УДК 004.896 , 044.738.5, 044.75

В.О. Максименко, магістр
О.В. Фролов, к.т.н., доцент
Харківський національний економічний університет ім. С. Кузнеця, м. Харків, Україна
[1]maksymenko.vladyslav.o@gmail.com
[2]frolgx@gmail.com

Порівняльний аналіз продуктивності gRPC та RESTful API мікросервісів з використанням MongoDB та MS SQL Server

У статті досліджується вплив архітектурних особливостей мікросервісної архітектури на вибір між gRPC та RESTful API. Мета дослідження - оцінка продуктивності gRPC та RESTful API у мікросервісній архітектурі залежно від використовуваної бази даних та проведених тестів продуктивності. Окрему увагу приділено ролі реляційних (SQL) та нереляційних (NoSQL) баз даних у комунікації між сервісами, а також методам тестування продуктивності за допомогою Apache JMeter. Виконано експериментальне дослідження власної мікросервісної системи для визначення оптимального підходу залежно від затримок, пропускної здатності та ефективності використання ресурсів.
У роботі розглядаються особливості архітектури gRPC та RESTful API, їхні відмінності у передачі даних, принципи масштабування та оптимізації продуктивності. Здійснено порівняння підходів на основі ключових метрик, включаючи швидкість відповіді сервера, споживання ресурсів та можливості обробки високих навантажень. Дослідження включає аналіз впливу вибору між SQL та NoSQL на ефективність API-запитів, а також оцінку використання JMeter для моделювання реальних сценаріїв навантаження.
Отримані результати можуть бути використані для прийняття обґрунтованих рішень щодо вибору технології комунікації в залежності від архітектурних вимог і характеристик програмних продуктів.

Ключові слова: мікросервіси, RESTful API, gRPC, SQL, HTTP/2, MongoDB, продуктивність, JMeter, Protocol Buffers

DOI: 10.31474/1996-1588-2025-1-40-51-59

Вступ

Мікросервісна архітектура стала домінуючою парадигмою у сучасній розробці програмного забезпечення завдяки її гнучкості, масштабованості та незалежності компонентів. Однією з ключових проблем є ефективна взаємодія між сервісами. Найбільш поширеними підходами до комунікації є gRPC та RESTful API.

Крім того, важливим фактором є підходи для побудови архітектури та вибір бази даних: реляційні (SQL) бази забезпечують транзакційну цілісність, тоді як нереляційні (NoSQL) бази підтримують горизонтальне масштабування. Додатково, тестування продуктивності за допомогою JMeter дозволяє оцінити вплив архітектурних рішень на швидкодію системи.

Метою роботи є оцінка продуктивності gRPC та RESTful API у мікросервісній архітектурі залежно від використовуваної бази даних та проведених тестів продуктивності. Дослідження має на меті виявити ключові фактори, що впливають на ефективність комунікації між мікросервісами, проаналізувати вплив вибору бази даних на продуктивність API та визначити оптимальні умови використання кожного з підходів у розроблюваних системах.

Дослідження також враховує специфіку використання JMeter для тестування продуктивності, що дозволяє змоделювати різні сценарії навантаження та визначити оптимальні параметри для роботи сервісів.

Окрему увагу приділено впливу потокової передачі даних у gRPC та її порівнянню з традиційним RESTful API, включаючи можливості зниження мережевого навантаження, зменшення затримки та підвищення загальної ефективності роботи мікросервісної системи.

Таким чином, це дослідження є важливим кроком у розумінні того, як архітектурні рішення, вибір API та бази даних впливають на продуктивність розроблюваних систем та допомагає сформувати розуміння для розробників щодо оптимізації мікросервісних архітектур.

Огляд об'єкта дослідження

Архітектура програмного забезпечення – це структура програмної системи, що охоплює компоненти, їхні взаємозв'язки, а також принципи взаємодії між ними. Вона є одним з ключових

аспектів розробки ПЗ, оскільки визначає високорівневі рішення, які впливають на всі етапи створення, тестування та підтримки програмного продукту [1] Це забезпечує напрямок для комунікації між різними зацікавленими сторонами, такими як розробники, замовники, тестувальники та користувачі.

Головною метою архітектури є забезпечення стійкого фундаменту, на основі якого будуватимуться всі компоненти програми. До основних характеристик належать:

1) підтримуваність – гарно спроєктована архітектура полегшує внесення змін, адаптацію системи до нових вимог і виправлення помилок;

2) масштабованість – важливий аспект, коли система повинна обробляти більші обсяги даних або користувачів;

3) продуктивність – архітектурні рішення щодо обміну даними між компонентами та використання обчислювальних ресурсів впливають на загальну швидкодію;

4) безпека – визначає, як система захищатиме дані та забезпечуватиме контроль доступу до них.

З розвитком інформаційних технологій з'явилося кілька основних архітектурних підходів, які відрізняються за рівнем взаємодії компонентів і гнучкістю адаптації до змін. Основні архітектурні моделі включають:

1) монолітна архітектура (MA) – традиційний підхід до побудови програмних систем, при якому всі компоненти програми інтегровані в один великий кодовий базис, всі частини системи, включаючи бізнес-логіку, доступ до даних, обробку запитів та інші функціональні одиниці, обробляються в межах одного додатку;

2) сервісно-орієнтована архітектура (SOA) – дозволяє створювати окремі сервіси, які взаємодіють через загальні шини даних;

3) подієво-орієнтована архітектура (EDA) – базується на асинхронному обміні подіями між компонентами системи;

4) мікросервісна архітектура (MSA) – передбачає розбиття програми на незалежні мікросервіси, що взаємодіють через API (рис. 1).

