МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
Харківський національний університет радіоелектроніки
Факультет Комп'ютерних наук
Кафедра Програмної інженерії

КВАЛІФІКАЦІЙНА РОБОТА
Пояснювальна записка

_____другий (магістерський)_____
(рівень вищої освіти)

Дослідження методів та архітектурних рішень міжсерверної взаємодії.

Виконав:
студент __2__ курсу групи ____ІПЗм-21-3__
_____Колесніков С.А._____
(прізвище, ініціали)

Спеціальність 121 – Інженерія програмного

_____забезпечення_____

Тип програми _____освітньо-наукова_____

Керівник _____доц. Чуприна А.С._____
(посада, прізвище, ініціали)

Допускається до захисту

Зав. кафедри _____ З.В. Дудар

2023 р.

Харківський національний університет радіоелектроніки

Факультет _____Комп'ютерних наук_____
(повна назва)

Кафедра _____Програмної інженерії_____
(повна назва)

Рівень вищої освіти_____другий (магістерський)_____

Спеціальність _____121 – Інженерія програмного забезпечення_____
(код і повна назва спеціальності)

Тип програми _____освітньо-наукова програма _____
(освітньо-професійна або освітньо-наукова)

Освітня програма _____Інженерія програмного забезпечення _____
(повна назва)

ЗАТВЕРДЖУЮ:
Зав. кафедри _____
(підпис)
«_____» _____20 ____ р.

ЗАВДАННЯ

НА КВАЛІФІКАЦІЙНУ РОБОТУ

студента _____*Колеснікова Савелія Андрійовича*_____
(прізвище, ім'я, по батькові)

1. Тема роботи «Дослідження методів та архітектурних рішень міжсерверної взаємодії»

затверджена наказом університету від «03» квітня 2023 р. № 83Стз

2. Термін подання роботи до екзаменаційної комісії «__» _____ 2022 р.

3. Вихідні дані до роботи *методи та архітектурні рішення для міжсерверної взаємодії, критерії їх оцінки, серверні застосунки, API, BFF, Visual Studio 2022.*

4. Перелік питань, що потрібно опрацювати в роботі *вступ, аналіз предметної області, аналіз методів міжсерверного зв'язку, критерії їх оцінки, постановка задачі, проведення експериментів та аналіз їх результатів, формування подальших рекомендацій.*_____

КАЛЕНДАРНИЙ ПЛАН

№	Назва етапів роботи	Терміни виконання етапів роботи	Примітка
1	Аналіз предметної області	25.02.2023	виконано
2	Постановка задачі	10.03.2023	виконано
3	Проведення дослідження	01.04.2023	виконано
4	Підготовка пояснювальної записки	28.04.2023	виконано
5	Підготовка презентації та доповіді	29.04.2023	виконано
6	Попередній захист	01.05.2023	виконано
7	Перевірка на академічний плагіат	02.05.2023	виконано
8	Нормоконтроль	04.05.2023	виконано
9	Рецензування	05.05.2023	виконано
10	Знесення диплома в електронний архів	08.05.2023	виконано
11	Допуск до захисту у зав. кафедри	09.05.2023	виконано

Дата видачі завдання 17.01.2023 р.

Студент _____

(підпис)

Керівник роботи_____ доцент Чуприна А.С.

(підпис)

РЕФЕРАТ / ABSTRACT

Кваліфікаційна робота магістра містить: 66 с., 18 рис., 1 табл., 24 джерела, 4 додатки.

СЕРВЕРНІ ЗАСТОСУНКИ, МІКРОСЕРВІСИ, API, BFF, REST, GRPC, GRAPHQL, .NET, АРХІТЕКТУРА ПРОГРАМНИХ СИСТЕМ.

Об’єктом дослідження є методи та архітектурні підходи міжсерверної взаємодії.

Метою дослідження є аналіз різних за складністю та структурою підходів до створення взаємодії між серверними застосунками для виявлення найбільш доцільного для побудування серверного застосунку.

Методи розробки базуються на основі архітектурних підходів до написання серверних застосунків, мові програмування C#, платформи .NET.

У результаті роботи було проведено ретельний аналіз існуючих методів та архітектурних підходів налагодження міжсерверного зв’язку та виділено параметри та характеристики для вибору найбільш влучного для задачі створення мікросервісних застосунків.

WEB APPLICATIONS, MICROSERVICES, API, BFF, REST, GRPC, GRAPHQL, .NET, SOFTWARE ARCHITECTURE.

The object of research is methods and architectural solutions of interserver interaction.

The aim of the study is to analyze variant in complexity and structure approaches for providing interserver interaction in order to find the most applicable for API development.

As a result, an analysis of existing studies about methods and architectural approaches for building interserver communication was conducted, and defined parameters and characteristics for choosing the best one during API design.

Я, Колесніков Савелій Андрійович, студент групи ІПЗм-21-3, здобувач вищої освіти на другому (магістерському) рівні, кафедра Програмної інженерії, заявляю: моя кваліфікаційна робота на тему «Дослідження методів та архітектурних рішень міжсерверної взаємодії», що буде представлена до ЕК для публічного захисту, виконана самостійно, в ній не містяться елементи плагіату і вона може бути опублікована в електронному архіві відкритого доступу ElArKhNURE. Всі запозичення з друкованих та електронних джерел мають відповідні посилання.

Я ознайомлений з діючим положенням «Про протидію академічному плагіату в ХНУРЕ», згідно з яким виявлення плагіату є підставою для відмови в допуску кваліфікаційної роботи до захисту та застосування дисциплінарних заходів.

ЗМІСТ

ПЕРЕЛІК УМОВНИХ СКОРОЧЕНЬ

API Application Programming Interface, Прикладний програмний інтерфейс

BFF Backend for Frontend, серверна частина для клієнтської

gRPC Google Remote Procedure Calls

REST Representational State Transfer

ART Average response time, Середній час відповіді

TP Throughput, Пропускна здатність

DTS Data Transfer Size, Розмір переданих даних

ВСТУП

Стрімкий розвиток IT-індустрії спричинив великий попит на різноманітні інтернет сервіси. Багато привичних сфер життя перейшло у цифровий вимір. Сучасний світ вже неможливо уявити без таких великих торгівельних площадок, на яких ми робимо покупки кожен день, як Amazon, AliExpress, Rozetka, або сервісів, на яких ми навчаємося або відпочиваємо, проводячи вільний час, таких як YouTube та Netflix. Спеціаліст відомої німецької компанії зі збору статистичних даних Стейсі Дайксон у своїй роботі 2022 року зазначає, що щоденно YouTube витримує навантаження у 2.1 мільярдів користувачів щомісячно [1].

Таку кількість користувачів неможливо б було витримати без особливих підходів до архітектури системи. Є безліч прикладів, коли великі компанії, обирали інвестиції саме у такий вид побудови програмного забезпечення для своїх серверів. Наприклад, веб сайти, наприклад Amazon та Netflix перейшли від монолітної архітектури до мікросервісної. Користувачі Netflix займають 30% усього інтернет трафіку, його сервіси витримують мільярди запитів у день, та кожен запит до API веде у середньому до 6 внутрішніх запитів усередині системи [2]. Amazon, у свою чергу, перейшов до мікросервісної архітектури через брак масштабованості, тобто в один момент навантаження стали такими великими, що сайт не витримував та не відповідав на запити користувачів, тобто бізнес втрачав покупців, а, отже і прибуток. Зараз Amazon складається з сотень сервісів. Для одного тільки завантаження сайту відбувається понад 100-150 викликів різноманітних сервісів, які використовуються для отримання даних для побудування сторінки, яка відображається користувачу [2].

Усі перелічені вище об’єдную те, що вони були перебудовані на основі мікросервісної архітектури, для підвищення швидкості роботи і можливості масштабування. Такий архітектурний стиль став популяризуватися у 2010х роках. Його основна ідея – це представлення застосунку у виді невеличких сервісів, які мають свої окремі бази даних та логіку, розбиту за доменною областю. Сервіси

мають API та спілкуються один з одним через нього запитами за допомогою протоколів передачі даних, переважно HTTP [3].

Такий підхід дозволяє збільшити кількість користувачів, які можуть одночасно користуватися сервісами у рази, за допомогою незалежного масштабування окремих частин системи, на які йде найбільше навантаження. Також, завдяки невеликій кодовій базі сервісів, їх розробка йде легше, бо команди не заважають одна одній і мають меншу кількість конфліктів, тому це прискорює випуск оновлень та нового функціоналу сервісів написаних за допомогою цього архітектурного підходу.

Треба зауважити, що мікросервісна архітектура перш за все, це взаємодія сервісів між собою, як у прикладі Amazon де сотні сервісів взаємодіють між собою. Тому вибір стилю і способу взаємодії може сильно вплинути на швидкість роботи та розробки усієї системи.

Тому для кваліфікаційної роботи було обрано тему, що дозволяє розглянути та дослідити підходи та архітектурні рішення до побудови взаємодії серверних застосунків один з одним.

Об’єктом дослідження є швидкодія мікросервісних застосунків з різними підходами до реалізації міжсерверного зв’язку.

Предметом дослідження є методи та архітектурні підходи для побудови міжсерверного зв’язку.

Методами дослідження є проведення вимірів швидкодії серверних застосунків з різними типами міжсерверного зв’язку та подальші обчислення метрик, основаних на цих даних.

Отримані результати дослідження можуть бути використані в процесі прийняття рішення щодо методу зв’язку при проектуванні веб-систем. Також можливе продовження дослідження в рамках більш складних веб-застосунків.

1 АНАЛІЗ ПРЕДМЕТНОЇ ГАЛУЗІ

1.1 Аналіз предметної галузі

За останні десятиріччя серверні застосунки радикально змінилися. Все починалося з монолітних веб-сервісів. Такий підхід є традиційним у написанні програмного забезпечення. У ньому програма складається з багатьох сервісів, які взаємодіють у рамках одного процесу. Це інтуїтивний шлях до написання програм, тому саме з нього почався розвиток серверних застосунків [4]. Переглянемо, як виглядає така архітектура, на прикладі первинної реалізації серверної частини сервісу таксі Uber (див. рис. 1.1).

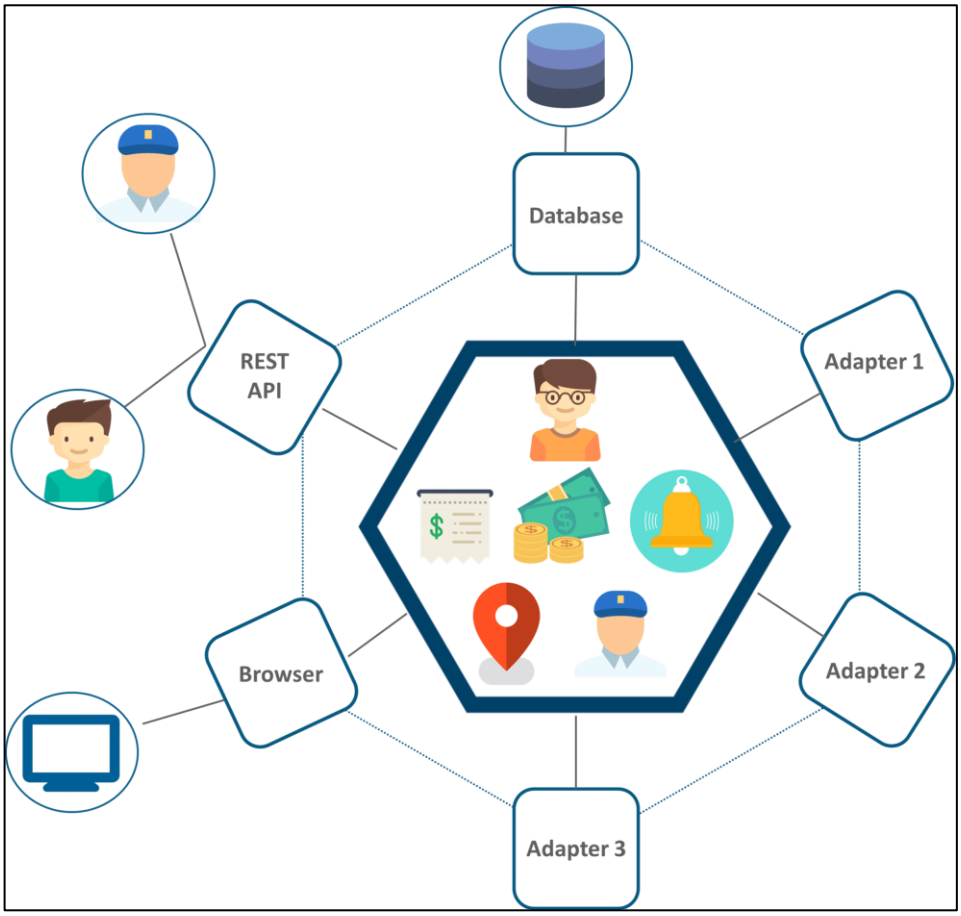

Рисунок 1.1 – Монолітна архітектура сервісу Uber.

