язкових параметрів виводиться повідомлення про помилку;
    - вихідні дані виводяться у вигляді текстових JSON повідомлень, завдяки чому можуть бути інтегровані в інші системи або використані в скриптах для автоматизації.

3.2 Вимоги до надійності

Вимоги до надійності наступні:

- одинична помилка оператора не повинна призводити до катастрофічних або серйозних наслідків;
- помилки користувачів або некритичні помилки обладнання не мають призводити до втрати даних;
- для вхідних даних потрібно забезпечити перевірку коректності введених даних;
- наявність архівної копії тексту програми на зовнішньому носії;
- у випадку збою – перезапустити команду.

3.3 Умови експлуатації

Умови експлуатації наступні:

- температура повітря має складати 21-25 °С;
- відносна вологість повітря 40-60%.

Програмою може користуватися людина, яка має досвід роботи на ПК та ознайомлена з керівництвом користувача.

3.4 Вимоги до складу і параметрів технічних засобів

Вимоги до складу і параметрів технічних засобів наступні:

- процесор з підтримкою архітектури x86-64 з частотою 1 ГГц або швидший;
- оперативна пам'ять: мінімум 512 M RAM;
- диск SSD або HDD: мінімум 500 МБ;
- мінімум один порт USB 2.0;
- мережевий адаптер з підтримкою протоколу TCP/IP;
- Wi-Fi роутер з підтримкою режиму 2.4 ГГц.

3.5 Вимоги до інформаційної і програмної сумісності

Вимоги до інформаційної і програмної сумісності наступні:

- операційна система: Windows 7, 8, 10, macOS 10.12 і вище, Linux;
- середовище розробки: Microsoft Visual Studio Code;
- мова програмування: Ruby;
- додаткові бібліотеки: встановлена бібліотека Bunle для розробки на Ruby on Rails.

3.6 Вимоги до маркування та упаковки

Приклад маркування приведений на рис. 1.

СИСТЕМА ПРОГРАМНОГО ІНТЕРФЕЙСУ
ДЛЯ ВЕДЕННЯ БЛОГУ

Розробник: Григоренко Андрій

УДУНТ, кафедра КІТ

2025

Рис. 1. Приклад маркування програми

Упаковка програмного продукту включаючи документацію повинна бути захищена від механічних та кліматичних пошкоджень, наприклад пластикова коробка для USB накопичувача.

3.7 Вимоги до транспортування та зберігання

Транспортування повинно забезпечити збереження програмного продукту та його цілісність, та забігати несанкціонованого доступу до нього. Програмний виріб міститься на флеш носії та повинен мати відповідну упаковку що має захист від механічних ушкоджень та атмосферного впливу (наприклад, пластиковий футляр).

4 ВИМОГИ ДО ПРОГРАМНОЇ ДОКУМЕНТАЦІЇ

До складу програмної документації мають входити:

- текст програми;
- керівництво користувача.

Вся документація до програми повинна задовольняти вимоги до програмної документації [1].

5 ТЕХНІКО-ЕКОНОМІЧНІ ПОКАЗНИКИ

Основна мета розробки техніко-економічного обґрунтування є надання фінансової оцінки передбачуваних витрат, а також оцінка рентабельності проекту і в кінцевому підсумку економічної доцільності його розробки та реалізації.

Початковий етап розрахунків – оцінка розміру програмного забезпечення. Згідно моделі COCOMO, розмір проекту S вимірюється в рядках коду LOC (KLOC), а трудовитрати в людино-місяцях.

$$E = a \cdot Sb \cdot EAF, \quad (5.1)$$

де E – витрати праці на проект (в людино-місяцях);

Sb – розмір коду (в KLOC);

EAF – фактор уточнення витрат (effort adjustment factor).

Для простих систем, a = 2,4; b = 1,05

За допомогою програми cloc [2] було виконано підрахунок розміру програмного коду. Загальний обсяг програмного коду становить 1054 рядків, див. рис. 5.1.

```
      46 text files.
      46 unique files.
       0 files ignored.

github.com/AlDanial/cloc v 2.06  T=0.01 s (3916.6 files/s, 120221.6 lines/s)
```

Language	files	blank	comment	code
Ruby	36	213	93	983
ERB	4	6	3	55
JavaScript	5	4	29	16
CSS	1	0	10	0
SUM:	46	223	135	1054

Рисунок 5.1 – Підрахунок розміру програмного коду

$$E = 2{,}4 \cdot 1{,}054^{1{,}05} \cdot 1 = 2{,}54$$

Отже, згідно моделі COCOMO, орієнтовні трудовитрати на проект складуть приблизно 2,54 людино-місяця. Нижче подано розрахунок вартості розробки. Основними статтями витрат є:

- базова заробітна плата
- відрахування на соціальні потреби;
- накладні та додаткові витрати;
- вартість персонального комп’ютера та програмних засобів.

Розрахуємо заробітну плату інженера-програміста, для оцінки основної заробітної плати. Згідно статистиці сайту вакансій Work.ua, середня зарплатня інженера-програміста на листопад 2025 року дорівнює 37 500 грн [3]. Розрахунок заробітної плати здійснюється у вигляді таблиці. 5.1.

Таблиця 5.1 – Фонд місячної заробітної плати

№ п/п	Посада виконавця	Оклад, грн/міс	Кількість		Сума зарплати, грн
			чол	місяців	
1	Інженер-програміст	37 500	1	2,54	95 250

Програмний продукт, описаний у проекті, був розроблений програмістом у період з 16.09.21 по 20.11.25, що становить 55 днів або приблизно 10 робочих тижнів. Витрати робочого часу прийняті за 40 годин на тиждень. Погодинна оплата праці кваліфікованого інженера-програміста становить 223,22 грн./год (згідно середній зарплаті та середньому робочому дню інженера-програміста, який складає 8 годин). Отже, робочий час витрачається:

$$t\text{розробки} = N\text{чол} \times N\text{тиж} \times N\text{год}, \quad (5.2)$$

де Nчол – кількість виконавців, чол;

Nтиж – тривалість розробки;

Nгод – витрати робочого часу, год;

tрозробки = 1 · 10 · 40 = 400 чол/год.

ОЗП визначається за формулою:

$$\text{ОЗП} = t\text{розробки} \cdot N \cdot KKB, \quad (5.3)$$

де tрозробки – витрати праці у чол/год;

N – погодинна ставка;

KKB – коефіцієнт кваліфікації програміста, обумовлений від стажу роботи з даної спеціальності. Коефіцієнт кваліфікації розробника (k) – ступінь підготовленості виконавця до дорученої йому роботі. Він визначається залежність від стажу праці та становить:

- для працюючих до 2 років- 0,75;
- від 2 до 3 років 1,0;
- від 3 до 5 років - 1,1-1,2;
- від 5 до 7 років - 1,3-1,4;
- понад 7 років - 1,5-1,6

В даному випадку KKB приймається 0,75. ОЗП складає:

$$\text{ОЗП} = 400 \cdot 223{,}22 \cdot 0{,}75 = 66966 \text{ грн.}$$

Відрахування на соціальні потреби встановлюються у відсотках від суми заробітної плати (22% [4]):

$$C\text{соц} = \text{ОЗП} \cdot 22\% \,/\, 100\% \quad (5.4)$$

$$C\text{соц} = 66966 \cdot 22\% \,/\, 100\% = 14\,732{,}52\text{грн}$$

Отримані результати за (5.3) та (5.4) підсумовуються. Вони складають 81 698,52грн. та визначають основні прямі витрати.

Накладні витрати враховують такі витрати для роботи над проектом як: опалення, електроенергія, амортизація будівель, зарплату адміністративного персоналу і т.д.

$$Снакл = (ОЗП + Ссоц) \cdot 35\% / 100\% \quad (5.5)$$

$$Снакл = (66966 + 14\,732{,}52) \cdot 35\% / 100\% = 28\,594{,}49 грн$$

Протягом усього терміну експлуатації нового обладнання компанія щороку витрачає певну суму грошей, пов’язану з її діяльністю.

Експлуатаційна вартість персонального комп'ютера визначається при розробці програмного забезпечення в залежності від вартості комп'ютера.

Операційні витрати включають:

- вартість витратних матеріалів;
- плата за ремонт;
- заробітна плата ремонтників;
- оренда приміщення;
- додаткові витрати - прибирання приміщення, охорона, оренда, комунальні послуги;
- амортизаційні витрати на персональні комп'ютери і програмне забезпечення;
- плата за електроенергію (Сел), визначається за такою формулою:

$$Сел = P \cdot B \cdot Tрозр, \quad (5.6)$$

де Р – потужність комп’ютера та допоміжних споживачів електричної енергії, приймається 0,5 кВт/год;

B – вартість 1 кВт/годин для непобутових споживачів, складає 10.77 грн [5];

$Tрозр$ – час роботи з ЕВМ, приймається рівним робочому часу

Витрати на електроенергію визначаються так:

$$Сел = 0{,}5 \cdot 10{,}77 \cdot 400 = 2\,154 грн$$

Витрати на витратні матеріали (Свм) протягом всього терміну експлуатації становлять приблизно 10% від вартості комп'ютеру. Вартість робочої станції приймається 25 000 грн. [6], термін експлуатації – 5 років [7]. Отже, можна визначити ці витрати за період створення програмного засобу:

$$Свм=Вком\cdot(NД / (Nексп\cdot365))\cdot10\%/100\%, \quad (5.7)$$

де $ВКОМ$ – вартість персонального комп'ютеру;

$NД$ – кількість днів розробки програмного продукту;

$Nексп$ – термін експлуатації персонального комп'ютеру.

Витрати на витратні матеріали визначаються так:

$$Свм=25\ 000 \cdot (55 / (5 \cdot 365)) \cdot 10/100 = 75{,}35грн$$

Зарплата ремонтника ($Срем$) визначається таким чином: системний інженер потрібен для ремонту 50 комп'ютерів. Припустимо, що його середньомісячна заробітна плата становить 20 000 грн. Тоді, виходячи з комп'ютера, його зарплата під час розробки програмного продукту становить:

$$Срем=Срем́ / NКОМ \cdot Тміс , \quad (5.8)$$

де $Срем́$ – середньомісячна заробітна плата;

$NКОМ$ – кількість комп'ютерів на одного ремонтника.