Мікросервісна архітектура дозволяє створювати програми як набір незалежних, самодостатніх сервісів, кожен з яких виконує чітко визначену функцію. Ці сервіси взаємодіють між собою через стандартизовані протоколи обміну даними, такі як HTTP, gRPC, AMQP та інші.

Кожен сервіс розробляється відповідно до конкретних бізнес-потреб і може бути оновлений, масштабований і розгорнутий без впливу на інші сервіси, що забезпечує гнучкість у процесі розробки і підтримки програмного забезпечення [2,3].

Сервіс у мікросервісній архітектурі – це не просто програмний модуль, який взаємодіє з базою даних, а добре продумана модель, орієнтована на реалізацію певних бізнес-цілей.

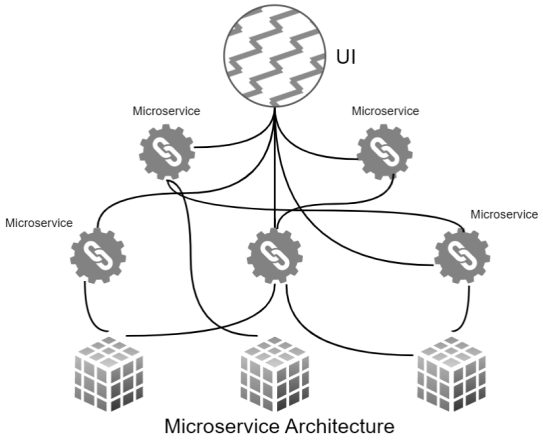

Рисунок 1 – Прототип мікросервісної архітектури
Джерело: Авторський рисунок

Розгортання таких сервісів відбувається незалежно, здебільшого автоматизованим чином. Однією з основних характеристик цієї архітектури є вибір протоколу і способу комунікації між сервісами, що має прямий вплив на технічні характеристики системи, такі як продуктивність, а також на її масштабованість, стабільність і ефективність інтеграції з іншими компонентами системи. Основні переваги даного підходу.

Гнучкість у розробці – кожен сервіс можна розробляти незалежно від інших. Завдяки цьому розробка може бути паралельною, що значно пришвидшує процес створення програмного забезпечення. Крім того, зміни в одному сервісі не впливають на інші, що зменшує ризик помилок при інтеграції та дозволяє швидше вносити нові функціональні можливості або виправлення [4].

Масштабованість – можна масштабувати лише ті частини системи, які цього потребують. Це дає можливість виділяти ресурси для тих частин програми, які найбільше потребують обчислювальних потужностей або обсягу пам'яті. Наприклад, якщо один сервіс обробляє великий потік запитів, його можна масштабувати окремо від інших частин системи, що знижує витрати на ресурси та підвищує ефективність використання інфраструктури [5].

Стійкість до збоїв – відмова одного сервісу не призводить до зупинки всієї системи. Кожен сервіс працює як окрема одиниця, що дозволяє ізолювати проблеми. Інші сервіси продовжують працювати, і система може використовувати механізми відновлення, наприклад, через повторні спроби або резервні копії.

Технологічна незалежність – кожен сервіс може використовувати різні технології в залежності від потреб кожного сервісу. Наприклад, один мікросервіс може бути написаний на Python і

використовувати PostgreSQL для зберігання даних, в той час як інший може бути написаний на Java з базою даних MongoDB [6]. Це дає змогу вибирати найкращі інструменти для конкретних завдань, підвищуючи продуктивність і зручність розробки.

Вибір архітектурного підходу залежить від вимог до продуктивності, масштабованості та простоти підтримки системи. Мікросервісна архітектура особливо добре підходить для великих, динамічних проєктів, які потребують гнучкості та можливості швидкого розширення функціоналу.

API протоколи

Протоколи прикладного програмного інтерфейсу (API) є наборами встановлених правил, угод та стандартів, що сприяють ефективному обміну даними та взаємодії між різними програмними додатками та інформаційними системами. Вони визначають, як саме формуються запити та відповіді між клієнтом і сервером, а також визначають методи взаємодії, що регулюють процеси передачі даних і виконання операцій. API можна розглядати як своєрідний міст, що дозволяє різним програмам та системам взаємодіяти та інтегрувати свої функціональні можливості.

Завдяки використанню API розробники можуть забезпечити доступ до специфічних функцій чи сервісів без необхідності глибокого занурення в деталі реалізації цих компонентів. Протоколи API визначають не лише формат даних, а й правила, що регламентують обробку помилок, авторизацію користувачів та інші аспекти комунікації між програмними продуктами.

Найпоширенішими та найвідомішими протоколами API є RESTful (Representational State Transfer), GraphQL та gRPC. Кожен з них має свої особливості, що дозволяють вибрати найбільш підходящий підхід залежно від вимог конкретної системи. Наприклад, REST є популярним завдяки своїй простоті і використанню стандартних HTTP методів для взаємодії з ресурсами, GraphQL дозволяє більш гнучко визначати запити, забезпечуючи точність у отриманні необхідних даних, а gRPC, використовуючи протокол HTTP/2, орієнтований на високу швидкість і ефективність у випадку складних систем, де важлива низька затримка та висока продуктивність.