Монолітний підхід не сильно розповсюджений зараз, але все ж має свою нішу у світі створення програмного забезпечення, бо має не тільки недоліки, але і

переваги перед мікросервісним, наприклад, під час ранніх етапів розробки серверної частини застосунку, він навіть виграє у свого конкурента мікросервісної архітектури, через легкість розгортання на сервері, легкість розробки, швидкості та процесу відлагоджування. Тобто, така архітектура найчастіше використовується для ранніх етапів розвитку програмного продукту, щоб перевірити гіпотезу. У такої архітектури немає великої кількості міжсерверних взаємозв'язків, усе відбувається у рамках одного сервера.

Однак, через свої недоліки, монолітна архітектура поступилася іншим. Так, наприклад, з ростом застосунку підтримка і розширення значно ускладняються, що проілюстрував Мартін Фаулер [5] на діаграмі порівнюючи мікросервісні і монолітні підходи до написання програмного забезпечення (див. рис. 1.2).

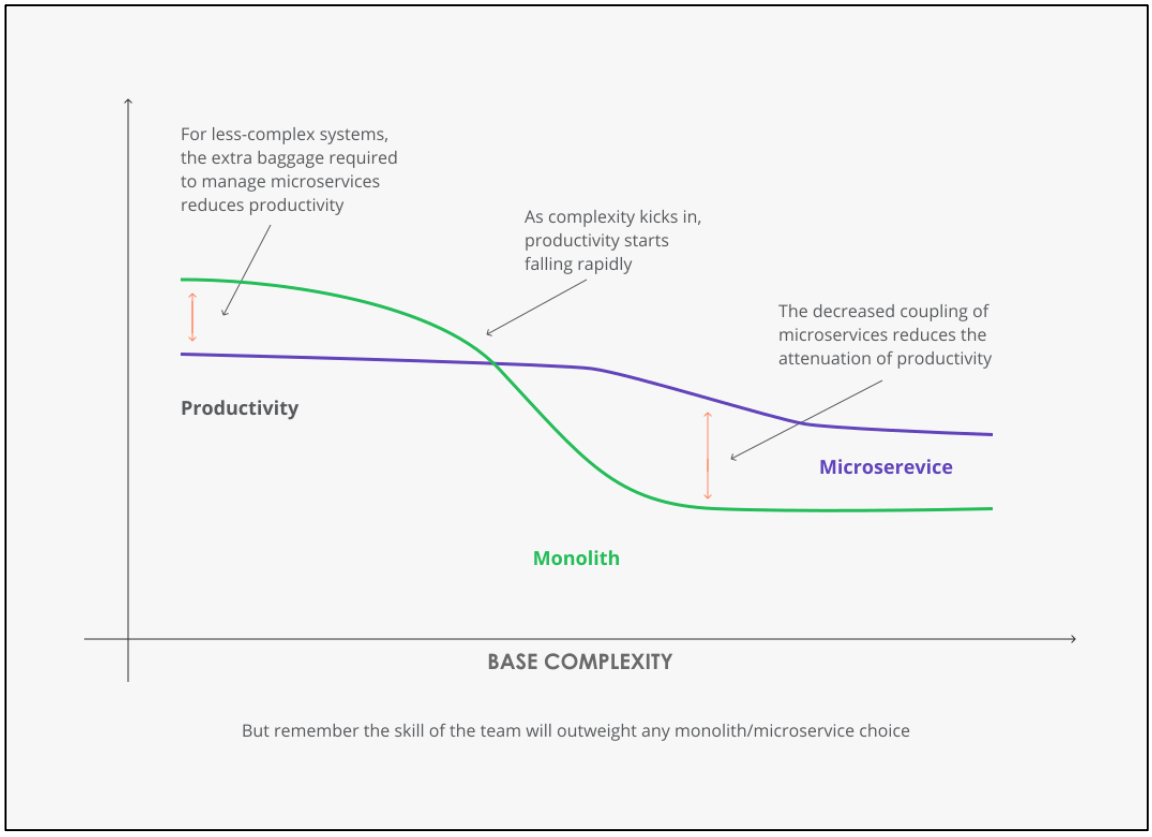

Рисунок 1.2 – Графік залежності затрат на підтримку веб-застосунку від його складності

Також, переходу до мікросервісної архітектури сприяло збільшення кількості користувачів.

За дослідженням Statista [6] кількість користувачів з 2005 року збільшилась майже в 5 разів (див. рис. 1.3).

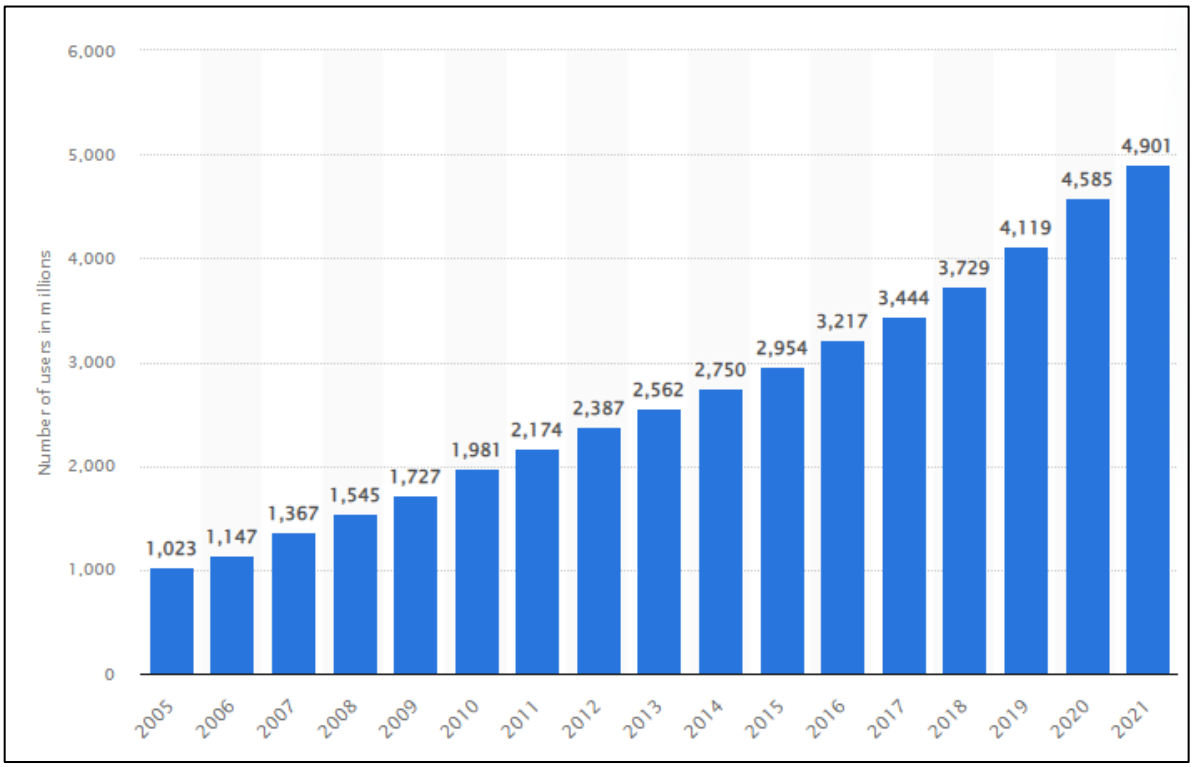

Рисунок 1.3 – Кількість користувачів Інтернет з 2005 по 2021 роки.

Тобто, з’явилася потреба у створенні програмних систем для серверної частини застосунку з високими характеристиками масштабування.

Також, на перехід до мікросервісної архітектури вплинула популяризація Agile технологій керування проектами. Підхід Scrum ідеально підходить для команд у рамках мікросервісних застосунків, через кількість людей у команді та гнучкість під час розробки таких сервісів.

Активний розвиток мікросервісної архітектури припадає на початок 2010х років. Мартін Фаулер стандартизував підхід та дав характеристику мікросервісній архітектурі у 2014 році [5]. З переваг можна назвати гнучкість, бо команди складаються з невеликої кількості людей, легке масштабування, бо кожен сервіс є незалежним та може масштабуватися під час високого навантаження саме на його функціонал.

Розглянемо, як виглядає мікросервісна архітектура на прикладі переробленої системи Uber, монолітну версію якої ми розглянули на рисунку 1.1, мікросервісна архітектура зображена на рисунку 1.4.

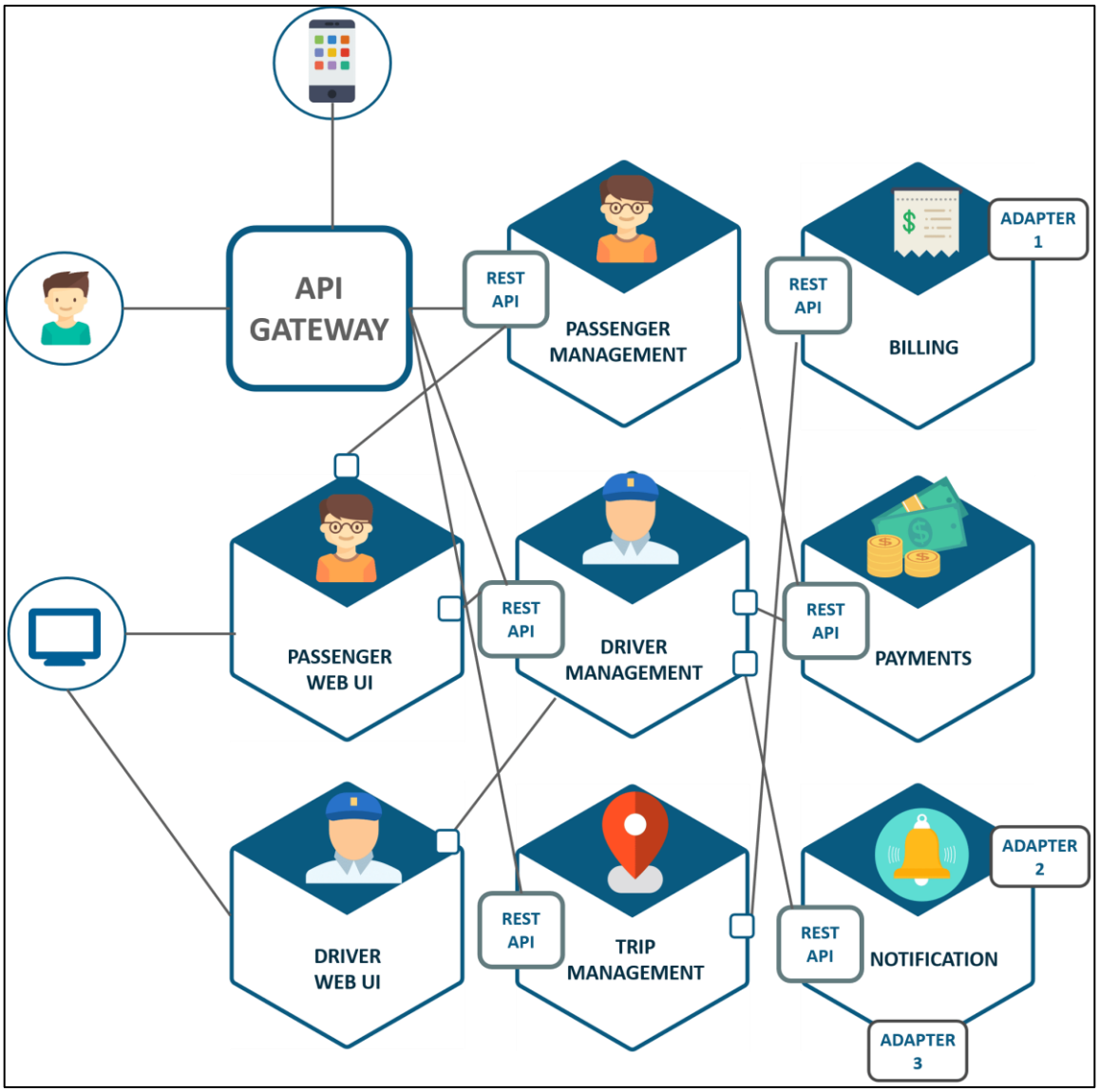

Рисунок 1.4 – Мікросервісна архітектура Uber.