Тміс – час розробки програмного продукта, міс.

Заробітна плата ремонтника ($Срем$) буде складати:

$$Срем=20000 / 50 \cdot 2.5 = 1\ 000грн$$

За статистикою витрати на комплектуючи вироби (СКОМ) для ремонту персонального комп'ютера складає 10% від його вартості за термін його експлуатації, тобто рівні витратам на витратні матеріали:

$$C\text{КОМ}=C\text{вм}=75.35\text{грн}. \quad (5.9)$$

Амортизаційні відрахування для персональних комп'ютерів (АПК) базуються на припущенні, що період амортизації на даний момент дорівнює терміну експлуатації і складає 5 років.

$$\text{АПК}= \text{Вком}\cdot(N\text{Д} / (N\text{експ}\cdot 365))\cdot \quad (5.10)$$

$$\text{АПК}=25\,000 \cdot (55 / (5 \cdot 365)) = 753{,}5\text{грн}$$

Для функціонування персонального комп'ютера використовувалася операційна система Ubuntu Linux та редактор Visual Studio Code, які є безкоштовними.

АПЗ=0

Розрахунок амортизаційних відрахувань ПЗ зведено в таблицю. 5.2 Додаткові витрати (Cдод): прибирання приміщень, охорона, комунальні послуги важко оцінити точно, і прийнято як 50% зарплати інженерів-програмістів, що становить 18 750 гривень на місяць.

Оренду приміщень для однієї людини приймемо рівною 3000 гривень на місяць [8].Тобто за весь період розробки (Cор) – 9000 грн. Загальна вартість експлуатації ПК становить:

$$C\text{експ} = C\text{ел} + C\text{вм} + C\text{рем} + \text{АПК} + \text{АПЗ} + C\text{ор} + C\text{дод}; \quad (5.11)$$

$$C\text{експ} = 2\,154 + 75{,}35 + 1000 + 753{,}5 + 0 + 9000 + 18750 = 31732.85\text{грн}$$

Результати розрахунків зведено у табл. 5.3.

Таблиця 5.2 – Використане програмне забезпечення

Найменування програмного забезпечення	Вартість програмного забезпечення, грн	Джерело придбання	Амортизаційні відрахування, грн
Ubuntu Linux	0	https:// ubuntu.com/download/desktop	0

Visual Studio Code	0	https://visualstudio.microsoft.com	0
Всього:	0		0

Таблиця 5.3 – Експлуатаційні витрати на ПК і ПЗ

Найменування витрат	Витрати, грн
Витрати на електроенергію	2 154
Вартість витратних матеріалів	75,35
Витрати на ремонт	1000
Амортизація персонального комп’ютера	753,5
Амортизація програмного забезпечення	0
Оренда приміщення	9000
Додаткові витрати	18750
Всього	31732.85

Таким чином, витрати на створення програмного продукту складають:

$$Срозробки = ОЗП + Ссоц + Снакл + Сексп; \quad (5.12)$$

$$Срозробки = 66966 + 14\,732{,}52 + 28\,594{,}49 + 31732{,}85 = 142025.86 \text{ грн}$$

Розрахунок витрат зведено у табл. 5.4

Таблиця 5.4 – Кошторис витрат на розробку програмного засобу

Найменування витрат	Витрати, грн
Основна заробітна плата	66966
Відрахування на соціальні потреби	14 732,52
Накладні витрати	28 594,49
Експлуатаційні витрати	31732,85
Всього	142025.86

За отриманими значеннями техніко-економічних показників проекту складено кошторис витрат на розробку програмного. За результатами розрахунків, вартість розробки складає 142025.86 грн.

6 СТАДІЇ І ЕТАПИ РОЗРОБКИ

В табл. 6.1 приведені стадії та етапи розробки програмного продукту.

Таблиця 6.1. Стадії та етапи розробки

Стадія	Зміст робіт	Строки виконання
Технічне завдання	Постановка задачі, збір інформації, виріб та обґрунтування критеріїв розробки. Попередній вибір методів рішення задач. Визначення вимог до технічних засобів. Узгодження та затвердження технічного завдання.	01.11.25 - 10.11.25
Робочий проект	Програмування та відладка програми	17.11.25 - 23.11.25
	Тестування програми	24.11.25 - 30.11.25
	Розробка, узгодження та затвердження програмної документації	01.12.25 - 15.12.25

7 ПОРЯДОК КОНТРОЛЮ І ПРИЙМАННЯ

Контроль за виконанням роботи здійснює керівник дипломного проекту: Горячкін В. М.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Івченко Ю. М. Основи стандартизації програмних систем [Текст] : методичні вказівки до дипломного проектування та лабораторних робіт / уклад.: Ю. М. Івченко, В. І. Шинкаренко, В. Г. Івченко. – Дніпропетровськ : Вид-во Дніпропетр. нац. ун-ту залізн. трансп. ім. акад. В. Лазаряна, 2009. – 38 с.
2. AlDanial. Cloc (Count Lines of Code) [Електронний ресурс] : [програмне забезпечення] / GitHub. – Режим доступу : URL: https://github.com/AlDanial/cloc. – Дата звернення: 10.11.2025.
3. Зарплата інженера-програміста в Україні [Електронний ресурс] / Work.ua. – Режим доступу : URL: https://www.work.ua/salary-инженер-программист/. – Дата звернення: 10.11.2025.
4. Довідники. Єдиний соціальний внесок (ЄСВ) [Електронний ресурс] / Дебет-Кредит. – Режим доступу : URL: https://services.dtkt.ua/catalogues/indexes/13. – Дата звернення: 10.11.2025.
5. Тарифи на електроенергію для населення [Електронний ресурс] / YASNO. – Режим доступу : URL: https://yasno.com.ua/b2c-tariffs. – Дата звернення: 10.11.2025.
6. Ноутбуки [Електронний ресурс] : [електронний каталог] / Інтернет-магазин Rozetka. – Режим доступу : URL: https://rozetka.com.ua/ua/notebooks/c80004/. – Дата звернення: 10.11.2025.
7. Мінімально допустимі строки амортизації основних засобів та необоротних активів [Електронний ресурс] / Uteka. – Режим доступу : URL: https://uteka.ua/ua/publication/news-14-ezhednevnyj-buxgalterskij-obzor-39-minimalno-dopustimye-sroki-amortizacii-osnovnyx-sredstv-i-vneoborotnyx-aktivov. – Дата звернення: 10.11.2025.
8. Оренда офісу в Дніпрі [Електронний ресурс] / Сервіс оголошень OLX.ua. – Режим доступу : URL: https://www.olx.ua/uk/list/q-оренда-офісу-в-дніпрі/. – Дата звернення: 10.11.2025.

ДОДАТОК Б

Текст програми

ЗАТВЕРДЖУЮ
Перший проректор Українського
державного університету
науки і технологій
Анатолій РАДКЕВИЧ

СИСТЕМА ПРОГРАМНОГО ІНТЕРФЕЙСУ ДЛЯ ВЕДЕННЯ БЛОГУ

Текст програми
ЛИСТ ЗАТВЕРДЖЕННЯ
44165850.1546-01 12 01

Представники
підприємства-розробника
Завідувач кафедри КІТ
Вадим ГОРЯЧКІН

Керівник розробки
Вадим ГОРЯЧКІН

Виконавець
Андрій ГРИГОРЕНКО

Нормоконтролер
Світлана ВОЛКОВА

2025

ЗАТВЕРДЖЕНО
44165850.1546-01 12 01

«СИСТЕМА ПРОГРАМНОГО ІНТЕРФЕЙСУ ДЛЯ ВЕДЕННЯ БЛОГУ»

Текст програми

Листів 25

2025

АНОТАЦІЯ

Документ 44165850.1546-01 12 01 «СИСТЕМА ПРОГРАМНОГО ІНТЕРФЕЙСУ ДЛЯ ВЕДЕННЯ БЛОГУ. Текст програми» входить до складу програмної документації програмного продукту, спрямованого створення веб інтерфейсу ведення блогу для дослідження продуктивності GraphQL у веб додатках.

У даному документі представлений текст програм. Програма написана на мові Ruby. Об'єм пам'яті, що займає додаток, складає 400 Мб. Конфігурація комп'ютера стандартна, з обов’язковою наявністю мережевого адаптеру. Комплекс функціонує в середовищі Linux.