Representational State Transfer – архітектурний стиль для розробки API, який забезпечує зв'язок між клієнтами і серверами через протокол HTTP (рис. 2). REST був вперше представлений Роєм Філдінгом у 2000 році в його докторській дисертації в Каліфорнійському університеті [7]. Основна ідея REST полягає в тому, що клієнт і сервер спілкуються через стандартний HTTP/1.1 протокол для передачі даних. У системах, що використовують REST, кожен сервіс зазвичай має певну кінцеву точку (endpoint), яка дозволяє взаємодіяти між сервісами та обмінюватися даними [8].

REST використовує кілька основних методів для роботи з даними, серед яких GET, POST, PUT, DELETE, PATCH, HEAD, TRACE. Найбільш часто використовуються формати представлення даних, такі як JSON та XML, при цьому JSON є переважаючим завдяки своїй простоті, легкості у використанні та ефективності.

Основні особливості:

1) чітка організація ресурсів – ресурси, такі як дані або сервіси, представлені у вигляді URI (унікальних ідентифікаторів), що забезпечує простоту доступу до кожного елемента, ресурси можуть бути отримані або змінені за допомогою стандартних HTTP методів;

2) статусні коди HTTP – RESTful API використовує стандартні HTTP статусні коди для індикації результатів операцій, це значно спрощує обробку помилок і допомагає клієнту та серверу визначити результат запиту;

3) stateless – кожен запит до RESTful API є незалежним від інших і не зберігає жодної інформації про попередні запити, це означає, що сервер не зберігає стан між запитами, що дозволяє простіше масштабувати систему, а також зменшує навантаження на сервер, оскільки кожен запит є самодостатнім.

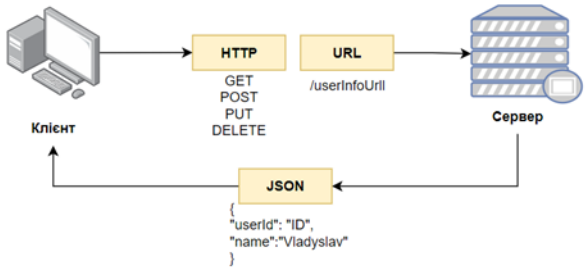

Рисунок 2 – Модель взаємодії між компонентами REST
Джерело: Авторський рисунок

Google Remote Procedure Call (gRPC) – високопродуктивний фреймворк з відкритим кодом, призначений для створення ефективних розподілених систем, мікросервісів і API незалежно від мови програмування та платформи. Розроблений компанією Google, gRPC забезпечує швидку та надійну взаємодію між програмними модулями, використовуючи механізм віддаленого виклику процедур (Remote Procedure Call, RPC) [8-9]. Однією з ключових особливостей gRPC є використання протокольних буферів (Protobufs) як основного формату серіалізації. Протокольні буфери – це компактний, ефективний і незалежний від мови спосіб визначення структур даних і методів обслуговування, що значно пришвидшує

процес комунікації між сервісами (рис. 3). На основі цих визначень gRPC автоматично генерує серверний і клієнтський код для широкого спектра мов програмування.

```
syntax = "proto3";

message User {
      int32 id = 1;          //Поле id має тег 1
      string name = 2;       //Поле name має тег 2
      string email = 3;      //Поле email має тег 3
      repeated string roles = 4; //Поле roles (повторюване) має тег 4
}
```

Рисунок 3 – Приклад файлу визначення схеми protobuf
Джерело: Авторський рисунок

gRPC використовує протокол HTTP/2, що надає ряд переваг у порівнянні з традиційним HTTP/1.1 (рис. 4), зокрема:

1) двонаправлену потокову передачу (Bidirectional Streaming) – клієнт і сервер можуть обмінюватися повідомленнями в режимі реального часу без необхідності створення окремих з'єднань;

2) мультиплексування запитів (Multiplexing) – одночасне надсилання декількох запитів по одному з'єднанню без блокування;

3) ефективну компресію даних – зменшення навантаження на мережу завдяки більш компактному формату передавання;

4) зменшену затримку (Low Latency) – швидкість обробки запитів є вищою порівняно з традиційними RESTful API завдяки використанню бінарного формату даних.

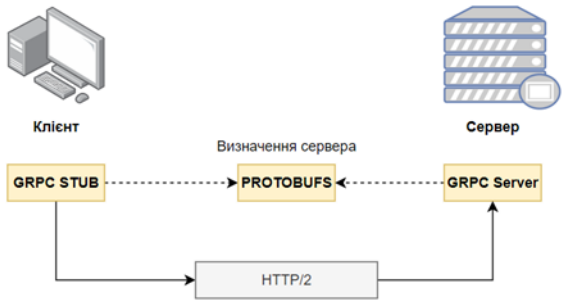

Рисунок 4 – Модель взаємодії між компонентами gRPC
Джерело: Авторський рисунок

Усі ці протоколи полегшують інтеграцію різних систем і сприяють розвитку гнучких і масштабованих архітектур, що дають змогу розробникам будувати ефективні, надійні та зручні у використанні застосунки.