На рисунку 1.4 ми бачимо, що важливою характеристикою мікросервісної архітектури є зв’язок між сервісами, згадаємо, що мікросервіси розробляються незалежно один від одного та пов’язані крізь запити. Так, наприклад, у Amazon при загрузці сторінки йде виклик 100-150 сервісів для отримання даних, а Netflix на серверній частині може викликати до 6 внутрішніх сервісів на кожен запит [2].

Тому час на виконання запиту напряму залежить від швидкості передачі даних між сервісами.

Особливо цей час важливий враховуючи нові підходи та шаблони для побудування мікросервісних застосунків. Один з найновіших шаблонів, що з'явився у 2021 році, та вже є широко розповсюдженим - BFF (Backend for Frontend).

Основною ідеєю цього шаблона є агрегація та фільтрація даних з багатьох сервісів для конкретної частини клієнтського застосунку, що делегує усю логіку, яка була на клієнтській частині застосунку до серверної. Діаграму цього архітектурного шаблону наведено на рисунку 1.5.

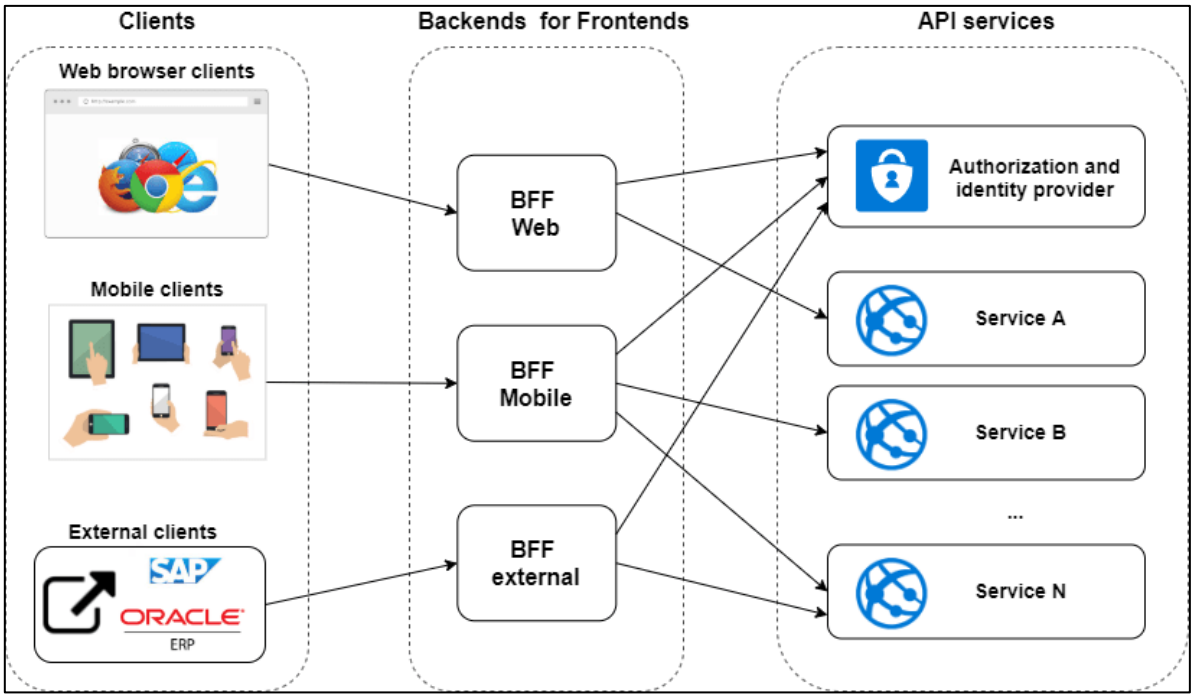

Рисунок 1.5 – Діаграма архітектурного шаблона BFF.

У цьому шаблоні роль міжсерверного зв'язку між внутрішніми елементами системи стає очевидною, бо кожен BFF може звертатися до великої кількості сервісів, і швидкість запитів з BFF буде напряму впливати на швидкість обробки основного запиту користувача. Цей параметр впливає на показник відмов, за даними Google, шанс того, що користувач не дочекається відкриття сторінки зростає на 30%, якщо час завантаження сторінки перевищує 1 секунду [7]. Цей факт

впливає на кількість користувачів сайту, а тому і на його прибуток. Тому забезпечення найоптимальнішого підходу для міжсерверного зв’язку є дуже важливим при побудові сучасних мікросервісних застосунків.

1.2 Виявлення проблем та актуалізація рішень

Швидкий розвиток нових підходів до написання мікросервісних застосунків створює нові задачі для вирішення. Так, при взаємодії мікросервісів між собою, а особливо з використанням шаблону BFF ми бачимо, що швидкість системи починає напряму залежати від правильно обраної моделі міжсерверного зв’язку компонентів.

Дослідження методів реалізації міжсервісного зв’язку у серверних застосунках є актуальним. Так, наприклад, у роботі «Comparison of REST and GraphQL Interfaces for OPC UA» [8] було розглянуто такі підходи як REST та GraphQL. Дослідниками було розглянуто три критерії:

- складність формування запиту;
- швидкість виконання запиту;
- сумісність для повторного використання.

У дослідженні було проведено серію тестів, що порівнювали ці підходи для різних об’ємів даних та кількості запитів. Автори цієї роботи прийшли висновку, що GraphQL виграє по усім зазначеним вище параметрам, та рекомендували підхід з використанням технології GraphQL.

Однак, це дослідження не покриває такі параметри, як складність підтримки та реалізації і проведено воно для специфічної системи, а не в узагальненому виді. Також це дослідження не покриває нову технологію від Google під назвою gRPC, що останні роки набирає популярність.

Інша робота «GraphQL or REST for Mobile Applications» [9], яка спрямована саме на дослідження впливу підходу побудови міжсерверного зв’язку на якість

програмного забезпечення, тобто на його здатність до розширення та складність підтримки. Дослідник порівняв показники серед одного API побудованого на основі підходу з GraphQL та два API на основі REST, які відрізнялися кількістю доступних інтерфейсів. Висновком цієї роботи є ствердження, що підхід з використанням GraphQL виграє REST за показником ефективності роботи застосунку та складності коду.

Розглянуті в обох дослідженнях методи не включають у себе новий підхід від компанії Google з використанням gRPC. Порівняння цього підходу є доречним, та навіть необхідним, бо він використовує новий протокол HTTP2.1 та новий формат для передачі даних, що може значно вплинути на результати дослідження.

1.3 Постановка задачі

Вибір доцільного методу налагодження міжсерверного зв’язку потрібен в усіх сучасних веб-застосунках побудованих на основі мікросервісної архітектури. При розробці доступно декілька підходів для реалізації взаємодії.

Одним з недоліків різноманітності варіантів для вирішення цієї задачі є те, що не має стандартів, що можна застосувати при вибору технології під час проектування нового застосунку. Це може призвести до того, що будуть обрані не доцільні для використання технології з точки зору продуктивності [8], а такі, з якими розробник мав позитивний досвід, або найбільш поширені.

Правильний вибір під час проектування системи позитивно позначиться на швидкодії системи, тобто на користувацькому досвіді. За дослідженнями від Google користувачі надають перевагу веб-застосункам, які працюють швидше та завантажуються за менший обсяг часу [7]. Також швидкість завантаження веб-сайтів впливає на позицію у пошуковій видачі Google, що сприяє розвитку системи за рахунок нових користувачів [10]. Перспективними напрямами досліджень, на основі результатів даного дослідження, можуть бути системи орієнтовані на

покращення якості зображень [11, 12], підвищення продуктивності обробки пошукових запитів [13], а також покращення ефективності вбудованих систем [14].

Це робить дослідження методів та архітектурних рішень міжсерверної взаємодії з точки зору швидкості справді важливим. Метою цього дослідження є порівняння таких методів та підходів як REST, GraphQL, gRPC для забезпечення міжсерверної взаємодії елементів мікросервісної архітектури.

Виходячи з усього перерахованого вище, в рамках дослідження необхідно вирішити наступні завдання:

- розглянути існуючі методи та архітектурні підходи побудови міжсерверної взаємодії;
- проаналізувати та обрати підходи та методи для побудови міжсерверної взаємодії для проведення дослідження;
- проаналізувати існуючі підходи до вимірювання швидкодії серверних застосунків;
- проаналізувати можливість впливу способу міжсерверного зв’язку на швидкодії серверних застосунків;
- визначити метрики для проведення експерименту і подальшого оцінювання
- визначити функціональні вимоги до серверного застосунку, що буде використано для проведення експерименту, та розробити його;
- виміряти та підрахувати значення обраних метрик для кожного з методів та підходів побудови міжсерверної взаємодії;
- порівняти та проаналізувати отримані дані;
- надати рекомендації щодо використання кожного з методів.

2 ВИБІР МЕТОДІВ УПРАВЛІННЯ СТАНОМ

2.1 Аналіз існуючих методів міжсерверного зв’язку

Обмін даними був критично важливим аспектом при побудові програмних систем з їх появи. Хоча у самому початку багато обміну даними відбувалося всередині мейнфрейму компанії, у певний момент з’явилася потреба поділитися інформацією з іншими комп’ютерами. Ранні форми обміну були фізичними, а саме оператори завантажували дані на котушки магнітної стрічки. Ці стрічки потім транспортували від одного мейнфрейму до іншого. Зі стандартизацією мережевого зв’язку, обмін даними почав відбуватися в цифровому вигляді по телефонних лініях і мережевих проводах за допомогою різноманітних протоколів: Telnet, SMTP, FTP та HTTP.

Згодом, стандартизували формат обміну даних, і з’явилися такі формати як XML, JSON, Protocol Buffers. Ця стандартизація форматів даних призвела до архітектурного дизайну, що позиціонує API у якості основи архітектури. Такий підхід до дизайну архітектури програмних застосунків дозволяє великій кількості клієнтів взаємодіяти з програмним забезпеченням в одному і тому ж форматі запитів.

API принесли новий підхід в архітектурний дизайн, але все ще не має узгодженого виду взаємодії між ними, кожна компанія має своє бачення щодо підходів до написання, яким саме чином клієнти будуть взаємодіяти з програмним інтерфейсом. Існує багато методів для забезпечення спілкування з API, найпопулярнішими з них є REST, GraphQL та gRPC. Так як API є основою сучасного Інтернету, важливо, щоб він був швидким, надійним, з передбачуваною поведінкою та безпечним.

Таким чином, через зазначені фактори важливо дослідити, який з цих форматів більше підходить для вирішення задачі зв’язку між компонентами складної мікросервісної системи.

2.2 Огляд REST

Спочатку REST створювався як набір практик для створення розподілених систем, таких як World-Wide Web, Роєм Філдінгом [15]. Ці правила описують архітектурні принципи Web та покращують його архітектурні параметри, зокрема масштабованість. Обов’язкові архітектурні обмеження описані Роєм:

а) клієнт-серверна модель для розділення обов’язків: клієнт відповідає за відображення даних та реагування на його дії, а сервер відповідає за зберігання і оновлення даних;

б) відсутність стану каже про те, що сервер не повинен зберігати ніякої інформації про клієнта, і клієнт відповідальний за збереження стану сесії. Тому кожен запит повинен містити усю необхідну інформацію для його обробки, включаючи наприклад дані авторизації. Цей пункт підвищує масштабованість системи, але підвищує кількість даних, що треба передавати з кожним запитом;

в) кешування дозволяє відповідати на ідентичні запити однаковими відповідями, збереженими заздалегідь. Це підвищує масштабованість, швидкість відповідей та навантаження на мережу. Актуальність даних контролюється спеціальним хедером Cache-Control;

г) однорідний інтерфейс усі компоненти мають мати однорідний інтерфейс, завдяки ідентифікації ресурсів, повідомлень, що описують себе, гіпермедія як рушія застосунку та маніпуляцією ресурсів через представлення;

д) багатошарова система складається з багатьох архітектурних шарів, що взаємодіють лише з шаром нижче та вище. Це зменшує складність системи, але додає часу на роботу програми. Цей підхід лежить в основі підходу зрівноваження навантаги (Load Balancing).