ЗМІСТ

1 ТЕКСТ ПРОГРАМИ

1.1 Модулі контролерів

1.1.1 app/controllers/api/v1/users_controller.rb

```
# frozen_string_literal: true

module Api
  module V1
    class UsersController < BaseController
      before_action :set_user, only: [:show, :update, :destroy, :posts, :comments, :published_posts, :recent_posts, :with_posts_and_comments]

      # GET /api/v1/users
      def index
        users = User.all

        render_success(
          users.map { |user| serialize_user(user) }
        )
      end

      # GET /api/v1/users/:id
      def show
        render_success(serialize_user_detailed(@user))
      end

      # POST /api/v1/users
      def create
        user = User.new(user_params)

        if user.save
          render_success(serialize_user_detailed(user), status: :created)
        else
          render_validation_errors(user)
        end
      end

      # PATCH/PUT /api/v1/users/:id
      def update
        if @user.update(user_params)
          render_success(serialize_user_detailed(@user))
        else
          render_validation_errors(@user)
        end
      end

      # DELETE /api/v1/users/:id
      def destroy
        @user.destroy
        render_success({ message: "User deleted successfully" })
      end

      # GET /api/v1/users/:id/posts
      def posts
        posts = @user.posts.includes(:user, :comments)

        posts = posts.published if params[:published_only] == 'true'
        posts = posts.limit(params[:limit]) if params[:limit].present?

        case params[:order_by]
        when 'published_at'
          posts = posts.by_publication_date
        else
          posts = posts.recent
        end

        render_success(
          posts.map { |post| serialize_post(post) }
        )
      end

      # GET /api/v1/users/:id/comments
      def comments
```

```
        comments =
@user.comments.includes(:user, :post).recent
        comments =
comments.limit(params[:limit]) if
params[:limit].present?

        render_success(
          comments.map { |comment|
serialize_comment(comment) }
        )
      end

      # GET
/api/v1/users/:id/published_posts
      def published_posts
        posts =
@user.published_posts.includes(:user,
:comments).recent
        posts = posts.limit(params[:limit]) if
params[:limit].present?

        render_success(
          posts.map { |post|
serialize_post(post) }
        )
      end

      # GET /api/v1/users/:id/recent_posts
      def recent_posts
        limit = params[:limit]&.to_i || 5
        posts = @user.recent_posts(limit)

        render_success(
          posts.map { |post|
serialize_post(post) }
        )
      end

      # GET
/api/v1/users/:id/with_posts_and_comments
      def with_posts_and_comments
        user_with_associations =
User.includes(posts: [:comments], comments:
[:post]).find(@user.id)
render_success(serialize_user_with_associations
(user_with_associations))
      end

      private

      def set_user
        @user = User.find(params[:id])
      end

      def user_params
        params.require(:user).permit(:name,
:email, :bio, :avatar_url)
      end

      def serialize_user(user)
        {
          id: user.id,
          name: user.name,
          email: user.email,
          bio: user.bio,
          avatar_url: user.avatar_url,
          display_name: user.display_name,
          posts_count: user.posts_count,
          published_posts_count:
user.published_posts_count,
          comments_count:
user.comments_count,
          created_at: user.created_at,
          updated_at: user.updated_at
        }
      end

      def serialize_user_detailed(user)
        serialize_user(user).merge(
          posts: user.posts.includes(:user,
:comments).map { |post| serialize_post(post) },
          comments:
user.comments.includes(:user, :post).map {
|comment| serialize_comment(comment) }
        )
      end
```

```
        def serialize_user_with_associations(user)
          {
            id: user.id,
            name: user.name,
            email: user.email,
            bio: user.bio,
            avatar_url: user.avatar_url,
            display_name: user.display_name,
            posts_count: user.posts_count,
            published_posts_count: user.published_posts_count,
            comments_count: user.comments_count,
            created_at: user.created_at,
            updated_at: user.updated_at,
            posts: user.posts.map { |post| serialize_post_with_comments(post) },
            comments: user.comments.map { |comment| serialize_comment_with_post(comment) }
          }
        end

        def serialize_post(post)
          {
            id: post.id,
            title: post.title,
            content: post.content,
            user_id: post.user_id,
            published: post.published,
            published_at: post.published_at,
            excerpt: post.excerpt,
            reading_time: post.reading_time,
            comments_count: post.comments_count,
            published_recently: post.published_recently?,
            created_at: post.created_at,
            updated_at: post.updated_at,
            user: {
              id: post.user.id,
              name: post.user.name,
              email: post.user.email
            }
          }
        end

        def serialize_post_with_comments(post)
          serialize_post(post).merge(
            comments: post.comments.map { |comment| serialize_comment(comment) }
          )
        end

        def serialize_comment(comment)
          {
            id: comment.id,
            content: comment.content,
            user_id: comment.user_id,
            post_id: comment.post_id,
            excerpt: comment.excerpt,
            created_at: comment.created_at,
            updated_at: comment.updated_at,
            user: {
              id: comment.user.id,
              name: comment.user.name,
              email: comment.user.email
            }
          }
        end

        def serialize_comment_with_post(comment)
          serialize_comment(comment).merge(
            post: {
              id: comment.post.id,
              title: comment.post.title,
              published: comment.post.published
            }
          )
        end
      end
    end
  end
```

1.1.2 app/controllers/api/v1/posts_controller.rb

```
# frozen_string_literal: true

module Api
  module V1
    class PostsController < BaseController
      before_action :set_post, only: [:show, :update, :destroy, :comments]

      # GET /api/v1/posts
      def index
        posts = Post.includes(:user, :comments)

        posts = posts.published if params[:published_only] == 'true'
        posts = posts.where(user_id: params[:user_id]) if params[:user_id].present?

        case params[:order_by]
        when 'published_at'
          posts = posts.by_publication_date
        else
          posts = posts.recent
        end

        posts = posts.limit(params[:limit]) if params[:limit].present?

        render_success(
          posts.map { |post| serialize_post(post) }
        )
      end

      # GET /api/v1/posts/:id
      def show

render_success(serialize_post_detailed(@post))
      end

      # POST /api/v1/posts
      def create
        user = User.find_by(id: post_params[:user_id])

        unless user
          render_error("User not found", status: :not_found)
          return
        end

        post = user.posts.build(post_params.except(:user_id))

        if post.save

render_success(serialize_post_simple(post), status: :created)
        else
          render_validation_errors(post)
        end
      end

      # PATCH/PUT /api/v1/posts/:id
      def update
        if @post.update(post_params.except(:user_id))

render_success(serialize_post_detailed(@post))
        else
          render_validation_errors(@post)
        end
      end

      # DELETE /api/v1/posts/:id
      def destroy
        @post.destroy
        render_success({ message: "Post deleted successfully" })
      end

      # GET /api/v1/posts/popular
      def popular
        limit = params[:limit]&.to_i || 10
        posts = Post.published.includes(:user, :comments).recent.limit(limit)
```

```
    render_success(
      posts.map { |post| serialize_post(post) }
    )
  end

  # GET /api/v1/posts/recent
  def recent
    limit = params[:limit]&.to_i || 10
    published_only = params[:published_only] != 'false' # Default to true

    posts = Post.includes(:user, :comments).recent
    posts = posts.published if published_only
    posts = posts.limit(limit)

    render_success(
      posts.map { |post| serialize_post(post) }
    )
  end

  # GET /api/v1/posts/:id/comments
  def comments
    comments = @post.comments.includes(:user, :post).recent
    comments = comments.limit(params[:limit]) if params[:limit].present?

    render_success(
      comments.map { |comment| serialize_comment(comment) }
    )
  end

  private

  def set_post
    @post = Post.includes(:user, :comments).find(params[:id])
  end

  def post_params
    params.require(:post).permit(:title, :content, :user_id, :published)
  end

  def serialize_post_simple(post)
    {
      id: post.id,
      title: post.title,
      content: post.content,
      user_id: post.user_id,
      published: post.published,
      published_at: post.published_at,
      created_at: post.created_at,
      updated_at: post.updated_at
    }
  end

  def serialize_post(post)
    {
      id: post.id,
      title: post.title,
      content: post.content,
      user_id: post.user_id,
      published: post.published,
      published_at: post.published_at,
      excerpt: post.excerpt,
      reading_time: post.reading_time,
      comments_count: post.comments_count,
      published_recently: post.published_recently?,
      created_at: post.created_at,
      updated_at: post.updated_at,
      user: {
        id: post.user.id,
        name: post.user.name,
        email: post.user.email,
        display_name: post.user.display_name
      }
    }
  end
```

```
        def serialize_post_detailed(post)
          serialize_post(post).merge(
            comments:
post.comments.includes(:user).map { |comment|
serialize_comment(comment) }
          )
        end

        def serialize_comment(comment)
          {
            id: comment.id,
            content: comment.content,
            user_id: comment.user_id,
            post_id: comment.post_id,
            excerpt: comment.excerpt,
            created_at: comment.created_at,
            updated_at: comment.updated_at,
            user: {
              id: comment.user.id,
              name: comment.user.name,
              email: comment.user.email,
              display_name:
comment.user.display_name
            }
          }
        end
      end
    end
  end
```

1.1.3 app/controllers/api/v1/comments_ controller.rb

```
# frozen_string_literal: true

module Api
  module V1
    class CommentsController <
BaseController
      before_action :set_comment,
only: [:show, :update, :destroy]

      # GET /api/v1/comments
      def index
        comments =
Comment.includes(:user, :post)

        comments =
comments.where(post_id:
params[:post_id]) if
params[:post_id].present?
        comments =
comments.where(user_id:
params[:user_id]) if
params[:user_id].present?

        case params[:order_by]
        when 'created_at_asc'
          comments =
comments.order(:created_at)
        else
          comments =
comments.recent
        end

        comments =
comments.limit(params[:limit]) if
params[:limit].present?

        render_success(
          comments.map { |comment|
serialize_comment(comment) }
        )
      end

      # GET /api/v1/comments/:id
      def show

render_success(serialize_comment_detail
ed(@comment))
      end

      # POST /api/v1/comments
      def create
        user = User.find_by(id:
comment_params[:user_id])
        post = Post.find_by(id:
comment_params[:post_id])
```

```
        errors = []
        errors << "User not found"
unless user
        errors << "Post not found"
unless post

        if errors.any?
          render_error("Validation
failed", errors: errors, status: :not_found)
          return
        end

        comment = Comment.new(
          content:
comment_params[:content],
          user: user,
          post: post
        )

        if comment.save

render_success(serialize_comment_detail
ed(comment), status: :created)
        else

render_validation_errors(comment)
        end
      end

      # PATCH/PUT
/api/v1/comments/:id
      def update
        if
@comment.update(content:
comment_params[:content])

render_success(serialize_comment_detail
ed(@comment))
        else

render_validation_errors(@comment)
        end
      end

      # DELETE
/api/v1/comments/:id
      def destroy
        @comment.destroy
        render_success({ message:
"Comment deleted successfully" })
      end

      private

      def set_comment
        @comment =
Comment.includes(:user,
:post).find(params[:id])
      end

      def comment_params

params.require(:comment).permit(:conte
nt, :user_id, :post_id)
      end

      def
serialize_comment(comment)
        {
          id: comment.id,
          content: comment.content,
          user_id: comment.user_id,
          post_id: comment.post_id,
          excerpt: comment.excerpt,
          created_at:
comment.created_at,
          updated_at:
comment.updated_at,
          user: {
            id: comment.user.id,
            name:
comment.user.name,
            email:
comment.user.email,
            display_name:
comment.user.display_name
          },
          post: {
            id: comment.post.id,
```

```
                    title: comment.post.title,
                    published:
comment.post.published
                  }
                }
              end

              def
serialize_comment_detailed(comment)

serialize_comment(comment).merge(
                post: {
                  id: comment.post.id,
                  title: comment.post.title,
                  content:
comment.post.content,
                  published:
comment.post.published,
                  published_at:
comment.post.published_at,
                  user: {
                    id: comment.post.user.id,
                    name:
comment.post.user.name,
                    email:
comment.post.user.email
                  }
                }
              )
            end
          end
        end
end
```