Використання баз даних у мікросервісній архітектурі

Одним із ключових аспектів побудови ефективної мікросервісної архітектури є вибір бази даних. Реляційні (SQL) та нереляційні (NoSQL) бази даних мають свої переваги та обмеження, які суттєво впливають на продуктивність і масштабованість системи.

Реляційні бази даних базуються на структурованих таблицях, що мають чіткі зв'язки між даними та підтримують транзакції ACID (Atomicity, Consistency, Isolation, Durability). Вони широко використовуються в системах, де важлива цілісність і узгодженість даних.

Microsoft SQL Server є потужною реляційною системою керування базами даних (RDBMS), яка використовує мову структурованих запитів (SQL) для роботи з даними. Вона забезпечує чітко визначену схему, де всі дані організовані в таблиці з фіксованою структурою, що унеможливлює довільні зміни моделі без попереднього планування. SQL Server підтримує ACID-транзакції, що гарантує узгодженість даних навіть у випадку збоїв або одночасного виконання операцій. Крім того, він має вбудовані засоби аналітики та звітності, що робить його ідеальним для використання у фінансових системах, ERP-рішеннях та інших сферах, де важливі складні взаємозв'язки між даними. Щодо масштабованості, SQL Server здебільшого використовує вертикальне масштабування (збільшення ресурсів сервера), хоча також підтримує реплікацію та горизонтальне розподілення даних.

MongoDB, на відміну від SQL Server, належить до класу NoSQL-баз даних і використовує документно-орієнтовану модель. Дані в MongoDB зберігаються у вигляді JSON-подібних документів, що забезпечує гнучку схему та дозволяє змінювати структуру без необхідності модифікації всієї бази. Це особливо корисно для сервісів, які працюють із напівструктурованими або неструктурованими даними, наприклад, у сфері веб-аналітики, обробки подій або зберігання інформації про користувачів. Важливою перевагою MongoDB є можливість горизонтального масштабування завдяки шардингу, що дозволяє обробляти великі обсяги даних із мінімальним навантаженням на окремі сервери. Водночас, на відміну від MS SQL Server, MongoDB не гарантує суворої узгодженості та натомість використовує більш гнучку модель BASE, що дає змогу досягти високої продуктивності в розподілених системах.

У мікросервісній архітектурі часто використовується підхід "Polyglot Persistence", який передбачає застосування різних баз даних залежно від вимог конкретного сервісу. Для критично важливих сервісів, що потребують високої узгодженості та складних бізнес-правил, доцільно використовувати MS SQL Server. Водночас для сервісів, які працюють із динамічними та масштабованими даними, більш

ефективним рішенням є MongoDB. Правильний вибір бази даних відіграє ключову роль побудові надійної та продуктивної мікросервісної архітектури. Порівняння характеристик розглядуваних СУБД зведено до табл.1.

Таблиця 1 – Порівняння MS SQL Server і MongoDB у мікросервісній архітектурі

Характеристика	SQL Server (RDBMS)	MongoDB (NoSQL)
Тип зберігання	Таблиці (рядки та стовпці)	Документи JSON/BSON
Схема	Строга, фіксована	Гнучка, динамічна
Узгодженість	ACID-транзакції	Гнучка узгодженість (BASE)
Масштабування	Вертикальне, можлива реплікація	Горизонтальне (шардинг)
Продуктивність	Висока для складних запитів	Висока для масових операцій
Сценарії використання	Фінанси, ERP, транзакції	Великі дані, IoT, логування

Проєктування архітектури експериментальної системи

Для проведення аналізу впливу архітектурних особливостей на вибір між gRPC та RESTful API розроблено тестове середовище, яке включає декілька шарів, що взаємодіють між собою через різні протоколи та використовують дві бази даних – SQL та NoSQL (MongoDB).

Проєктна архітектура складається з трьох основних компонентів (рис. 5):

1) User Layer (Користувацький рівень): забезпечує взаємодію користувача із системою за допомогою Postman або Swagger. Swagger використовується для автоматичної генерації документації API, що полегшує тестування та інтеграцію, Postman дозволяє моделювати запити до сервісів у різних форматах, таких як JSON для RESTful API та Protocol Buffers для gRPC;

2) Back-End Structure (Серверний рівень): містить основну бізнес-логіку системи та складається з двох підрівнів:

– Experience Microservices Layer – включає WebApiProject, що реалізує RESTful API, та GrpcServiceProject, який використовує gRPC для обміну даними через HTTP/2 та бінарний формат,

– Domain Microservices Layer – представлений DataConnectorLibraryProject, який обробляє запити мікросервісів і взаємодіє з базами даних;

3) Data Stores Layer (Рівень зберігання даних): містить дві бази даних:

– SQL DB – використовується для реляційного зберігання структурованих даних,

– NoSQL DB (MongoDB) – зберігає гнучкі та динамічні дані, що не мають чіткої схеми.