Підхід запропонований Роєм став де факто архітектурним стилем створення веб застосунків і часто використовується як синонім для будь якого API. API

побудований за такими принципами має назву RESTful, список практик для його створення наведений нижче:

1. Дані розподілені по ресурсам та кожен ресурс ідентифікується за допомогою URI [16];
2. HTTP методи (POST, GET, PUT / PATCH, DELETE) використовуються для операцій створення, читання, оновлення та видалення [17];
3. Коректне використання статус кодів HTTP запитів, наприклад 201, що означає створення, для відповідей на POST запити після успішного створення ресурсу [17].

Вигляд програмного інтерфейсу системи створеної за цими принципами наведено на рисунку 2.1.

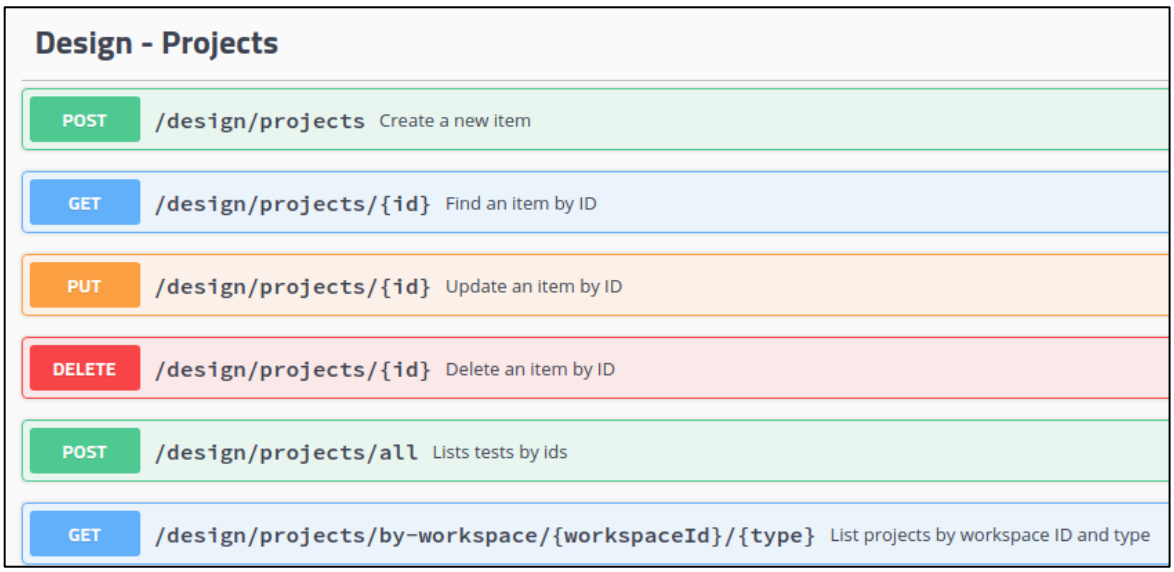

Рисунок 2.1 – Приклад RESTful API.

2.3 Огляд GraphQL

GraphQL це мова запитів і маніпуляцій даними для виконання запитів на сервері [18]. Перша робота над цим проектом розпочалася у 2012 році компанією Facebook, коли вони працювали над оптимізацією операцій читання даних на

мобільних застосунках [19]. У 2015 році його зробити проектом з відкритим початковим кодом. Багато світових компаній почали використовувати такий метод роботи з API, серед них такі гіганти як Airbnb, GitHub, Netflix та Twitter [20]. Найчастіше використовується з протоколом HTTP 1.1, так як і REST.

GraphQL, як альтернатива REST, має інший спосіб роботи з програмним інтерфейсом, чим сильно відрізняється від свого конкурента. Такими відмінностями насамперед є те, що запити відправляються на один і той же ресурс, а HTTP методи не мають семантичного значення [19]. Дата моделюється у виді графу, а структура даних описана схемою. Схема також містить в собі інформацію над операціями, які надає серверний застосунок для даних. Приклад запиту до API з реалізацією підходу GraphQL наведено на рисунку 2.2.

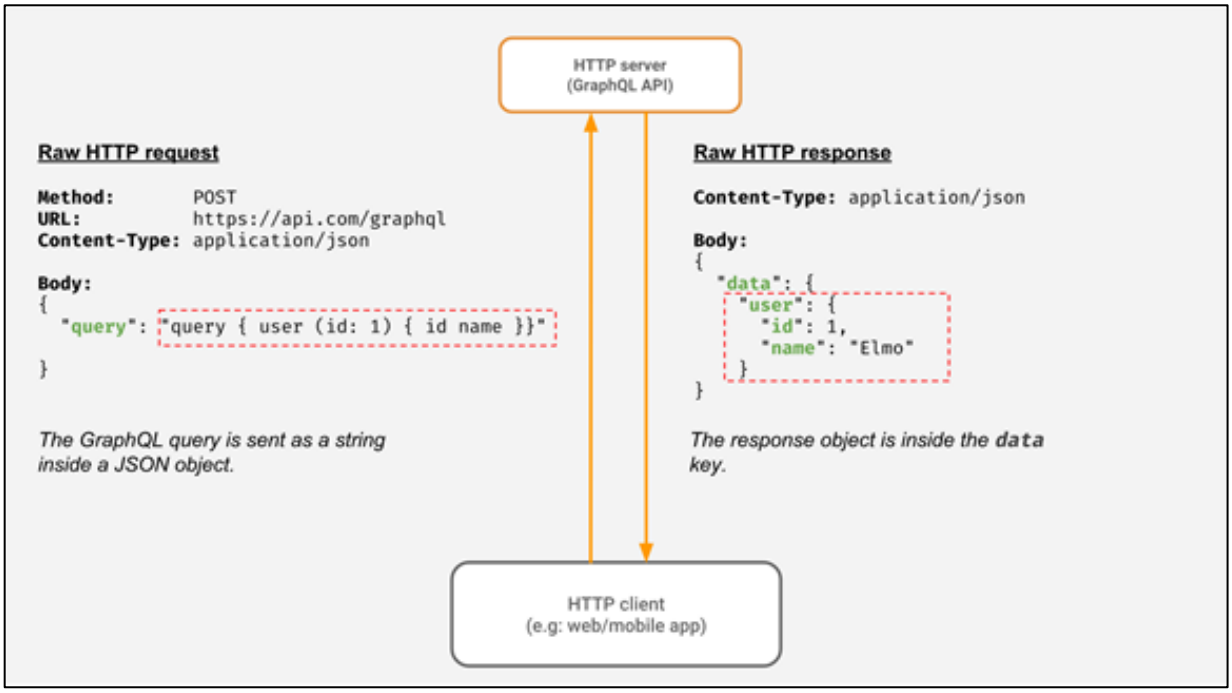

Рисунок 2.2 – GraphQL API.

Основні архітектурні принципи побудови API у підході GraphQL описані нижче [15]:

а) ієрархічність. Запити повинні бути ієрархічними та повторювати структуру даних. Запит та відповідь мають однакову структуру;

б) строга типізація. Схема даних і типів строго визначена, таким чином сервер може перевіряти правильність запитів;
в) структура відповіді формується клієнтом. Клієнт у запиті указує дані та їх структуру. Відповідь сервера строго відповідає структурі клієнтського запиту;
г) інтроспективний. Система побудована з використанням GraphQL може бути повністю досліджена за допомогою запитів. Для цього використовується програмне забезпечення GraphiQL [20].

2.4 Огляд gRPC

gRPC це фреймворк для віддаленого виклику процедур з відкритим початковим кодом, який використовується для високошвидкісної та ефективної комунікації між мікросервісами. Розроблена компанією Google у 2015 році, як нове покоління інфраструктури Stubby [21]. У 2017 році став проектом Cloud Native Computing Foundation [22].

gRPC використовує протокол HTTP/2, що відрізняється від попередньо описаних технологій та дає значні переваги протоколу нового покоління. Для опису інтерфейсу використовується мова Protobuf. Використання HTTP/2 робить неможливим реалізацію клієнта gRPC у браузері, тому цей підхід використовується саме у випадку, коли потрібно налаштувати зв’язок у системі з мікросервісною архітектурою.

Низка великих компаній після оприлюднення фреймворку запровадили у своїх програмних системах цю нову технологію. У список цих компаній входять: Netflix, Docker, Cisco, Spotify, Google, Dropbox [22].

gRPC, на відміну від попередніх методів спілкування, підтримує чотири види міжсерверного спілкування. В залежності від потреб системи, та даних, що будуть

передаватися, можна обрати саме той спосіб, який буде найефективнішим. Повний список цих видів зазначено нижче:

а) унарний. Клієнт надсилає один запит на сервер, та отримує одну відповідь. Такий підхід є класичним і добре відомим усім;

б) серверний потоковий. Клієнт надсилає запит на сервер та у відповідь отримує потік даних для читання послідовності повідомлень. Клієнт читає усі повідомлення до останнього. gRPC гарантує правильну послідовність при читанні;

в) клієнтський потоковий. Той же підхід, що і у минулому пункті, але навпаки. Тепер клієнт відправляє на сервер послідовність, а він її читає до кінця;

г) двоспрямована потокова передача. Тут дві сторони – клієнт і сервер, відправляють один одному послідовності повідомлень, використовуючи потоки читання-запису. Потоки читання та запису незалежні один від одного, тому можливі ситуації, коли спочатку клієнт надсилає усі дані для запиту, а потім отримує весь результат. Або отримує результати по ходу обробки кожного елементу даних у потоці.

Описані потокові підходи до обробки запитів стали можливими через той факт, що gRPC використовує протокол HTTP/2. Цей протокол має низку переваг ніж його попередник. По-перше, він розбиває зміст запиту на маленькі шматки, для більш ефективної передачі по мережі. По-друге, вже зазначена вище підтримка потоків, тобто дуплекс зв’язок.

У gRPC клієнтський застосунок може напряму визивати методи на серверному застосунку, який знаходиться на іншому комп’ютері, у такому виді, ніби цей метод написаний на клієнті.

Описаний підхід є типовим для систем з віддаленим викликом процедур, та базується він на ідеї опису методів на сервері у файлах Protobuf, туди входять назва методу, параметри та тип даних, що метод повертає. Потім сервер реалізує описаний у Protobuf файлі інтерфейс і запускає у собі gRPC сервер для обробки

запитів. Клієнтські застосунки, у свою чергу, мають метод заглушку, через який може звертатися до методів на сервері.

Треба зазначити, що застосунки, які спілкуються за допомогою gRPC можуть бути написані на різних мовах програмування, тобто не має обмежень у використанні однієї мови програмування у системі.

Приклад системи з використанням gRPC наведено на рисунку 2.3.