1.1.4 app/controllers/graphql_controller.rb

```
# frozen_string_literal: true

class GraphqlController <
ApplicationController
  # If accessing from outside this domain,
nullify the session
  # This allows for outside API access
while preventing CSRF attacks,
  # but you'll have to authenticate your
user separately
  protect_from_forgery with:
:null_session

  def execute
    variables =
prepare_variables(params[:variables])
    query = params[:query]
    operation_name =
params[:operationName]
    context = {
      # Query context goes here, for
example:
      # current_user: current_user,
    }
    result =
GraphqlTestSchema.execute(query, variables:
variables, context: context, operation_name:
operation_name)
    render json: result
  rescue StandardError => e
    raise e unless Rails.env.development?
    handle_error_in_development(e)
  end

  private

  # Handle variables in form data, JSON
body, or a blank value
  def prepare_variables(variables_param)
    case variables_param
    when String
      if variables_param.present?
        JSON.parse(variables_param) || {}
      else
        {}
      end
    when Hash
      variables_param
    when ActionController::Parameters
      variables_param.to_unsafe_hash #
GraphQL-Ruby will validate name and type of
incoming variables.
    when nil
```

```
            {}
          else
            raise ArgumentError, "Unexpected
parameter: #{variables_param}"
          end
        end

        def handle_error_in_development(e)
          logger.error e.message
          logger.error e.backtrace.join("\n")

          render json: { errors: [{ message:
e.message, backtrace: e.backtrace }], data: {} },
status: 500
        end
      end
```

1.2 Модулі моделей

1.2.1 app/models/comment.rb

```
      class Comment < ApplicationRecord
        belongs_to :user
        belongs_to :post

        validates :content, presence: true,
length: { minimum: 1, maximum: 1000 }

        scope :recent, -> { order(created_at:
:desc) }
        scope :for_post, ->(post) { where(post:
post) }

        def excerpt(length = 100)
          content.truncate(length)
        end
      end
```

1.2.2 app/models/post.rb

```
      class Post < ApplicationRecord
        belongs_to :user
        has_many :comments, dependent:
:destroy

        validates :title, presence: true, length: {
minimum: 5, maximum: 200 }
        validates :content, presence: true,
length: { minimum: 10 }
        validates :published_at, presence: true,
if: :published?

        before_validation :set_published_at, if: -
> { published? && published_at.blank? }
        before_save :clear_published_at, if: -> {
!published? && published_at.present? }

        scope :published, -> { where(published:
true) }
        scope :draft, -> { where(published:
false) }
        scope :recent, -> { order(created_at:
:desc) }
        scope :by_publication_date, -> {
order(published_at: :desc) }
        scope :with_comments, -> {
joins(:comments).distinct }

        def comments_count
          comments.count
        end

        def excerpt(length = 150)
          content.truncate(length)
        end

        def reading_time
          words = content.split.size
          (words / 200.0).ceil
        end

        def published_recently?(days = 7)
          published? && published_at &&
published_at > days.days.ago
        end

        private

        def set_published_at
          self.published_at = Time.current
        end
```

```
  def clear_published_at
    self.published_at = nil
  end
end
```

1.2.3 app/models/user.rb

```
class User < ApplicationRecord
  has_many :posts, dependent: :destroy
  has_many :comments, dependent:
:destroy
  has_many :published_posts, -> {
where(published: true) }, class_name: 'Post'

  validates :name, presence: true, length:
{ minimum: 2, maximum: 50 }
  validates :email, presence: true,
uniqueness: true, format: { with:
URI::MailTo::EMAIL_REGEXP }
  validates :bio, length: { maximum: 500
}
  validates :avatar_url, format: { with:
URI::DEFAULT_PARSER.make_regexp(%w[ht
tp https]), allow_blank: true }

  scope :recent, -> { order(created_at:
:desc) }
  scope :with_posts, -> {
joins(:posts).distinct }
  scope :active_users, -> {
joins(:posts).where(posts: { published: true
}).distinct }

  def posts_count
    posts.count
  end

  def published_posts_count
    published_posts.count
  end

  def comments_count
    comments.count
  end

  def recent_posts(limit = 5)
    posts.published.order(created_at:
:desc).limit(limit)
  end

  def display_name
    name.presence || email.split('@').first
  end
end
```

1.3 Модулі GraphQL

1.3.1 app/graphql/mutations/create_co
mment.rb

```
# frozen_string_literal: true

module Mutations
  class CreateComment < BaseMutation
    # Return fields
    field :comment,
Types::CommentType, null: true
    field :errors, [String], null: false

    # Arguments
    argument :content, String, required:
true
    argument :user_id, ID, required: true
    argument :post_id, ID, required: true

    def resolve(content:, user_id:, post_id:)
      user = User.find_by(id: user_id)
      post = Post.find_by(id: post_id)

      errors = []
      errors << "User not found" unless
user
      errors << "Post not found" unless post

      if errors.any?
        return {
          comment: nil,
          errors: errors
        }
      end

      comment = Comment.new(
```

```
            content: content,
            user: user,
            post: post
          )

          if comment.save
            {
              comment: comment,
              errors: []
            }
          else
            {
              comment: nil,
              errors:
comment.errors.full_messages
            }
          end
        end
      end
    end
```

1.3.2 app/graphql/mutations/create_post.rb

```
# frozen_string_literal: true

module Mutations
  class CreatePost < BaseMutation
    # Return fields
    field :post, Types::PostType, null: true
    field :errors, [String], null: false

    # Arguments
    argument :title, String, required: true
    argument :content, String, required:
true
    argument :user_id, ID, required: true
    argument :published, Boolean,
required: false, default_value: false

    def resolve(title:, content:, user_id:,
published: false)
      user = User.find_by(id: user_id)

      unless user
        return {
          post: nil,
          errors: ["User not found"]
        }
      end

      post = user.posts.build(
        title: title,
        content: content,
        published: published
      )

      if post.save
        {
          post: post,
          errors: []
        }
      else
        {
          post: nil,
          errors: post.errors.full_messages
        }
      end
    end
  end
end
```

1.3.3 app/graphql/mutations/create_user.rb

```
# frozen_string_literal: true

module Mutations
  class CreateUser < BaseMutation
    # Return fields
    field :user, Types::UserType, null: true
    field :errors, [String], null: false

    # Arguments
    argument :name, String, required: true
    argument :email, String, required: true
    argument :bio, String, required: false
    argument :avatar_url, String, required:
false

    def resolve(name:, email:, bio: nil,
avatar_url: nil)
```

```
            user = User.new(
              name: name,
              email: email,
              bio: bio,
              avatar_url: avatar_url
            )

            if user.save
              {
                user: user,
                errors: []
              }
            else
              {
                user: nil,
                errors: user.errors.full_messages
              }
            end
          end
        end
      end
```

1.3.4 app/graphql/types/comment_type.rb

```
      # frozen_string_literal: true

      module Types
        class CommentType < Types::BaseObject
          field :id, ID, null: false
          field :content, String, null: false
          field :user_id, Integer, null: false
          field :post_id, Integer, null: false
          field :created_at, GraphQL::Types::ISO8601DateTime, null: false
          field :updated_at, GraphQL::Types::ISO8601DateTime, null: false

          field :excerpt, String, null: false do
            argument :length, Integer, required: false, default_value: 100
          end

          field :user, Types::UserType, null: false
          field :post, Types::PostType, null: false

          def excerpt(length:)
            object.excerpt(length)
          end

          def user
            dataloader.with(Sources::ActiveRecordAssociation, :user).load(object)
          end

          def post
            dataloader.with(Sources::ActiveRecordAssociation, :post).load(object)
          end
        end
      end
```

1.3.5 app/graphql/types/post_type.rb

```
      # frozen_string_literal: true

      module Types
        class PostType < Types::BaseObject
          field :id, ID, null: false
          field :title, String, null: false
          field :content, String, null: false
          field :user_id, Integer, null: false
          field :published, Boolean, null: false
          field :published_at, GraphQL::Types::ISO8601DateTime
          field :created_at, GraphQL::Types::ISO8601DateTime, null: false
          field :updated_at, GraphQL::Types::ISO8601DateTime, null: false

          field :excerpt, String, null: false do
            argument :length, Integer, required: false, default_value: 150
          end
          field :reading_time, Integer, null: false
          field :comments_count, Integer, null: false
```

```
        field :published_recently, Boolean,
null: false do
          argument :days, Integer, required:
false, default_value: 7
        end

        field :user, Types::UserType, null:
false
        field :comments,
[Types::CommentType], null: false

        def excerpt(length:)
          object.excerpt(length)
        end

        def reading_time
          object.reading_time
        end

        def comments_count
          object.comments_count
        end



        def published_recently(days:)
          object.published_recently?(days)
        end

        def user

dataloader.with(Sources::ActiveRecordAssociati
on, :user).load(object)
        end

        def comments

dataloader.with(Sources::ActiveRecordAssociati
on, :comments).load(object)
        end
      end
    end
```