На рис. 6 відображено дані у JSON форматі, що складається з одного об'єкту з ключ-значення пар.


```
{
    "customer": {
        "customerName": "customerNameTest",
        "edpouCode": "edpouCodeTest",
        "firstName": "firstNameTest",
        "lastName": "lastNameTest",
        "patronymic": "patronymicTest",
        "positionId": "bea41aa9-8a38-480a-853d-c557c7d5688e",
        "id": "8271a2a3-fde4-4887-aba4-795ca9c438c1"
    },
    "metrics": {
        "sqlQueryTime": "00:00.362",
        "mongoQueryTime": "",
        "totalExecutionTime": "00:06.754"
    }
}
```


Рисунок 5 – Приклад відповіді з не ієрархічними даними (flat data)
Джерело: Авторський рисунок

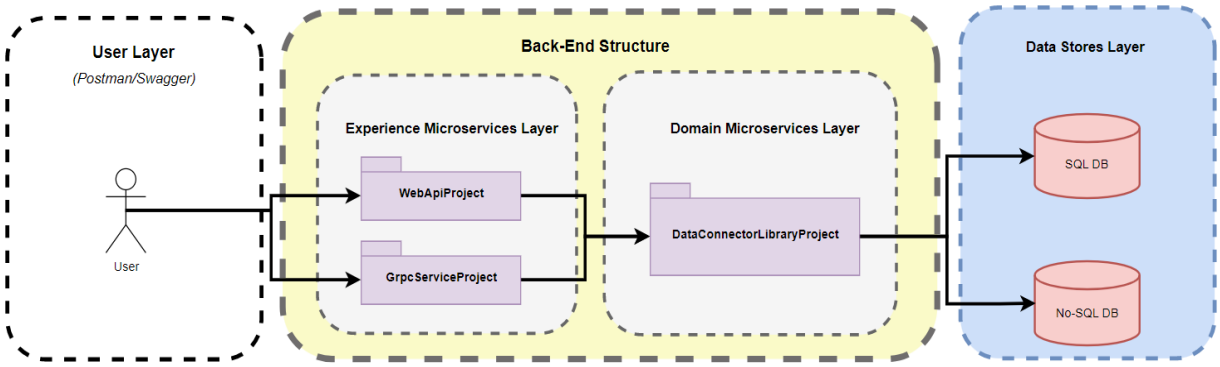

Рисунок 6 – Проєктна логічна архітектурна структура
Джерело: Авторський рисунок

Для об'єктивного порівняння продуктивності RESTful API та gRPC було реалізовано набір мікросервісів, що виконують стандартні CRUD-операції над базою даних. Процес виконання запитів від користувачів зображено на рисунку 7, який демонструє типовий сценарій обробки запитів у розробленій системі.

Важливим аспектом дослідження є оцінка швидкодії обох підходів при роботі з реляційними (SQL) та нереляційними (NoSQL) базами даних. Для забезпечення чистоти експерименту структура баз даних уніфікована незалежно від типу сховища. Вона моделює предметну область бухгалтерії, що включає основні таблиці, наповнені 10 000 записами, та допоміжні таблиці з 1000 записами. Цільова технологія, яка була використана .NET, її потужностей цілком достатньо, щоб задовільнити потреби дослідження. Кожне тестування виконувалося в десяти ітераціях, щоб отримати точні та стабільні результати.

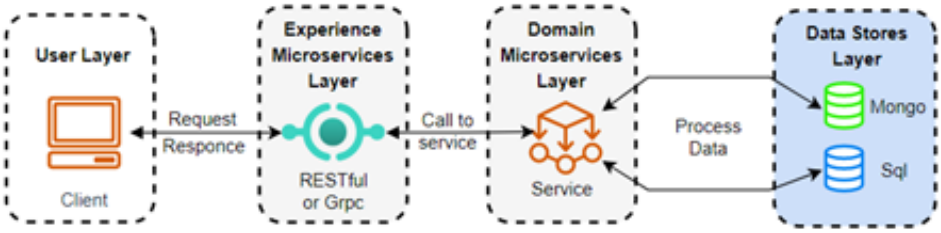

Рисунок 7 – Процес роботи з даними
Джерело: Авторський рисунок

Результати експериментів та їх обговорення

Оцінка продуктивності системи проводилася для аналізу впливу навантаження при отриманні даних на час відгуку та завантаженість процесора. Метою тестування було визначення оптимального способу обміну даними між REST та gRPC для простих структурованих даних.

Для навантажувального тестування API використовувався Apache JMeter – інструмент з відкритим кодом, призначений для перевірки продуктивності та функціональних можливостей систем [10, 11].

Під час дослідження одночасних запитів моделювалися сценарії з різною інтенсивністю навантаження: від 100 до 500 клієнтських запитів, що виконувалися одночасно. Це дозволило оцінити, як система реагує на підвищену активність користувачів, аналізуючи зміну часу відгуку та використання ресурсів процесора. Середній час відповіді розраховується за формулою [6]:

$$aveRT = \frac{1}{n}\sum_{i=1}^{n}(t_{resp_i} - t_{req_i}), \qquad (1)$$

де $aveRT$ – середній час відгуку;
n – кількість запитів;
i – номер запиту;
t_{resp_i} – мітка часу надсилання запиту;
t_{req_i} – мітка часу отримання відповіді.