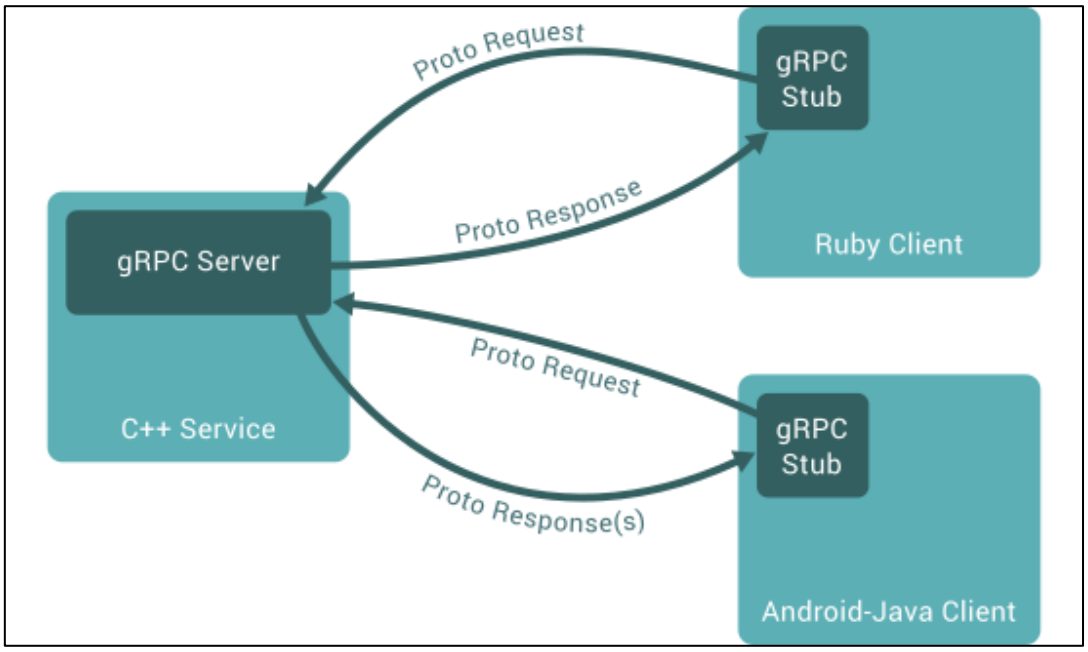

Рисунок 2.3 – приклад зв’язку з використанням gRPC.

Опис інтерфейсів відбувається з використанням мови Protobuf. Приклад опису сервісу наведено нижче.

```
service Greeter {
    rpc SayHello (HelloRequest) returns (HelloReply) {}
}
message HelloRequest {
  string name = 1;
}
message HelloReply {
  string message = 1;
}
```

У наведеному описі, ми бачимо, що опис сервісу та його методів з типами параметру та значення, що повертається. Таким чином, стандартизований підхід до

опису інтерфейсу з легкістю дозволяє використовувати gRPC у системах з мікросервісами написаними на різних мовах програмування.

Під час написання серверного застосунку з використанням gRPC, не треба писати код, що поєднує серверні застосунки. Усе генерується на основі Protobuf файлу, за допомогою protoc компілятора. Цей процес зображено на рисунку 2.4.

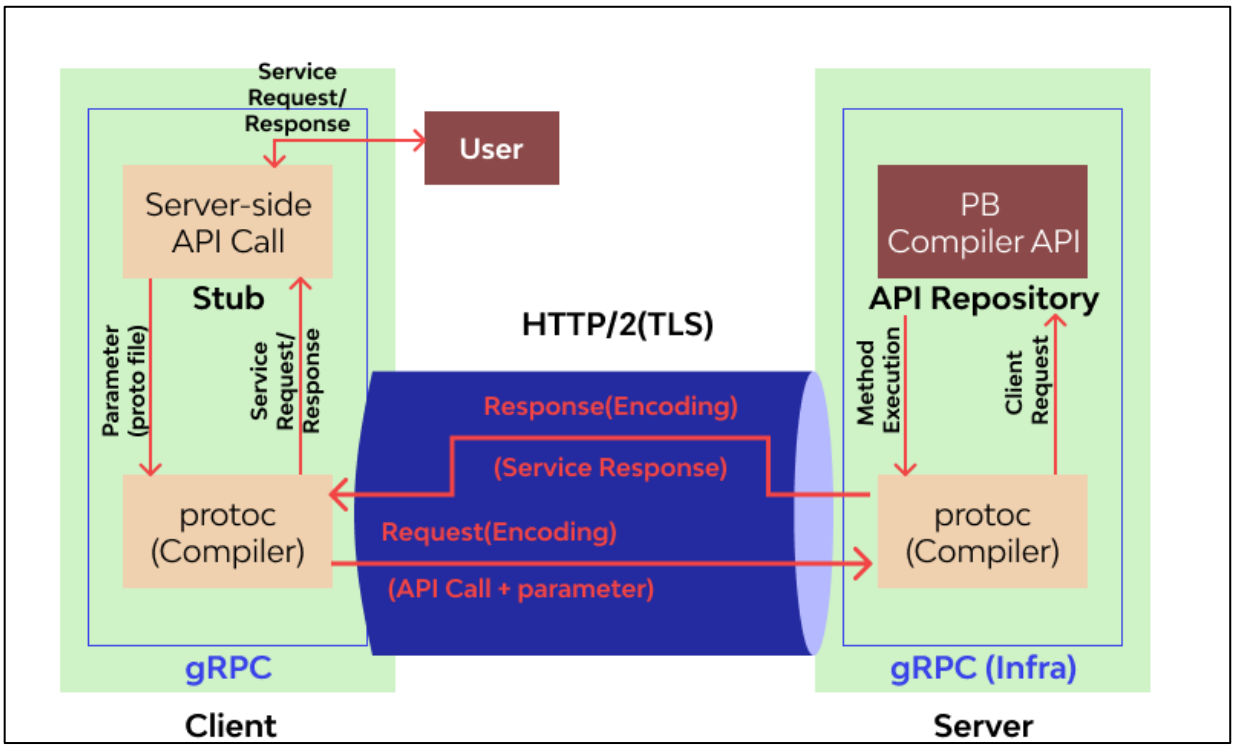

Рисунок 2.4 – схема роботи gRPC системи.

Із мінусів такого підходу треба зазначити те, що немає підтримки кешування даних через використання методу POST. Дані передаються у бінарному виді, через що людина не може їх прочитати, з одного боку це плюс, бо покращує безпеку застосунку, але, з іншого боку, людині важче побачити дані під час виправлення помилок у системі.

3 ОПИС ПРОГРАМНОЇ РЕАЛІЗАЦІЇ

3.1 Архітектура програмної системи

Система буде реалізована на основі мікросервісної архітектури. Мікросервісна архітектура найрозповсюдженіший підхід у сучасній індустрії розробки програмного забезпечення та використовується для написання складних, здатних до масштабування систем [15]. Система буде розділена на API 3 сервіси, кожен з яких буде реалізовувати свій підхід до міжсерверної взаємодії, BFF та клієнтський застосунок для тестування навантаженням. Такий підхід дозволить забезпечити незалежну розробку, тестування та розгортання елементів системи для проведення експерименту.

Серверна частина застосунку буде включати три окремі веб-сервіси, а саме RESTful, gRPC та GraphQL. Ці три різних методи реалізації API будуть використовуватися для проведення експерименту з порівнянням їх продуктивності, гнучкості та придатності для різних випадків використання. Веб-сервіси будуть звертатися до бази даних, яка знаходиться у хмарному середовищі. База даних знаходиться у хмарному середовищі, що дозволить забезпечити високу швидкість доступу до даних.

Також, буде реалізовано додатковий сервіс BFF. Саме він буде другим елементом у міжсерверній взаємодії, та буде пов’язувати клієнтський застосунок з серверними. BFF було обрано через те, що цей підхід до реалізації серверної частини застосунку є розповсюдженим у реальних застосунках великих компаній. Тому дослідження з урахуванням цього підходу буде актуальним.

Крім того, архітектура передбачає використання спеціального додатку для тестування навантаженням. Це дозволить провести реалістичне тестування, щоб оцінити продуктивність і масштабованість кожного методу реалізації API, а також порівняти їх з точки зору відповідності до вимог у різних сценаріях реалізації серверних застосунків.

Для кращого розуміння, наведемо графічне представлення архітектури описаної системи. Архітектура системи зображена на рисунку 3.1.

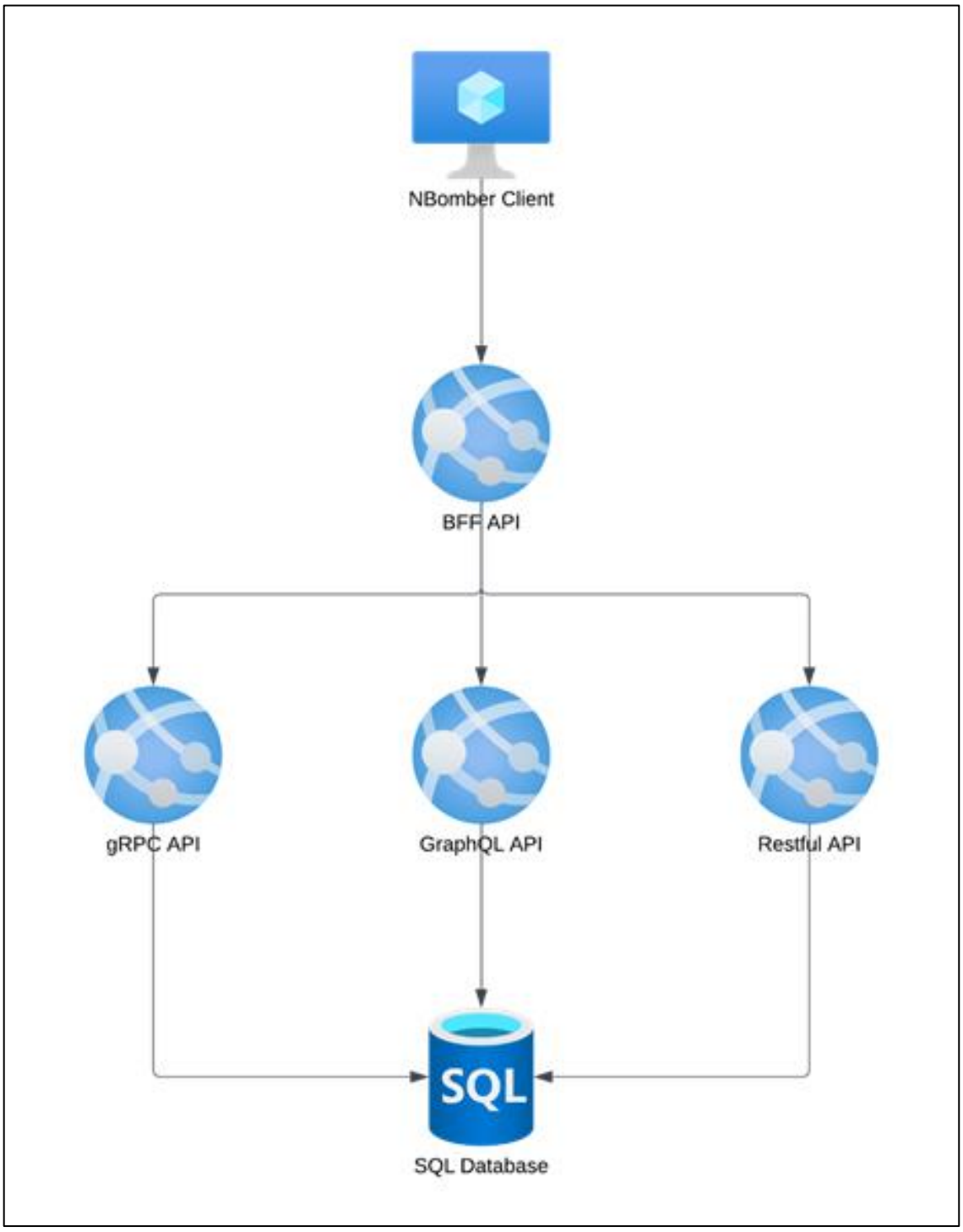

Рисунок 3.1 – Архітектура програмної системи.

Такий підхід до розробки системи є оптимальним для проведення експерименту з порівнянням різних методів реалізації API. Завдяки мікросервісній

архітектурі та використанню окремих веб-сервісів можливо забезпечити незалежну розробку, тестування та розгортання елементів системи. Використання трьох різних методів реалізації API дозволить провести дослідження їх продуктивності, гнучкості та придатності для різних випадків використання. Крім того, використання BFF як елемента міжсерверної взаємодії є актуальним та дозволить провести експеримент з урахуванням реальних потреб у сучасній індустрії розробки програмного забезпечення. Тестування навантаженням дозволить провести реалістичне тестування та порівняти продуктивність та масштабованість кожного методу реалізації API, а також оцінити їх відповідність вимогам у різних сценаріях реалізації серверних застосунків.