1.3.6 app/graphql/types/user_type.rb

```
# frozen_string_literal: true

module Types
  class UserType < Types::BaseObject
    field :id, ID, null: false
    field :name, String, null: false
    field :email, String, null: false
    field :bio, String
    field :avatar_url, String
    field :created_at,
GraphQL::Types::ISO8601DateTime, null: false
    field :updated_at,
GraphQL::Types::ISO8601DateTime, null: false

    field :display_name, String, null: false
    field :posts_count, Integer, null: false
    field :published_posts_count, Integer,
null: false
    field :comments_count, Integer, null:
false

    field :posts, [Types::PostType], null:
false
    field :published_posts,
[Types::PostType], null: false
    field :comments,
[Types::CommentType], null: false
    field :recent_posts, [Types::PostType],
null: false do
      argument :limit, Integer, required:
false, default_value: 5
    end

    def display_name
      object.display_name
    end

    def posts_count
      object.posts.count
    end

    def published_posts_count
      object.published_posts.count
    end

    def comments_count
      object.comments.count
```

```
        end

        def posts

dataloader.with(Sources::ActiveRecordAssociati
on, :posts).load(object)
        end

        def published_posts

dataloader.with(Sources::ActiveRecordAssociati
on, :published_posts).load(object)
        end

        def comments

dataloader.with(Sources::ActiveRecordAssociati
on, :comments).load(object)
        end

        def recent_posts(limit:)
          object.recent_posts(limit)
        end
      end
    end
```

1.4 Конфігураційні модулі

1.4.1 db/schema.rb

```
    ActiveRecord::Schema[8.0].define(versio
n: 2025_09_23_165802) do
      # These are extensions that must be
enabled in order to support this database
      enable_extension "pg_catalog.plpgsql"

      create_table "comments", force:
:cascade do |t|
        t.text "content", null: false
        t.bigint "user_id", null: false
        t.bigint "post_id", null: false
        t.datetime "created_at", null: false
        t.datetime "updated_at", null: false
        t.index ["created_at"], name:
"index_comments_on_created_at"
        t.index ["post_id", "created_at"], name:
"index_comments_on_post_id_and_created_at"
        t.index ["post_id"], name:
"index_comments_on_post_id"
        t.index ["user_id", "created_at"], name:
"index_comments_on_user_id_and_created_at"
        t.index ["user_id"], name:
"index_comments_on_user_id"
      end

      create_table "posts", force: :cascade do
|t|
        t.string "title", null: false
        t.text "content", null: false
        t.bigint "user_id", null: false
        t.boolean "published", default: false,
null: false
        t.datetime "published_at"
        t.datetime "created_at", null: false
        t.datetime "updated_at", null: false
        t.index ["created_at"], name:
"index_posts_on_created_at"
        t.index ["published", "published_at"],
name:
"index_posts_on_published_and_published_at"
        t.index ["published"], name:
"index_posts_on_published"
        t.index ["published_at"], name:
"index_posts_on_published_at"
        t.index ["user_id", "published"], name:
"index_posts_on_user_id_and_published"
        t.index ["user_id"], name:
"index_posts_on_user_id"
      end

      create_table "users", force: :cascade do
|t|
        t.string "name", null: false
        t.string "email", null: false
        t.text "bio"
        t.string "avatar_url"
        t.datetime "created_at", null: false
        t.datetime "updated_at", null: false
        t.index ["created_at"], name:
"index_users_on_created_at"
        t.index ["email"], name:
"index_users_on_email", unique: true
```

```
      t.index ["name"], name:
"index_users_on_name"
    end

    add_foreign_key "comments", "posts"
    add_foreign_key "comments", "users"
    add_foreign_key "posts", "users"
  end
```

1.4.2 db/seeds.rb

```
  # This file should ensure the existence of
records required to run the application in every
environment (production,
  # development, test). The data can then
be loaded with the bin/rails db:seed command
(or created alongside the database with
db:setup).

  # Clear existing data in development
  if Rails.env.development?
    Comment.destroy_all
    Post.destroy_all
    User.destroy_all
  end

  # Create sample users for GraphQL
performance testing
  puts "Creating users..."
  users = []

  10.times do |i|
    user = User.create!(
      name: "User #{i + 1}",
      email: "user#{i + 1}@example.com",
      bio: "This is the bio for user #{i + 1}.
Lorem ipsum dolor sit amet, consectetur
adipiscing elit.",
      avatar_url:
"https://via.placeholder.com/150/#{['FF0000',
'00FF00', '0000FF', 'FFFF00', 'FF00FF',
'00FFFF'].sample}"
    )
    users << user
  end

  puts "Created #{users.count} users"

  # Create sample posts
  puts "Creating posts..."
  posts = []

  users.each do |user|
    # Each user creates 3-8 posts
    rand(3..8).times do |i|
      published = [true, false].sample
      post = user.posts.create!(
        title: "Post #{i + 1} by
#{user.name}",
        content: "This is the content for post
#{i + 1} by #{user.name}. " * rand(10..50),
        published: published,
        published_at: published ?
rand(30.days).seconds.ago : nil,
      )
      posts << post
    end
  end

  puts "Created #{posts.count} posts"

  # Create sample comments
  puts "Creating comments..."
  comment_count = 0

  posts.select(&:published?).each do |post|
    # Each published post gets 0-15
comments
    rand(0..15).times do
      commenter = users.sample
      comment = post.comments.create!(
        user: commenter,
        content: "This is a comment by
#{commenter.name} on #{post.title}. " *
rand(1..5)
      )
      comment_count += 1
    end
  end
```

```
puts "Created #{comment_count}
comments"

# Print summary
puts "\n=== SEED DATA SUMMARY
==="
puts "Users: #{User.count}"
puts "Posts: #{Post.count}"
puts "  - Published:
#{Post.published.count}"
puts "  - Drafts: #{Post.draft.count}"
puts "Comments: #{Comment.count}"
puts "\nSeed data created successfully!"
```

1.4.3 test.sh

```
#!/bin/bash

# --- Configuration ---
# Check if k6 script path is
provided as command-line argument
if [ -z "$1" ]; then
    echo "Usage: $0 <k6-script-
path>"
    echo "Example: $0 k6-
tests/graphql1.js"
    exit 1
fi

K6_SCRIPT_NAME="$1"

# Check if the k6 script file exists
if [ ! -f "$K6_SCRIPT_NAME" ];
then
    echo "Error: k6 script file
'$K6_SCRIPT_NAME' not found."
    exit 1
fi

RAILS_LOG_FILE="rails_monit
oring.log"
OUTPUT_CSV_FILE="performa
nce_results.csv"

# --- Main Script Logic ---
echo "Starting the Rails server in
the background..."
# Start Rails server and redirect
its output to /dev/null to keep the
terminal clean
rails s > /dev/null 2>&1 &
# Capture the Process ID (PID) of
the last background command
RAILS_PID=$!

# Wait for Rails server to be ready
by checking if it responds to requests
echo "Waiting for Rails server to
be ready..."
MAX_WAIT=30
WAIT_COUNT=0
SERVER_READY=false

while [ $WAIT_COUNT -lt
$MAX_WAIT ]; do
    # Check if the process is still
running
    if ! kill -0 $RAILS_PID
2>/dev/null; then
        echo "Error: Rails server
failed to start."
        exit 1
    fi

    # Try to connect to the server
    if curl -s -o /dev/null -w
"%{http_code}" http://localhost:3000 >
/dev/null 2>&1; then
        SERVER_READY=true
        break
    fi

    sleep 1

WAIT_COUNT=$((WAIT_COUNT +
1))
done
```

```
    if [ "$SERVER_READY" = false ]; then
        echo "Error: Rails server did not become ready within $MAX_WAIT seconds."
        kill $RAILS_PID 2>/dev/null
        exit 1
    fi

    echo "Rails server started with PID: $RAILS_PID and is ready to accept requests"

    echo "Starting monitoring with pidstat..."
    # Start pidstat to monitor CPU (-u) and memory (-r) every 1 second
    # The output is redirected to our log file
    pidstat -p $RAILS_PID -u -r 1 > $RAILS_LOG_FILE &
    PIDSTAT_PID=$!

    echo "Running k6 load test..."
    # Run the k6 script. The script will pause here until k6 is finished.
    k6 run $K6_SCRIPT_NAME

    echo "k6 test finished."

    echo "Stopping monitoring and Rails server..."
    # Stop the pidstat monitor
    kill $PIDSTAT_PID
    # Stop the Rails server
    kill $RAILS_PID

    # Wait a moment for processes to terminate gracefully
    sleep 2
    echo "Cleanup complete."

    echo "Processing log file to create CSV..."

    # Use 'awk' to parse the log file and create a clean CSV
    # pidstat outputs CPU and memory metrics on alternating lines
    # We need to match them by timestamp and combine into one CSV row
    awk '
    BEGIN {
        print "Elapsed_Time(s),CPU_Usage(%),RAM_Usage(KB)";
        start_time = "";
    }
    # Process lines with PID in column 4 (data lines)
    $4 ~ /^[0-9]+$/ {
        timestamp = $1 " " $2;
        # CPU lines have 11 fields, %usr is in column 5
        if (NF == 11) {
            cpu[timestamp] = $5;
            timestamps[timestamp] = 1;
        }
        # Memory lines have 10 fields, RSS is in column 8
        else if (NF == 10) {
            ram[timestamp] = $8;
            timestamps[timestamp] = 1;
        }
    }
    # At the end, output all matched timestamp pairs in sorted order
    END {
        # Collect and sort all timestamps
        count = 0;
        for (ts in timestamps) {
            sorted_ts[++count] = ts;
        }
        n = asort(sorted_ts);

        # Get the start time from the first timestamp
        if (n > 0) {
```

```
                start_time = sorted_ts[1];
            }

            # Output data with elapsed
    seconds
            for (i = 1; i <= n; i++) {
                ts = sorted_ts[i];
                if (ts in cpu && ts in ram) {
                    # Calculate elapsed
    seconds (i-1 since we start from 0)
                    elapsed = i - 1;
                    print elapsed "," cpu[ts] ","
    ram[ts];
                }
            }
        }
        ' $RAILS_LOG_FILE >
    $OUTPUT_CSV_FILE

        echo "-------------------------------
    -----------------"
        echo "Success! Performance test
    completed."
        echo "K6 Script:
    $K6_SCRIPT_NAME"
        echo "Results saved to:
    $OUTPUT_CSV_FILE"
        echo "You can now import this
    file into Excel, Google Sheets, or any
    graphing tool."
    echo "-------------------------------------
--------"
```