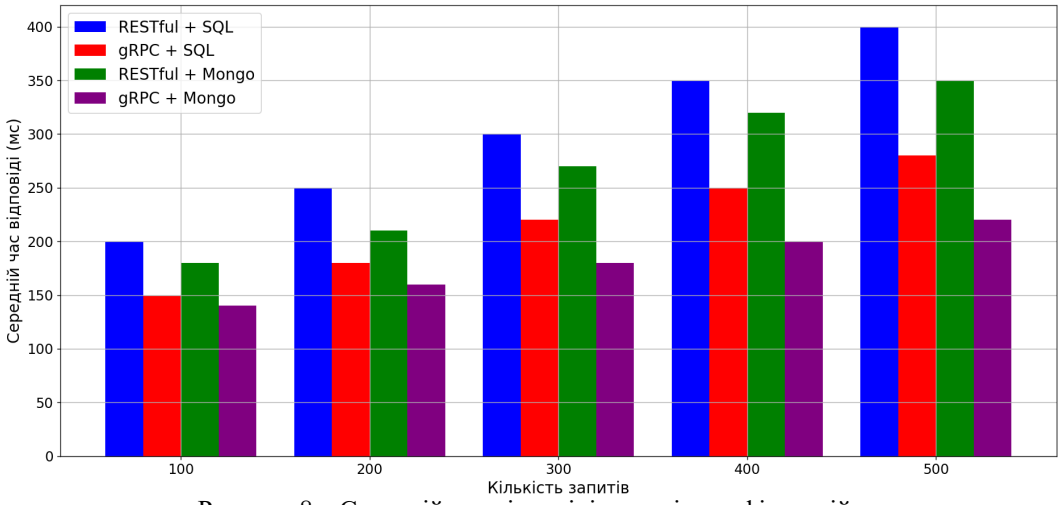

Рисунок 8 – Середній час відповіді для всіх конфігурацій
Джерело: Авторський рисунок

На рис. 8 представлено діаграму, яка показує середній час відповіді для різних конфігурацій API та баз даних при різних обсягах запитів. RESTful API та SQL – ця конфігурація демонструє середній час відповіді для традиційного RESTful API з MS SQL Server базою даних. Як видно на графіку, час відповіді збільшується з ростом кількості запитів, що є типовим для SQL баз даних, де масштабування може впливати на продуктивність.

gRPC та SQL показує середній час відповіді для gRPC, який є більш ефективним для високопродуктивних операцій в порівнянні з вище описаним підходом, gRPC демонструє менший час відповіді для кожної кількості запитів, що вказує на вищу ефективність при обробці запитів.

MongoDB має більш високу продуктивність при зростанні навантаження, ніж традиційні SQL бази. Саме тому RESTful API в такій версії є більш продуктивною. Найшвидша конфігурація на графіку, яка поєднує gRPC з MongoDB. Час відповіді для цієї конфігурації значно менший у порівнянні з усіма іншими. Це підтверджує перевагу використання gRPC разом з NoSQL базою даних, оскільки MongoDB добре масштабується при високому навантаженні, а gRPC забезпечує швидше оброблення запитів.

RESTful та SQL демонструє лінійне зростання використання CPU з кількістю запитів (рис. 9). При цьому зростання плавне, але часом спостерігаються незначні аномалії через вплив бази даних SQL та структури RESTful API. Оскільки RESTful API використовує більше ресурсів на обробку кожного запиту, зростання CPU відбувається з більшою швидкістю.

gRPC та SQL показує дещо менший рівень використання CPU (рис. 9) порівняно з RESTful API, навіть при збільшенні кількості запитів. Це можна пояснити ефективнішою архітектурою gRPC, яка працює швидше в порівнянні з традиційним RESTful API.

В результаті gRPC обробляє запити з меншими витратами ресурсів, що підтверджує його високу продуктивність.

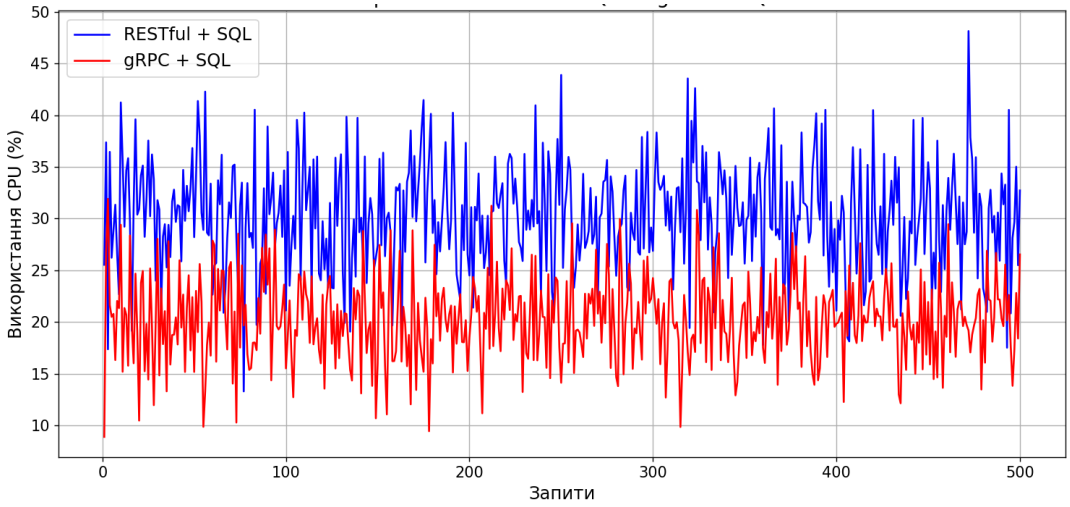

Рисунок 9 – Завантаження CPU протягом 500 запитів для отримання однорідних даних з використанням реляційної бази даних
Джерело: Авторський рисунок

Рис. 10 демонструє використання тих самих технології з нереляційною базою даних – RESTful показує дещо вищі показники використання CPU, порівняно з gRPC, оскільки MongoDB може працювати швидше за SQL при високому навантаженні, але RESTful API потребує більше часу та ресурсів на обробку кожного запиту. З огляду на всі отримані результати gRPC та MongoDB виявляється найбільш ефективним варіантом, оскільки поєднання гнучкості NoSQL бази даних і високої ефективності gRPC дозволяє обробляти запити з найменшим навантаженням на CPU. Це відображається в значно нижчому використанні CPU, навіть при збільшенні кількості запитів до 500.