3.2 Вибір засобів розробки

Програмна система буде написана на платформі .NET 6 з використанням мови програмування C# версії 10. Такий вибір обумовлено швидкодією платформи, спрямованістю на серверні застосунки та великою кількістю бібліотек і інструментів. Такі фактори впливають на коректність результатів експерименту та ефективність реалізації.

Для розробки програмної системи для проведення експерименту було обрано середовище Visual Studio 2022 з плагіном ReSharper. Така комбінація дозволяє отримати зручний користувацький інтерфейс та можливості статичного аналізу коду, які надає ReSharper. Це дозволяє покращити якість коду та зменшити кількість помилок під час його написання, що може позитивно вплинути на результати експерименту.

Також, Visual Studio 2022 дозволяє полегшити та візуалізувати роботу з системою контроля версій GIT, яка була використана під час реалізації програмного забезпечення для проведення експерименту необхідного для

дослідження у цій роботі. Це дозволило ефективно керувати версіями програмного забезпечення та спростити його розробку та тестування.

3.3 Вибір бази даних

Вибір бази даних є критичним для успішного проведення експерименту, оскільки база даних відіграє важливу роль у забезпеченні високої продуктивності та масштабованості досліджуваних API. Для цього експерименту було вирішено використовувати Azure SQL Database [23] як систему управління базами даних.

Azure SQL Database - це повністю керована, хмарна база даних, яка забезпечує високу доступність, відмовостійкість та масштабованість. Azure SQL Database широко використовується в індустрії та є випробуваною часом базою даних, що ідеально підходить для нашого дослідження.

Azure SQL Database має такі переваги [23]:

1. Висока доступність і відмовостійкість - база даних Azure SQL забезпечує високий рівень доступності та надійності, забезпечуючи збереження даних і працездатність додатків у випадку відмови;
2. Масштабованість - Azure SQL Database дозволяє масштабувати базу даних залежно від вимог системи. Це дозволяє підтримувати продуктивність додатку при збільшенні обсягу даних та навантаження на систему;
3. Зручність управління - Azure SQL Database є повністю керованою базою даних, що означає, що не потрібно встановлювати та конфігурувати базу даних, а також не потрібно займатися налаштуваннями засобів моніторингу та зняття метрик.

Оскільки цей експеримент має на меті порівняти продуктивність та масштабованість програмних інтерфейсів, Azure SQL Database є ідеальним вибором для цього, оскільки вона забезпечує високу доступність, відмовостійкість та масштабованість бази даних. Крім того, Azure SQL Database є повністю

керованою базою даних, що дозволяє виконувати моніторинг бази даних без необхідності встановлювати та налаштовувати додаткові програми. Це дозволяє збільшити продуктивність експерименту, оскільки можна сконцентруватися на експерименті та аналізі результатів, не витрачаючи час на управління базою даних. Крім того, Azure SQL Database підтримує широкий спектр мов програмування та інструментів розробки, що дозволяє використати обрану мову програмування C# для розробки та тестування програмних інтерфейсів. З цими функціями Azure SQL Database стає ідеальним вибором для проведення дослідження даної кваліфікаційної роботи.

3.4 Робота з даними

У даному експерименті для доступу до даних, збережених у базі даних Azure SQL Database, ми будемо використовувати Entity Framework Core 7 - популярний вибір для шару доступу до даних, який широко використовується у сучасній розробці програмного забезпечення [24].

Завдяки використанню Entity Framework Core 7, ми отримаємо значну скорочення часу розробки, оскільки Entity Framework дозволяє працювати з об'єктами даних, а не з низькорівневими операціями бази даних. Крім того, це спростить розробку та забезпечить стандартизацію коду.

Також, Entity Framework Core 7 забезпечує зручність в роботі з даними завдяки можливості використання об'єктно-орієнтованого підходу, який дозволяє працювати з даними як з класами та об'єктами. Використання такого підходу спрощує розробку програмного забезпечення та робить її більш зрозумілою.

Таким чином, для проведення експерименту, під час якого будуть виконуватись нескладні запити, Entity Framework Core 7 є оптимальним варіантом, оскільки він забезпечує простість та швидкість розробки, що в свою чергу дозволяє швидко та ефективно отримувати результати.

3.5 Реалізація API

Реалізація API є важливим елементом експерименту, оскільки саме через нього будуть отримані дані, що будуть оброблені та проаналізовані. У цьому експерименті для реалізації API будуть використані мова програмування C# та технологія Web API.

У якості платформи було обрано .NET 6, вона забезпечує швидку та ефективну розробку програмного забезпечення для веб-додатків. Зокрема, використання .NET 6 дозволяє забезпечити високу продуктивність та оптимізацію роботи з пам'яттю, що особливо важливо для експерименту, де необхідно обробляти великі обсяги даних.

Для створення API використовуватиметься технологія Web API, яка забезпечує можливість звернення до веб-сервера через HTTP-протокол та отримання відповіді у форматі JSON. Використання цієї технології забезпечує високу доступність та швидкість взаємодії між клієнтом та сервером.

Реалізовані API забезпечать можливість взаємодії з базою даних та отримання даних для подальшого аналізу та порівняння продуктивності та масштабованості різних програмних інтерфейсів.

3.5.1 Реалізація RESTful

Для реалізації звичайного RESTful API використаємо стандартні засоби, які доступні у фреймворку ASP.NET. Реалізовано сервер який оброблює запити за такими URL:

а) GET /api/Recipes/all - отримати список всіх рецептів;
б) GET /api/Recipes/partial - отримати список рецептів з частковою інформацією (без списку кроків приготування);

в) GET /api/Recipes/{id} - отримати детальну інформацію про рецепт з заданим id.

3.5.2 Реалізація GraphQL

GraphQL - це мова запитів та середовище виконання запитів, що дозволяє отримувати тільки необхідні дані. GraphQL використовується як альтернатива REST API і забезпечує більш гнучкий та ефективний спосіб отримання даних з сервера.

У реалізації GraphQL API для цього дослідження було використано бібліотеку HotChocolate для створення GraphQL API на базі .NET Core.

Першим кроком є додавання сервісу GraphQLServer до контейнера залежностей за допомогою методу AddGraphQLServer. Метод AddQueryType дозволяє визначити запити, які можуть бути виконані через GraphQL API. Код наведено нижче:

```
builder.Services.AddGraphQLServer()
    .AddQueryType<GetRecepiesQuery>()
    .RegisterDbContext<DataContext>()
    .AddProjections().AddFiltering().AddSorting();
```

У цьому коді ми реєструємо клас GetRecepiesQuery як запит, який може бути виконаний через GraphQL API. Ми також реєструємо клас DataContext, щоб HotChocolate міг знайти його під час виконання запитів.

```
public class GetRecepiesQuery
{
    [UseProjection]
    [UseFiltering]
    [UseSorting]
    public IQueryable<Recipe> GetRecipes(DataContext context) => context.Recipes.AsNoTracking();
}
```

У класі GetRecepiesQuery визначимо запит для отримання рецептів. Запит повертає IQueryable, який потім буде оброблений HotChocolate. Атрибути

UseProjection, UseFiltering та UseSorting дозволяють користувачам виконувати проекції, фільтрування та сортування даних.

Наостанок, використаємо метод MapGraphQL для додавання GraphQL маршруту до нашого додатку.

```
app.MapGraphQL("/graphql");
```

Таким чином, GraphQL API готове для використання.

3.5.3 Реалізація gRPC

gRPC - це відкритий протокол від Google для побудови розподілених систем, що підтримує різні мови програмування. gRPC використовує протокол HTTP/2, що дозволяє використовувати більше одного запиту на одному з'єднанні, що спрощує процес розробки та підтримки мережевих додатків. gRPC забезпечує механізми автоматичної генерації коду, які дозволяють автоматизувати створення та розробку мережевих додатків.

У цьому випадку, gRPC API використано для отримання даних з бази даних та передачі їх клієнту. Було створено сервіс RecipesGrpcService, який містить два методи: GetRecipes та GetPartialRecipes. Обидва методи приймають порожній запит від клієнта та повертають відповідь, яка містить список рецептів у вигляді повного списку або списку з обмеженим набором полів.

Було використано protobuf для передачі даних між сервером та клієнтом. Файл recipes.proto описує структуру даних, які будуть використані для передачі даних. У цьому файлі описано сервіс RecipesGrpcService та повідомлення, які використовуються у нашому API.

У RecipesService виконується взаємодія з базою даних та перетворення даних з бази даних в повідомлення protobuf. Було використано бібліотеку AutoMapper для зведення між сутностями бази даних та повідомленнями protobuf.

Для звернення до цього API можна використовувати будь-яку мову програмування, яка підтримує gRPC. Одним з популярних пакетів для створення клієнтського коду на C# є Grpc.Net.Client. Цей пакет дозволяє створювати клієнтські додатки, які можуть звертатися до gRPC-серверів за допомогою синхронного та асинхронного програмування.

3.6 Реалізація BFF

BFF (Backend for Frontend) - це шаблон проектування архітектури програмного забезпечення, який застосовується для побудови серверної частини додатків з великою кількістю клієнтів. Діаграма архітектури з використанням цього шаблону наведена на рисунку 4.1.

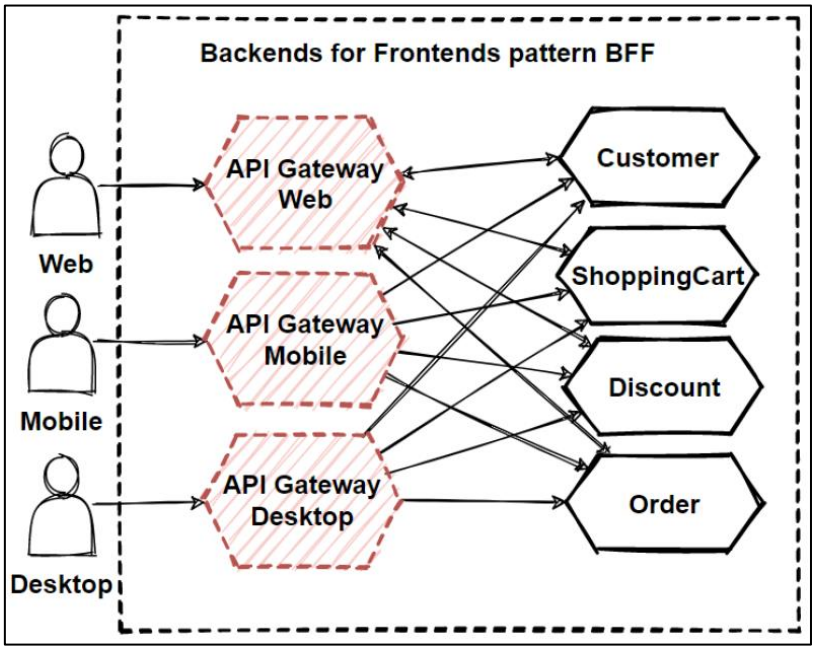

Рисунок 3.2 – Діаграма архітектури системи з шаблоном проектування BFF.

Цей шаблон дозволяє зменшити кількість запитів до сервера і покращити продуктивність додатку, створивши спеціальний проміжний рівень - Backend for Frontend.

В рамках цього шаблону створюється окремий серверний компонент, який відповідає за взаємодію з клієнтською частиною, виконуючи функції, необхідні для її роботи.

Для реалізації BFF створимо стандартний RESTful API, який буде включати в себе декілька клієнтів для роботи з серверами з реалізацією RESTful, GraphQL та gRPC. Особливостями реалізації є створення клієнтів.

Для клієнта GraphQL вимогами є побудова правильного запиту до сервера. Для реалізації цієї цілі було використано бібліотеки GraphQL.Client, GraphQL.Query.Builder та GraphQL.SystemTextJson.

Для клієнта gRPC клієнт створюється на основі protobuf файлу, створеного під час реалізації gRPC API. Для цього ми копіюємо цей файл і клієнтський застосунок за допомогою бібліотеки Grpc.Net.Client автоматично створює клієнта для gRPC API.

Для RESTful клієнта реалізуємо звичайні виклики за протоколом HTTP до веб-сервісу.