1.4.4 k6-tests/graphql_create_post_min.js

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

    export const options = {
      vus: 10,
      duration: '30s',
    };

    export default function () {
      const mutation = `
        mutation CreatePost($input:
CreatePostInput!) {
          createPost(input: $input) {
            post {
              id
            }
            errors
          }
        }
      `;

      const variables = {
        input: {
          title: `Test Post ${Date.now()}`,
          content: 'This is a test post content
created by k6 load testing tool. It contains
enough text to meet validation requirements.',
          userId: '22',
          published: true
        }
      };

      const headers = {
        'Content-Type': 'application/json',
      };

      const res = http.post(
        'http://localhost:3000/graphql',
        JSON.stringify({ query: mutation,
variables: variables }),
        { headers: headers }
      );

      check(res, {
        'status was 200': (r) => r.status == 200,
        'no errors': (r) => {
          const body = JSON.parse(r.body);
          return !body.errors &&
body.data.createPost.errors.length === 0;
        }
      });

      sleep(1);
    }
```

1.4.5 k6-tests/graphql_create_post.js

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

export const options = {
  vus: 10,
  duration: '30s',
};

export default function () {
  const mutation = `
    mutation CreatePost($input: CreatePostInput!) {
      createPost(input: $input) {
        post {
          id
          title
          content
          userId
          published
          publishedAt
          createdAt
          updatedAt
        }
        errors
      }
    }
  `;

  const variables = {
    input: {
      title: `Test Post ${Date.now()}`,
      content: 'This is a test post content created by k6 load testing tool. It contains enough text to meet validation requirements.',
      userId: '22',
      published: true
    }
  };

  const headers = {
    'Content-Type': 'application/json',
  };

  const res = http.post(
    'http://localhost:3000/graphql',
    JSON.stringify({ query: mutation, variables: variables }),
    { headers: headers }
  );

  check(res, {
    'status was 200': (r) => r.status == 200,
    'no errors': (r) => {
      const body = JSON.parse(r.body);
      return !body.errors && body.data.createPost.errors.length === 0;
    }
  });

  sleep(1);
}
```

1.4.6 k6-tests/graphql_posts_full.js

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

export const options = {
  vus: 10,
  duration: '30s',
};

export default function () {
  const query = `
    query {
      posts {
        id
        title
        content
        userId
        published
        publishedAt
        excerpt
        readingTime
        commentsCount
        publishedRecently
        updatedAt
        createdAt
        user {
          id
          name
```

```
        email
        displayName
      }
    }
  }
`;

const headers = {
  'Content-Type': 'application/json',
};

const res =
http.post('http://localhost:3000/graphql',
JSON.stringify({ query: query }), { headers:
headers });
check(res, { 'status was 200': (r) =>
r.status == 200 });
sleep(1);
}
```

1.4.7 k6-tests/graphql_posts_min.js

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

export const options = {
  vus: 10,
  duration: '30s',
};

export default function () {
  const query = `
    query {
      posts {
        id
        title
      }
    }
  `;

  const headers = {
    'Content-Type': 'application/json',
  };

  const res =
http.post('http://localhost:3000/graphql',
JSON.stringify({ query: query }), { headers:
headers });
  check(res, { 'status was 200': (r) =>
r.status == 200 });
  sleep(1);
}
```

1.4.8 k6-tests/graphql_users_full.js

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

export const options = {
  vus: 10,
  duration: '30s',
};

export default function () {
  const query = `
    query {
      users {
        id
        name
        email
        bio
        avatarUrl
        displayName
        updatedAt
        createdAt
      }
    }
  `;

  const headers = {
    'Content-Type': 'application/json',
  };

  const res =
http.post('http://localhost:3000/graphql',
JSON.stringify({ query: query }), { headers:
headers });
  check(res, { 'status was 200': (r) =>
r.status == 200 });
  sleep(1);
}
```

1.4.9 k6-tests/graphql_users_id_name.js

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

export const options = {
  vus: 10,
  duration: '30s',
};

export default function () {
  const query = `
    query {
      users {
        id
        name
      }
    }
  `;

  const headers = {
    'Content-Type': 'application/json',
  };

  const res =
http.post('http://localhost:3000/graphql',
JSON.stringify({ query: query }), { headers:
headers });
  check(res, { 'status was 200': (r) =>
r.status == 200 });
  sleep(1);
}
```

1.4.10 k6-tests/rest_create_post.js

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

export const options = {
  vus: 10,
  duration: '30s',
};

export default function () {
  const payload = JSON.stringify({
    post: {
      title: `Test Post ${Date.now()}`,
      content: 'This is a test post content
created by k6 load testing tool. It contains
enough text to meet validation requirements.',
      user_id: 22,
      published: true
    }
  });

  const headers = {
    'Content-Type': 'application/json',
  };

  const res =
http.post('http://localhost:3000/api/v1/posts',
payload, { headers: headers });

  check(res, {
    'status was 201': (r) => r.status == 201,
    'success response': (r) => {
      const body = JSON.parse(r.body);
      return body.success === true;
    }
  });

  sleep(1);
}
```

1.4.11 k6-tests/rest_posts.js

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

export const options = {
  vus: 10, // 10 virtual users
  duration: '30s',
};

export default function () {
  const res =
http.get('http://localhost:3000/api/v1/posts');
  check(res, { 'status was 200': (r) =>
r.status == 200 });
  sleep(1);
}
```

1.4.12 k6-tests/rest_users_full.js

```
import http from 'k6/http';
```

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

export const options = {
  vus: 10, // 10 virtual users
  duration: '30s',
};