Висновки

gRPC є оптимальним вибором для розподілених систем з високими вимогами до продуктивності та ефективної обробки великих обсягів даних. Його використання забезпечує мінімальні накладні витрати завдяки компактному формату Protocol Buffers, що зменшує час передачі та покращує масштабованість додатків. Це робить gRPC ідеальним рішенням для систем з численними мікросервісами, де важлива висока швидкість комунікації та асинхронна обробка запитів.

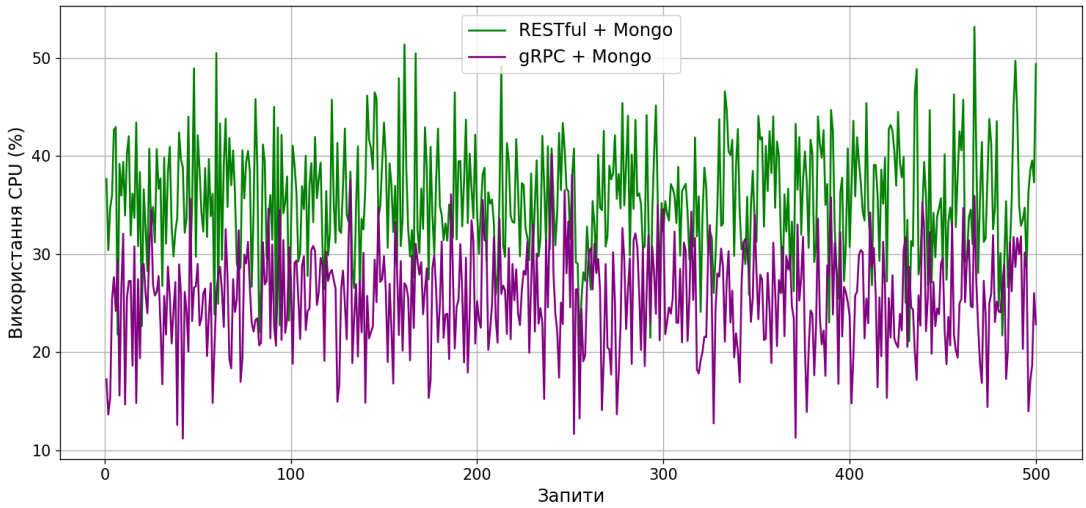

Рисунок 10 – Завантаження CPU протягом 500 запитів для отримання однорідних даних з використанням не реляційної бази даних

Джерело: Авторський рисунок

Однак, для систем, які повинні працювати з широким спектром клієнтів, таких як веб-додатки або мобільні пристрої, RESTful API є більш підходящим. Він є більш універсальним, сумісним з різними платформами і дозволяє розробникам зосередитися на простоті інтеграції за допомогою стандартного HTTP та JSON.

Основні рекомендації для розробників.

Якщо ви працюєте над проектом, де важлива висока продуктивність і ефективність передачі даних (наприклад, при обміні великими обсягами даних або з високою частотою запитів), обирайте gRPC. Для менш вимогливих до швидкості та простих систем, де важлива сумісність з численними клієнтами, RESTful API буде кращим вибором.

У великих проєктах з кількома сервісами рекомендується комбінувати обидва підходи. Наприклад, можна використовувати gRPC для внутрішніх запитів між мікросервісами і REST для взаємодії з зовнішніми клієнтами.

RESTful API, завдяки своїй простоті, має широку документацію і підтримку, тому для команд з меншим досвідом в розподілених системах це може бути перевагою. gRPC вимагає більш детальної налаштування та кращого розуміння технології, але при правильному впровадженні може значно покращити продуктивність.

Для gRPC важливо налаштувати відповідні інструменти тестування, оскільки для нього необхідно писати специфічні тести на основі схем і сервісів, що використовуються в Protocol Buffers. Для RESTful API існує багато інструментів тестування, що полегшує процес розробки і підтримки.

Для gRPC важливо продумати стратегію управління версіями через зміни у схемах, що можуть порушити сумісність з попередніми версіями. В RESTful API зазвичай використовують версіонування через URL, що дає більше гнучкості для оновлень без порушення роботи старих клієнтів. Для зменшення навантаження на систему в умовах великих обсягів запитів можна застосовувати оптимізації в gRPC, такі як стиснення даних і використання асинхронних запитів, щоб знизити час очікування і покращити масштабованість сервісів.