3.7 Реалізація клієнта для тестування навантаження

У цій науковій роботі ключовим фактором є ефективність, яку ми вимірюємо за допомогою фреймворку для тестування навантаженням NBomber, який базується на .NET і є проектом з відкритим програмним кодом. Цей інструмент дозволяє нам моделювати різні сценарії, що можуть виникати в реальних умовах, надаючи метрики про продуктивність веб-сервісів. Детальні звіти, що генерує NBomber, надають комплексний аналіз характеристик продуктивності програмних інтерфейсів, що є важливим фактором для прийняття обґрунтованого рішення щодо вибору відповідного інтерфейсу для конкретного проекту. У нашому випадку ми виміряємо характеристики різних API інтерфейсів та використаємо ці дані для порівняння і аналізу.

4 ПРОВЕДЕННЯ ЕКСПЕРИМЕНТУ

4.1 Методологія проведення експерименту

Основна ціль експерименту – порівняння ефективності різних підходів до реалізації міжсерверної взаємодії на основі розроблених веб-сервісів. Перевірка різних підходів дозволить визначити найбільш оптимальний метод реалізації міжсерверної взаємодії залежно від конкретних потреб системи. Для досягнення цієї мети, запропоновано використати визначений набір головних метрик, за якими будуть порівнюватися результати експерименту. Ці критерії є вирішальними для визначення ефективності кожної з технологій з точки зору середнього часу відгуку, пропускної здатності та розміру переданих даних.

а) середній час відповіді (Average response time, ART) – час, потрібний API для отримання запиту та надсилання відповіді клієнту, має вирішальне значення для визначення ефективності API. Чим коротший час відповіді, тим краща продуктивність API. Його можна виміряти в мілісекундах (мс) і розраховувати як різницю між часом, коли було зроблено запит, і часом, коли було отримано відповідь. ART можна визначити, як: $ART = \frac{\sum_1^n t_{n2} - t_{n1}}{n}$, де t_2 – час отримання відповіді, t_1- час відправки запиту, n – кількість запитів;

б) пропускна здатність (Throughput, TP) – кількість запитів, які може обробити система за одиницю часу. Він виражається в запитах за секунду (RPS) і може бути розрахований як кількість запитів, оброблених за певний проміжок часу, поділена на проміжок часу. Значення TP можна представити у вигляді: TP= $\frac{N}{T}$, де N – кількість оброблених запитів, T – часовий інтервал;

в) розмір переданих даних (Data Transfer Size, DTS) - вимірює обсяг даних, що передаються між клієнтом і сервером. DTS можна розрахувати як розмір даних отриманої клієнтом відповіді, плюс розмір даних, надісланих

на сервер. Розмір передачі даних може бути представлений DTS= $\frac{RS}{RD}$, де RS – розмір даних запиту, RD – розмір даних відповіді.

За допомогою вказаних критеріїв, наша мета полягає в повному порівнянні продуктивності програмних інтерфейсів REST API, gRPC та GraphQL. Визначені критерії дозволять нам виявити сильні та слабкі сторони кожної технології та надати рекомендації щодо найкращого підходу на основі конкретних випадків використання.

4.2 Підготовка набору даних

Вибір правильного набору даних для проведення дослідження з ефективності API інтерфейсів є критично важливим етапом у процесі оцінки продуктивності системи. Вірний вибір даних дозволяє отримати точні результати, які можуть бути використані для подальшого аналізу і порівнянь.

При виборі набору даних необхідно звернути увагу на його об'єм та структуру. Набір даних повинен бути достатньо великим, щоб забезпечити достатню кількість тестових запитів та обсягів даних, а також повинен відображати різні види запитів, які виконує система. Це дозволить визначити поведінку системи при різних типах запитів та навантаженнях. Також, необхідно звернути увагу на якість даних.

Для цього експерименту була розроблена модель бази даних з метою імітації системи керування рецептами.

Основною сутністю в цій моделі є сутність «рецепти», яка містить інформацію про різні рецепти, наприклад їх назву, опис, час приготування та харчову цінність. Ця сутність має зв’язок «один до багатьох» з двома іншими сутностями, а саме «інгредієнти» та «кроки». Кожен рецепт може містити кілька інгредієнтів і кілька кроків, які пов’язані з сутністю рецепта за допомогою зовнішніх ключів. Сутність «інгредієнти» містить інформацію про різні

інгредієнти, які використовуються в кожному рецепті, наприклад їх назву, кількість та одиницю вимірювання. Подібним чином об’єкт «кроки» містить інформацію про різні етапи приготування рецепту, наприклад час приготування, температуру та інструкції. Діаграма бази даних наведена на рисунку 4.1.

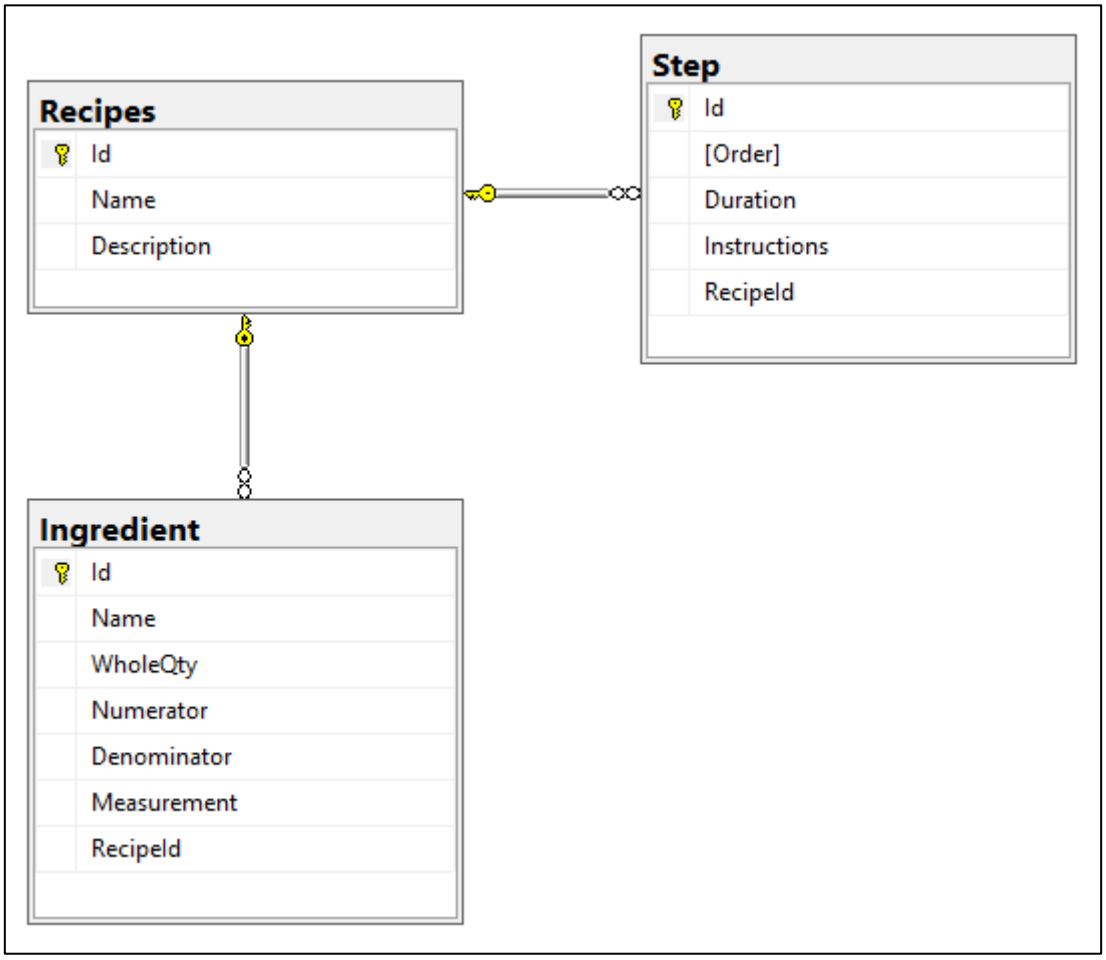

Рисунок 4.1 – Діаграма моделі бази даних для проведення експерименту.

Очікується, що ця структура даних забезпечить реалістичне представлення системи керування рецептами та використовуватиметься для перевірки продуктивності Restful, GraphQL і gRPC API під час отримання та обробки даних із цієї моделі.

Для генерації тестових даних використовується бібліотека Bogus, яка дозволяє генерувати фейкові дані на основі заданих правил. Для кожного типу даних (рецепт, інгредієнт, крок) створюється відповідний об'єкт Faker, який задає правила генерації кожного поля об'єкту.

Наприклад, для рецепта задаються правила генерації Id, Name та Description. Для інгредієнта задаються правила генерації Id, RecipeId (вибирається випадковий рецепт зі списку), Name, WholeQty, Numerator, Denominator та Measurement. Для кроку задаються правила генерації Id, RecipeId (вибирається випадковий рецепт зі списку), Order, Duration та Instructions.

Після того, як були задані правила генерації для кожного типу даних, викликається метод Generate, який приймає кількість об'єктів, які потрібно згенерувати. В нашому випадку, кожен список генерується з 50 рецептів, 100 інгредієнтів та 100 кроків (для кожного рецепту генерується 2 інгредієнти та 2 кроки).

Отже, дана реалізація генерації тестових даних на основі фейкових даних забезпечує достатньої рівень репрезентативності для дослідження ефективності API інтерфейсів.

4.3 Опис тестових сценаріїв

Для для проведення експерименту було обрано фреймворк для тестування NBomber, який може змоделювати необхідні рівні користувацького трафіку і забезпечити точні вимірювання часу відповіді, пропускної здатності та розміру переданих даних. Тестові сценарії будуть виконуватись на спеціалізованому тестовому середовищі з достатнім обладнанням для забезпечення незалежності продуктивності API від обмежень системи. Тестове середовище має налаштування, схожі на ті, що використовуються у реальних проектах, щоб забезпечити реалістичні умови тестування.

Тестування буде проводитись для декількох сценаріїв, а саме для повного отримання даних про сутність, та часткового отримання даних. Ці сценарії будуть використовувати відповідні API, створені раніше.

Для сценарію повного отримання сутності кожен API буде перевірено для випадку отримання повної сутності рецепта, включаючи всі пов’язані з ним інгредієнти та кроки. Час відповіді, пропускна здатність і розмір переданих даних кожного API вимірюватимуться для конфігурацій 10, 50 і 100 запитів у секунду. Використання ресурсів кожного API також буде відстежуватися під час тестових прогонів, щоб оцінити продуктивність кожного API за різних рівнів навантаження.

Для сценарію часткового отримання сутності кожен API буде перевірено для отримання підмножини полів сутності рецепта, наприклад назви рецепта, імен інгредієнтів або покрокових інструкцій. Час відповіді, пропускна здатність і розмір передачі даних кожного API вимірюватимуться для конфігурацій 10, 50 і 100 запитів на секунду. Використання ресурсів кожного API також буде відстежуватися під час тестових прогонів, щоб оцінити продуктивність кожного API за різних рівнів навантаження.

Результати отримані під час виконання сценаріїв тестування будуть проаналізовані, щоб виявити вузькі місця або обмеження продуктивності в різних типах API, а саме Restful, GraphQL і gRPC. Також буде оцінено масштабованість кожного API, щоб визначити його придатність для обробки високого рівня трафіку користувачів. Ці результати будуть використані для прийняття обґрунтованих рішень про те, який API найкраще підходить для різних типів додатків і випадків використання.

4.4 План експерименту

План експерименту — це багатоетапний процес, призначений для перевірки та порівняння продуктивності Restful, GraphQL і gRPC API за допомогою тестування навантаженням. У наступних параграфах детально описується кожен крок.

Для початку буде створено базу даних Azure та заповнено тестовими даними для сутності рецепта. Ці тестові дані включатимуть зовнішні ключі до сутностей інгредієнтів і кроків для створення зв’язку «один до багатьох». Це дозволить API отримувати дані з бази даних за допомогою складних запитів, імітуючи сценарії реального світу.

Наступним кроком є впровадження Restful, GraphQL і gRPC API для взаємодії з базою даних і надання сутності рецепта клієнтам. Усі API оброблятимуть однаковий набір запитів і повертатимуть ті самі дані, щоб забезпечити справедливе порівняння. Ці API розроблені таким чином, щоб бути масштабованими, ефективними та оптимізованими для продуктивності.