export default function () {
  const res =
http.get('http://localhost:3000/api/v1/users');
  check(res, { 'status was 200': (r) =>
r.status == 200 });
  sleep(1);
}
```

ДОДАТОК В

Керівництво користувача

ЗАТВЕРДЖУЮ
Перший проректор Українського
державного університету
науки і технологій
Анатолій РАДКЕВИЧ

СИСТЕМА ПРОГРАМНОГО ІНТЕРФЕЙСУ ДЛЯ ВЕДЕННЯ БЛОГУ

Текст програми
ЛИСТ ЗАТВЕРДЖЕННЯ
44165850.1546-01 І3 01

Представники
підприємства-розробника
Завідувач кафедри КІТ
Вадим ГОРЯЧКІН

Керівник розробки
Вадим ГОРЯЧКІН

Виконавець
Андрій ГРИГОРЕНКО

Нормоконтролер
Світлана ВОЛКОВА

2025

ЗАТВЕРДЖЕНО
44165850.1546-01 І3 01

СИСТЕМА ПРОГРАМНОГО ІНТЕРФЕЙСУ ДЛЯ ВЕДЕННЯ БЛОГУ

Керівництво користувача

Листів 12

2025

АНОТАЦІЯ

Документ 44165850.1546-01 І3 01 «СИСТЕМА ПРОГРАМНОГО ІНТЕРФЕЙСУ ДЛЯ ВЕДЕННЯ БЛОГУ. Керівництво користувача» входить до складу програмної документації програмного продукту, спрямованого створення веб інтерфейсу ведення блогу для дослідження продуктивності GraphQL у веб додатках.

У даному документі представлено керівництво користувача.

ЗМІСТ

ВСТУП

Розроблений веб-додаток є інструментальним стендом для дослідження продуктивності GraphQL та порівняння його з традиційним REST API в однакових умовах. Стенд реалізовано на базі Ruby on Rails і побудовано над спільною доменною моделлю блогу (користувачі, пости, коментарі) та спільною базою даних PostgreSQL, що забезпечує коректність і відтворюваність порівняння двох підходів доступу до даних.

Додаток був розрахований на функціонування в операційних системах Unix.

Застосунок надає:

GraphQL API через єдиний ендпоінт /graphql із підтримкою запитів і мутацій, вкладених вибірок та оптимізації N+1 через GraphQL::Dataloader;

REST API через ендпоінти /api/v1/* з уніфікованим форматом JSON-відповідей;

інструменти для експериментів: генерацію тестових даних (seed), сценарії навантаження k6 для мінімальних/повних вибірок і збір метрик CPU/RAM з експортом результатів у performance_results.csv для подальшого аналізу.

Користувачу не потрібні навички розробки або внесення змін у програмний код, однак для роботи потрібні базові технічні навички: виконання HTTP-запитів через GraphiQL/Postman/cURL та, за потреби, запуск команд у терміналі для навантажувальних тестів і перегляду результатів.

1 УМОВИ ЗАСТОСУВАННЯ

1.1 Вимоги до складу і параметрів технічних засобів:

- процесор: x86-64, не менше 2 ядер (рекомендовано 4);
- оперативна пам’ять: не менше 4 ГБ (рекомендовано 8 ГБ і більше);
- накопичувач: не менше 5 ГБ вільного місця;
- підтримка мереж - підтримка Ethernet/ WiFi.

1.2. Вимоги до інформаційного та програмного середовища:

- операційна система: Unix-подібні;
- інтерпретатор: Ruby 3.4.2;
- СУБД: PostgreSQL (рекомендовано 14+), доступна для Rails;
- інструменти експериментів: k6, pidstat, awk/bash.

2 ПІДГОТОВКА ДО ВИКОРИСТАННЯ

Встановлення додаткового ПЗ для виконання запитів не потрібне: достатньо браузера (для GraphiQL у режимі розробки) або будь-якого HTTP-клієнта (Postman/Insomnia/cURL) для виклику REST та GraphQL ендпоінтів.

Якщо додаток запускається локально з вихідного коду, необхідно попередньо підготувати програмне середовище:

- Відкрити термінал у каталозі проекту та активувати середовище Ruby;
- Встановити залежності застосунку за допомогою команди bundle install
- Ініціалізувати базу даних за допомогою команди bundle exec rails db:setup
- Запустити сервер застосунку: bundle exec rails server -p 3000.

Після успішного запуску сервер буде доступний за адресою http://localhost:3000, яка є точкою входу для подальшої роботи з API.

3 ОПИС ЕКРАНІВ ТА ОПЕРАЦІЙ

Оскільки застосунок є API-орієнтованим та не має окремого графічного інтерфейсу, взаємодія користувача відбувається через екран HTTP-клієнта (Postman / браузер / cURL) для REST і GraphQL, а також екран термінала для запуску навантажувальних тестів.

Операції з REST API виконуються через HTTP-клієнт. Приклад результату операції переліку користувачів за допомогою бразуера представлено на рис. 4.1

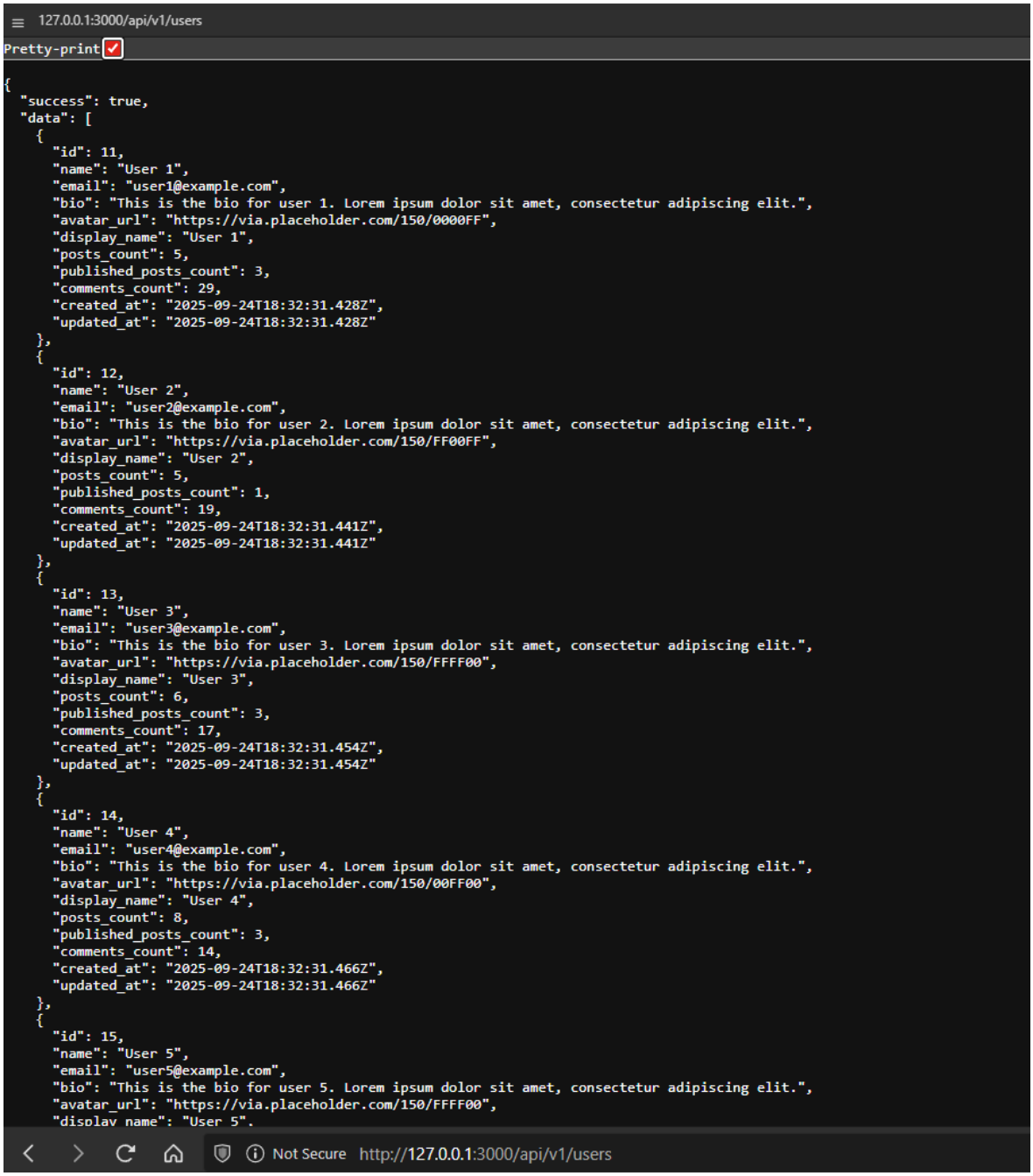

Рисунок 4.1 – отримання переліку користувачів

Доступні точки API складаються з:

- отримання списку ресурсів: виконати GET /api/v1/users або GET /api/v1/posts або GET /api/v1/comments;
- отримання ресурсу за id: GET /api/v1/users/:id (аналогічно для posts/comments);
- створення ресурсу: POST /api/v1/... з JSON-тілом (user, post або comment);
- оновлення ресурс: PATCH /api/v1/.../:id з JSON-тілом;
- видалення ресурсу: DELETE /api/v1/.../:id.

Операції з GraphQL виконуються за допомогою інтерфейсу graphiql, який находиться за адресою /graphiql та представлений на рис. 4.2

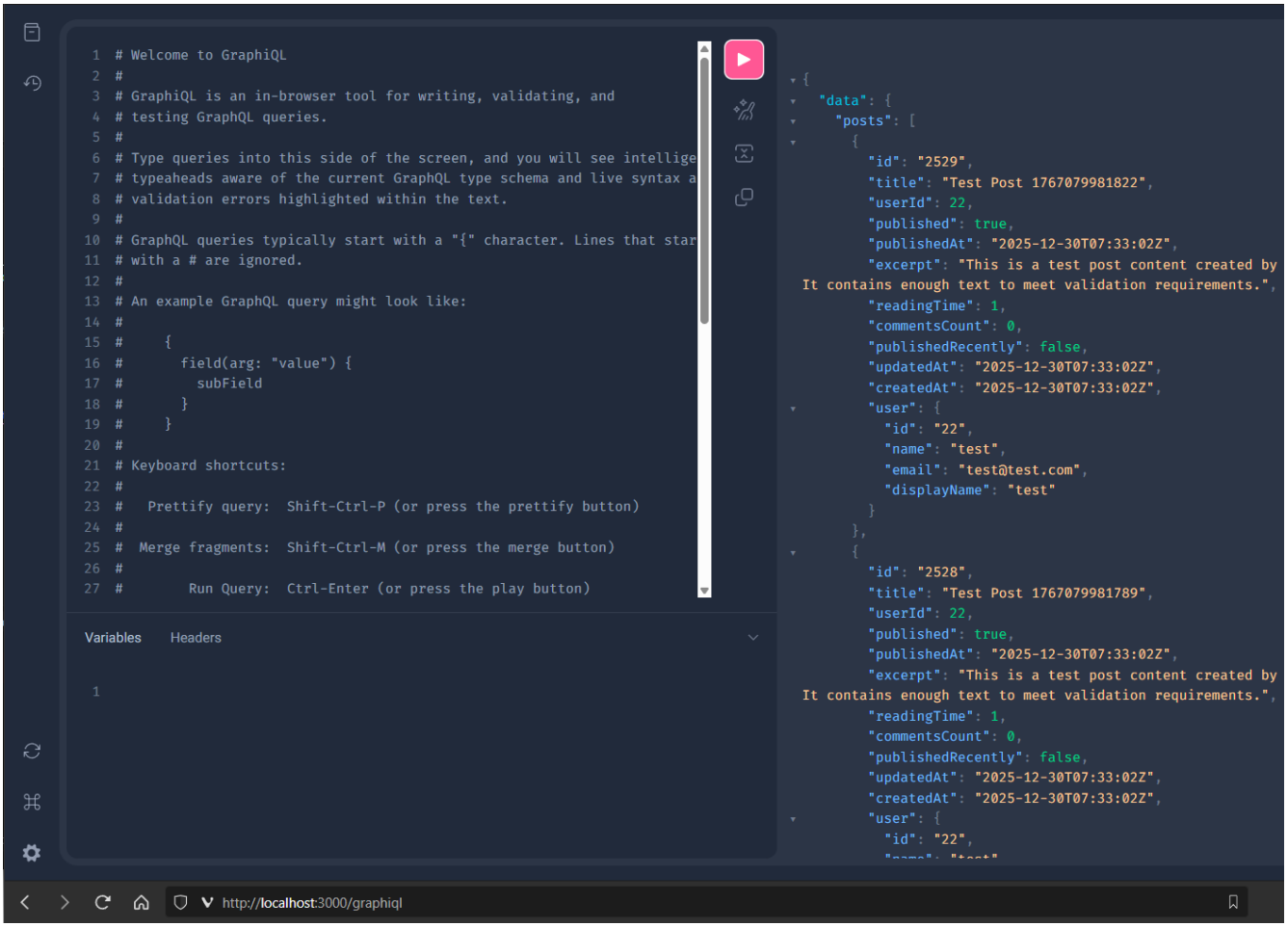

Рисунок 4.2 – інтерфейс graphiql

Основні області екрана:

- редактор запиту (Query editor): поле для введення GraphQL-запиту або мутації;
- панель змінних (Variables): JSON-поле для параметрів запиту/мутації (variables);
- панель відповіді (Response/Result): відображення JSON-результату виконання (data) або помилок (errors);
- документація (Docs/Schema Explorer): перегляд типів, полів, аргументів та прикладів структури запиту.

Операції в GraphiQL:

- виконати запит (Query): ввести запит у редактор - натиснути кнопку виконання - переглянути data у панелі результату;
- виконати мутацію (Mutation): ввести мутацію - за потреби заповнити variables у форматі JSON - виконати - перевірити створений об’єкт і список помилок;
- переглянути схему: відкрити Docs/Schema Explorer - знайти потрібний тип/поле - переглянути аргументи й повернені типи;
- аналіз помилок: при помилці перевірити блок errors (текст помилки, поле/аргумент, що спричинив помилку), після чого скоригувати запит або змінні.

Для запуску локального сервера, навантажувальних сценаріїв та збору метрик використовується екран терміналу. Приклад запуску навантажувального тесту за допомогою терміналу представлено на рис. 4.3.

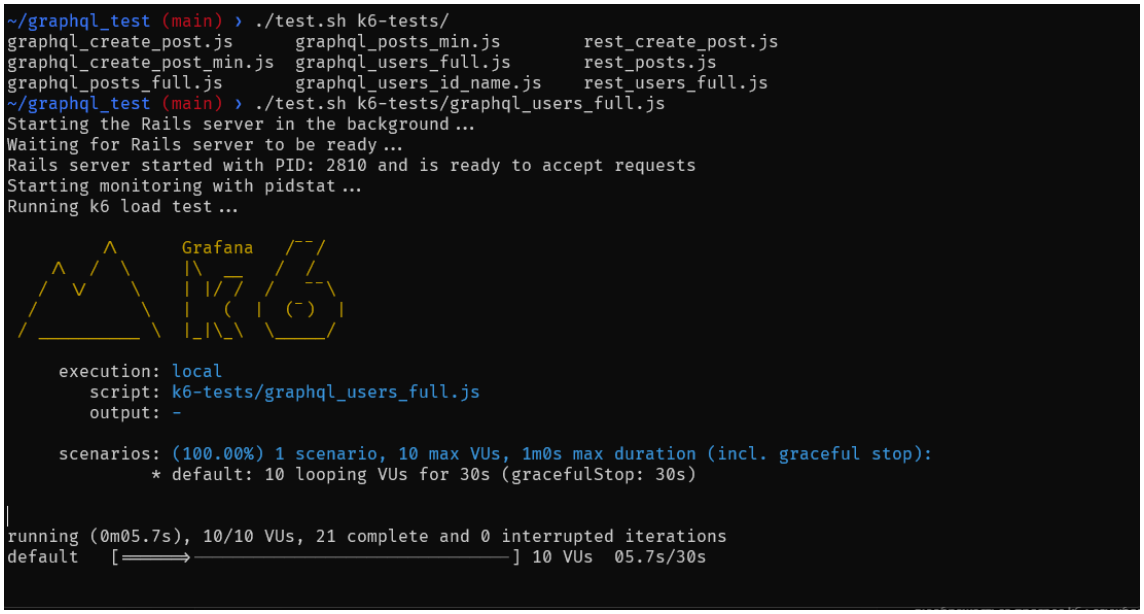

Рисунок 4.3 – запуск навантажувального тесту

Термінал може використовуватись для наступних операцій:

- запуск навантаження: виконати test.sh <шлях_до_k6_сценарію> → дочекатися завершення тесту → скрипт автоматично збере метрики і сформує CSV;
- контроль виконання: під час роботи відображається прогрес k6 і службові повідомлення про запуск/зупинку процесів.