Література

1. Jain R. Art of computer systems performance analysis. Wiley, 1991. 685 p.
2. Söylemez M., Tekinerdogan B., Tarhan A. K. Challenges and solution directions of microservice architectures: A systematic literature review applied sciences. Appl. Sci. 2022. Vol.12 (11), 5507, 40p.
3. Wyciślik Ł., Latusik Ł., Kamińska A. M. A Comparative assessment of JVM frameworks to develop microservices. Appl. Sci., 2023. Vol.13 (3), 5793, 22 p. DOI: https:// doi.org/10.3390/app13031343.
4. Containerized microservices orchestration and provisioning in cloud computing: A conceptual framework and future perspectives / A. Saboor, M.F. Hassan, R. Akbar, S.N.M. Shah, F. Hassan, S.A. Magsi, M.A. Siddiqui, Appl. Sci., 2022. Vol.12, 5793, 21 p. DOI: https:// doi.org/10.3390/app12125793.
5. Архітектура мікросервісів: Особливості, переваги, реальні приклади. URL: https://www.hostzealot.com.ua/blog/about-solutions/arxitektura-mikroservisiv-osoblivosti-perevagi-realni-prikladi (Дата звернення 02.04.2025).

6. Performance evaluation of microservices communication with REST, GraphQL, and gRPC / M. Niswar, R. A. Safruddin, A. Bustamin, I. Aswad, International Journal of Electronics and Telecommunications, 2024. Vol.70 (2), P. 429-436. DOI: https://doi.org/10.24425/ijet.2024.149562.
7. Fielding R. T. Architectural styles and the design of network-based software architectures. University of California, Irvine, 2000. 180 p.
8. Jeremy H. gRPC vs. REST: Key similarities and differences. URL: https://blog.dreamfactory.com/grpc-vs-rest-how-does-grpc-compare-with-traditional-rest-apis/ (Дата звернення 02.04.2025).
9. Ryan L. gRPC motivation and design principles. 2015. URL: https://grpc.io/blog/principles/ (Дата звернення 02.04.2025).
10. Winteringham M. Testing Web APIs. Manning, 2022. 264 p.
11. Apache JMeter™. URL: https://jmeter.apache.org/ (Дата звернення 02.04.2025).

References

1. Jain, R. (1991), *Art of computer systems performance analysis*, Wiley, 685 p.
2. Söylemez, M., Tekinerdogan, B. and Tarhan, A. K. (2022), "Challenges and solution directions of microservice architectures: A systematic literature review applied sciences", *Appl. Sci.*, Vol.12 (11), 5507, 40 p.
3. Wyciślik, Ł., Latusik, Ł. and Kamińska, A. M. (2023), "A Comparative assessment of JVM frameworks to develop microservices", Appl. Sci., Vol.13 (3), 5793, 22 p.
4. Saboor, A., Hassan, M.F., Akbar, R., Shah, S.N.M., Hassan, F., Magsi, S.A. and Siddiqui, M.A. (2022), "Containerized microservices orchestration and provisioning in cloud computing: A conceptual framework and future perspectives", Appl. Sci., Vol.12, 5793, 21 p. DOI: https:// doi.org/10.3390/app12125793.
5. "Microservices architecture: features, benefits, real-world examples" ["Arkhitektura mikroservisiv: Osoblyvosti, perevahy, realni pryklady"], available at: https://www.hostzealot.com.ua/blog/about-solutions/arxitektura-mikroservisiv-osoblivosti-perevagi-realni-prikladi .
6. Niswar, M., Safruddin, R. A., Bustamin, A. and Aswad, I. (2024), "Performance evaluation of microservices communication with REST, GraphQL, and gRPC", *International Journal of Electronics and Telecommunications*, Vol.70 (2), pp. 429–436. DOI: https://doi.org/10.24425/ijet.2024.149562.
7. Fielding, R. T. (2000), *Architectural styles and the design of network-based software architectures*, University of California, Irvine, 180 p.
8. Jeremy, H. "gRPC vs. REST: Key similarities and differences", available at: https://blog.dreamfactory.com/grpc-vs-rest-how-does-grpc-compare-with-traditional-rest-apis/ .
9. Ryan, L. (2015), "gRPC motivation and design principles", available at: https://grpc.io/blog/principles/.
10. Winteringham, M. (2022), *Testing Web APIs*, Manning, 264 p.
11. "Apache JMeter™", available at: https://jmeter.apache.org/ .

Надійшла до редакції: 31.03.2025

V. MAKSYMENKO, O. FROLOV
Simon Kuznets Kharkiv National University of Economics, Kharkiv, Ukraine
maksymenko.vladyslav.o@gmail.com, frolgx@gmail.com

COMPARATIVE PERFORMANCE ANALYSIS OF GRPC AND RESTFUL API MICROSERVICES USING MONGODB AND MS SQL SERVER

This article examines the impact of architectural features of microservice architecture on the selection of gRPC or RESTful API. Particular attention is given to the role of relational (SQL) and non-relational (NoSQL) databases in inter-service communication, as well as to performance evaluation methodologies using JMeter. An experimental study of a custom microservice system was conducted to determine the optimal approach based on latency, throughput, and resource efficiency. The research explores the architectural principles of gRPC and RESTful API, their differences in data exchange mechanisms, scalability strategies, and performance optimization techniques. A comparative analysis is carried out based on key metrics, including server response time, resource consumption, and the ability to handle high loads. Additionally, the study investigates the impact of SQL and NoSQL database choices on API request efficiency and assesses JMeter's effectiveness in simulating real-world load scenarios. The findings of this study can serve as a basis for informed decision-making regarding the selection of communication technologies, taking into account architectural requirements and the specific characteristics of software systems.

Key words: microservices, RESTful API, gRPC, SQL, HTTP/2, Mongo, performance, Protocol Buffers