4 АВАРІЙНІ СИТУАЦІЇ

Додаток спроектовано так, щоб максимально уникнути аварійних ситуацій. Безпека даних зумовлюється вбудованими засобами PostgreSQL.

У таблиці 5.1 наведені повідомлення, які може отримати користувач у процесі роботи:

Текст повідомлення	Опис ситуації	Рекомендовані дії
Запис не знайдено (Record not found)	REST API: запит до ресурсу з id, якого немає в БД (HTTP 404).	Перевірити id, отримати актуальний список (GET /api/v1/...) і повторити запит.
Помилка валідації (Validation failed)	REST API: дані не пройшли валідацію або під час створення коментаря не знайдено пов’язані.	Переглянути errors[], виправити вхідні дані та повторити операцію.
Користувача не знайдено (User not found)	REST/GraphQL: створення поста/коментаря з неіснуючим user_id.	Використати коректний user_id або створити користувача й повторити.
Пост не знайдено (Post not found)	REST/GraphQL: створення коментаря з неіснуючим post_id.	Використати коректний post_id або створити пост і повторити.
Відсутній параметр post (param is missing or the value is empty or invalid: post)	REST API: не передано ключ post у JSON для create/update (HTTP 400).	Надіслати JSON з ключем post і дозволеними полями (title, content, user_id, …).

Текст повідомлення	Опис ситуації	Рекомендовані дії
Відсутній параметр comment (param is missing or the value is empty or invalid: comment)	REST API: не передано ключ comment у JSON для create/update (HTTP 400).	Надіслати JSON з ключем comment і полями (content, user_id, post_id).
Неочікуваний параметр (Unexpected parameter: …)	GraphQL: параметр variables передано в неочікуваному форматі/типі (development повертає errors[].message).	Передавати variables як JSON-об’єкт або JSON-рядок; перевірити тіло POST /graphql.
Ім’я не може бути порожнім (Name can't be blank)	Валідація моделі User при створенні/оновленні (REST або GraphQL).	Заповнити name і повторити запит.
Email некоректний (Email is invalid)	Валідація email у User.	Вказати коректний email і повторити.
Email вже використовується (Email has already been taken)	Спроба створити користувача з email, що вже існує.	Вказати інший email або видалити/змінити існуючий запис.

ДОДАТОК Г

[redacted]

ABSTRACTS
OF THE XIX INTERNATIONAL CONFERENCE
«MODERN INFORMATION AND COMMUNICATION TECHNOLOGIES ON A TRANSPORT, IN INDUSTRY AND EDUCATION»
18-19, December, 2025

СУЧАСНІ ІНФОРМАЦІЙНІ ТА КОМУНІКАЦІЙНІ ТЕХНОЛОГІЇ НА ТРАНСПОРТІ, В ПРОМИСЛОВОСТІ І ОСВІТІ

ПРИСВЯЧЕНО ПАМ'ЯТІ ПРОФЕСОРА ІГОРЯ ЖУКОВИЦЬКОГО

ТЕЗИ
XIX МІЖНАРОДНОЇ НАУКОВО-ПРАКТИЧНОЇ КОНФЕРЕНЦІЇ
18-19 ГРУДНЯ 2025

ДНІПРО
2025

Дослідження продуктивності GraphQL при використанні у веб-додатках

Григоренко А. Л., Горячкін В. М., Український державний університет науки та технологій, Україна

У системах реального часу, таких як промисловість та транспорт, неефективний API (традиційний REST) спричиняє неприпустимі затримки через надлишкову вибірку даних (overfetching). Традиційний підхід REST, що спирається на простоту та стабільність, протистоїть гнучкості GraphQL, який пропонує вирішення фундаментальних проблем надлишкової (overfetching) та недостатньої (underfetching) вибірки даних. Це протистояння перетворює архітектурний вибір на інженерно обґрунтоване рішення з прямими наслідками для продуктивності. Метою даної роботи є емпірична оцінка та порівняльний аналіз продуктивності GraphQL і REST за ключовими метриками – часом відгуку, навантаженням на центральний процесор та оперативну пам'ять, а також обсягом мережевого трафіку – у сценаріях, що моделюють реальні умови експлуатації веб-додатків.

Дослідження проводилося у контрольованому програмно-апаратному середовищі на базі веб-застосунку, розробленого на фреймворку Ruby on Rails з використанням системи управління базами даних PostgreSQL. Навантаження генерувалося за єдиним профілем, для забезпечення точності результатів використовувалися дані другого запуску кожного тесту, що дозволило нівелювати ефекти холодного старту. Експеримент включав три сценарії, що моделюють типові операції: отримання списку користувачів, отримання публікацій з даними авторів, та створення нової публікації. Для кожного сценарію порівнювалися три реалізації: класичний REST, GraphQL із запитом повного набору полів та GraphQL із запитом мінімально необхідного набору полів.

Аналіз отриманих даних демонструє, що ефективність кожного підходу суттєво залежить від характеру операції. У сценарії отримання списку користувачів класичний REST зберігає перевагу в простих, "пласких" запитах (сценарій списку користувачів), демонструючи кращий час відгуку. Водночас GraphQL із запитом мінімально необхідного набору полів виявився значно ефективнішим з погляду використання мережі.

Ключовий експеримент – отримання ієрархічних даних (публікацій з авторами) – став вирішальним доказом переваг GraphQL, оскільки він ефективно усуває проблему overfetching. У цьому сценарії GraphQL значно перевершив REST за всіма показниками: медіанна затримка зменшилася на ~38% (0.643с проти 1.03 с), навантаження на ЦП знизилося з 26.58% до 16.23%, мережевий трафік був радикально скорочений – з 452 КБ/с до 15 КБ/с. Це наочно демонструє, як здатність GraphQL запитувати лише необхідні пов'язані дані ефективно вирішує проблему надлишкових запитів.

Проведене дослідження підтверджує, що вибір між REST та GraphQL не є однозначним і повинен ґрунтуватися на специфіці конкретного завдання. REST залишається оптимальним та високопродуктивним рішенням для простих запитів, де надлишковість даних мінімальна. Водночас GraphQL демонструє значні переваги у продуктивності, ефективності використання ресурсів та мережевого трафіку при виконанні складних ієрархічних запиті.

У контексті сучасних промислових і транспортних систем, де API слугують для обміну складними, ієрархічними даними – від телеметрії обладнання до логістичних ланцюгів – проблема overfetching стає не теоретичною, а практичною перешкодою для продуктивності. Проведене дослідження показує, що обґрунтоване застосування GraphQL може стати ключовим фактором оптимізації продуктивності та зниження мережевих витрат, забезпечуючи зниження затримки на ~38% та скорочуючи трафік більш ніж у 30 разів порівняно з REST, що є критичним для систем реального часу.