Після впровадження API буде розроблено серверну частину для сервісу BFF. Цей сервіс об’єднає три API в єдиному місці для клієнтської програми. BFF спростить для клієнта взаємодію з серверними застосунками та отримання бажаних даних від відповідного API.

Після впровадження веб-сервісів їх буде розгорнуто в Azure. Це забезпечить високу доступність служб для інструмента тестування та можливість моніторингу за допомогою вбудованих інструментів моніторингу Azure.

Для вимірювання продуктивності API під навантаженням буде реалізовано інструмент навантажувального тестування за допомогою NBomber, фреймворку навантажувального тестування на основі .NET. Цей інструмент генеруватиме контрольоване та масштабоване навантаження на API для вимірювання продуктивності за різних рівнів навантаження. Для кожного API буде виконано такі сценарії тестування: повне отримання сутності та часткове отримання сутності. Ці сценарії виконуватимуться протягом 5 хвилин для кожної з трьох конфігурацій запитів на секунду (10, 50 і 100) для кожного API.

Під час тестових запусків інформація моніторингу з Azure збиратиметься та аналізуватиметься. Ця інформація включатиме використання ресурсів, час відповіді, пропускну здатність і розмір переданих даних для кожного API за різних рівнів навантаження.

Результати буде порівняно та проаналізовано, щоб виявити будь-які вузькі місця або обмеження продуктивності в API Restful, GraphQL і gRPC. Результати використовуватимуться для прийняття обґрунтованих рішень щодо того, який API найкраще підходить для різних типів додатків і випадків використання.

4.5 Технічні характеристики обладнання

Для проведення експерименту дуже важливо враховувати технічні характеристики обладнання, що використовується під час тестування.

У якості клієнтського застосунку, що робить тестування навантаженням, буде виступати комп’ютер з характеристиками зазначеними у таблиці 4.2.

Таблиця 4.2 – Характеристики обладнання клієнтського застосунку

Процесор	AMD Ryzen 5 3600
Графічний процесор	RX 570
Оперативна пам’ять	16 GB
Мережевий адаптер	Intel AX200 (2.4 Гбіт/с)
Операційна система	Windows 11

Серверні застосунки, задля коректності експерименту, будуть працювати, як веб-сервіси у хмарі Azure. Розміщення кожної частини системи в подібному хмарному середовищі гарантує, що всі компоненти працюють у рівних умовах середовища виконання. Це допомагає запобігти будь-яким несправедливим перевагам чи недолікам і забезпечує рівні умови гри. Крім того, подібне хмарне середовище може допомогти оптимізувати керування та підвищити загальну продуктивність системи. Такий підхід забезпечить рівновагу у використанні ресурсів і максимально наблизить тестове середовище до реальних умов комерційних проектів.

Для серверних застосунків буде використовуватись план Azure Premium v3 P2V3, що включає в себе 8 віртуальних процесорів, 32 гігабайти оперативної пам’яті та мережеві інтерфейси з пропускною здатністю 25 Гбіт/с. У бази даних буде використано план Standard-series.

4.6 Результати експерименту

Для того, щоб оцінити продуктивність трьох різних API в умовах різного навантаження, було проведено серію тестів з використанням різних рівнів запитів за секунду (RPS). Метою тестування було дослідити, як кожен з API працює в умовах високого навантаження та які є відмінності в їх продуктивності. У рамках першого сценарію тестування, який включав повне отримання моделі об'єкта, було зібрано результати для трьох API та представлено на трьох різних графіках. На рисунку 4.3 зображено графік середнього часу відповіді для першого сценарію.

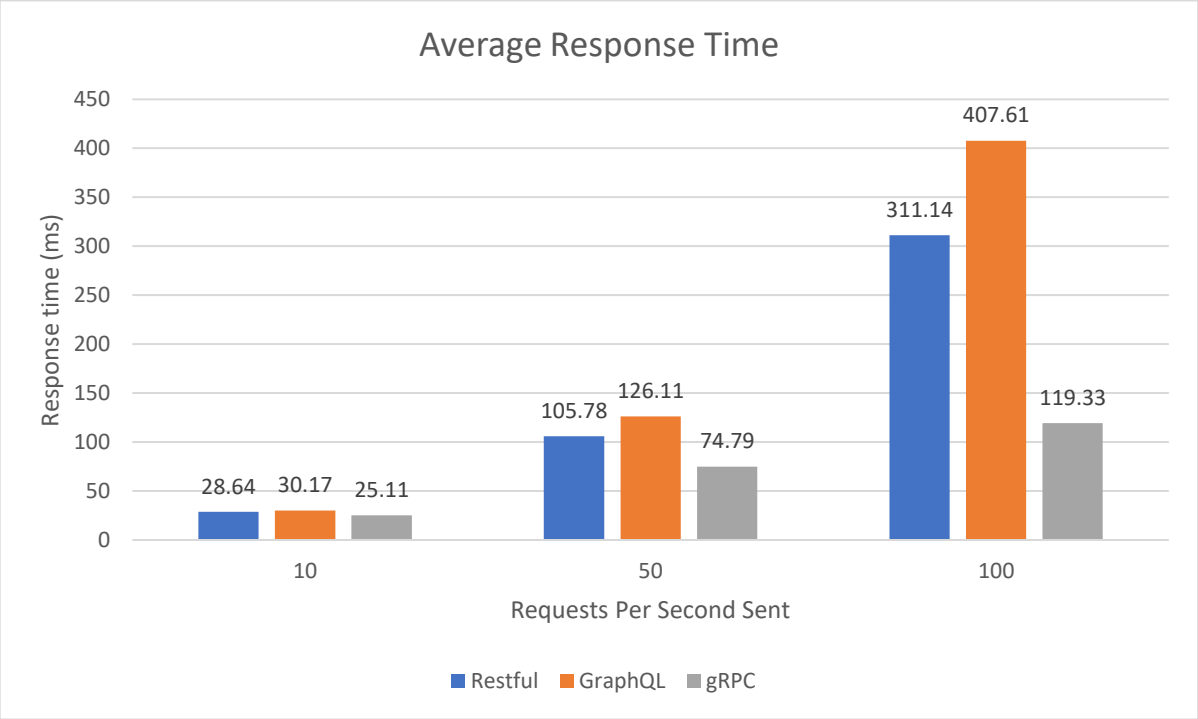

Рисунок 4.3 – Середній час відповіді для першого сценарію.

Для збору метрик було використано NBomber, який дозволяє відправляти запити на Backend For Frontend (BFF) та налаштовувати кількість запитів у секунду. В ході тестування було налаштовано відправку 10, 50 та 100 запитів у секунду протягом 5 хвилин. На рисунку 4.4 зображено пропускну здатність серверних застосунків для першого тестового сценарію.

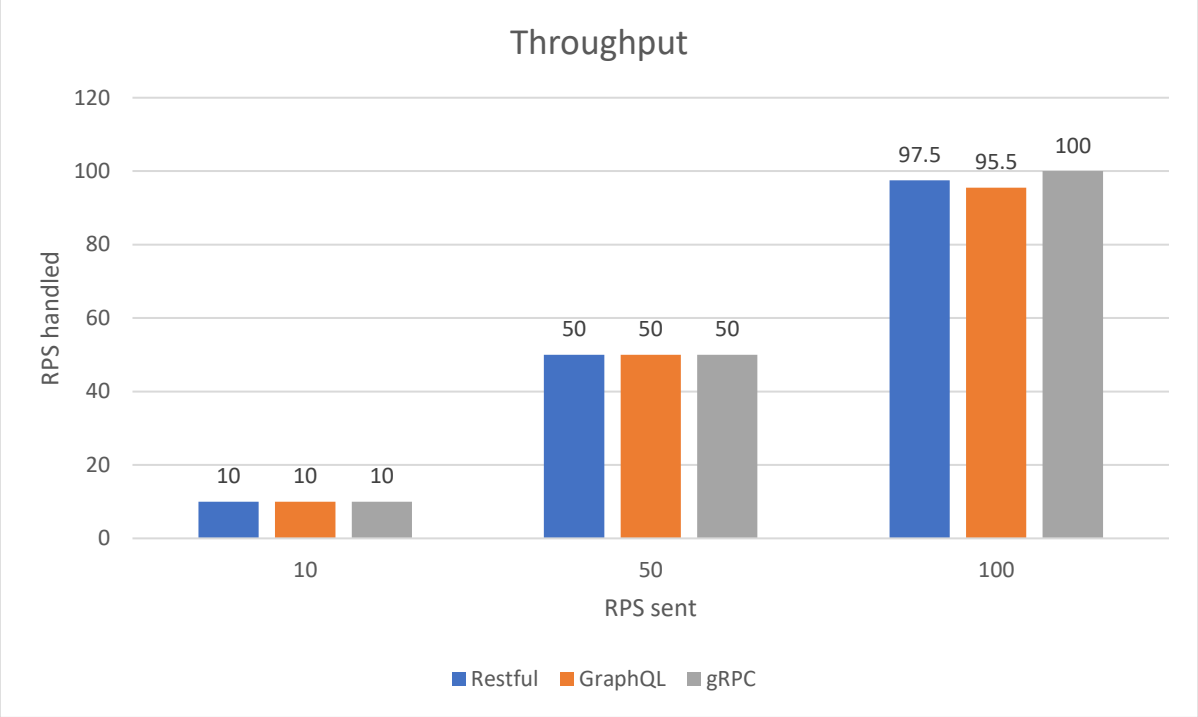

Рисунок 4.4 – Пропускна здатність для першого сценарію.

Останньою метрикою для першого сценарію залишається кількість переданих даних. Data Transfer Size (DTS) - це об’єм даних, який передається між двома точками на мережі. Вимірювання DTS є важливою частиною будь-якого дослідження продуктивності мережі та систем, що підтримують мережеву взаємодію.

Вимірювання DTS дозволяє оцінити продуктивність системи під різними навантаженнями. Наприклад, при великому DTS може бути виявлено, що система працює добре при невеликому навантаженні, але не може витримати велику кількість запитів.

Результати вимірювання кількості переданих даних зображено на рисунку 4.5.

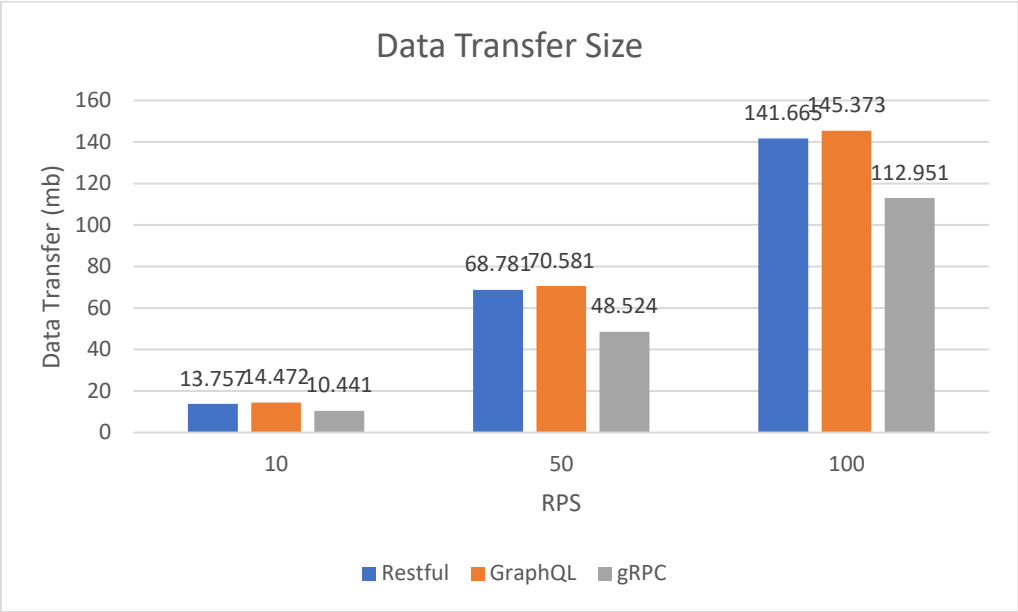

Рисунок 4.5 – Розмір переданих даних для першого тестового сценарію.

Щоб оцінити продуктивність трьох кінцевих точок API у сценаріях часткового отримання об’єкту, ми провели другий набір тестів. Рисунок 4.6 демонструє середній час відповіді у цьому сценарії.

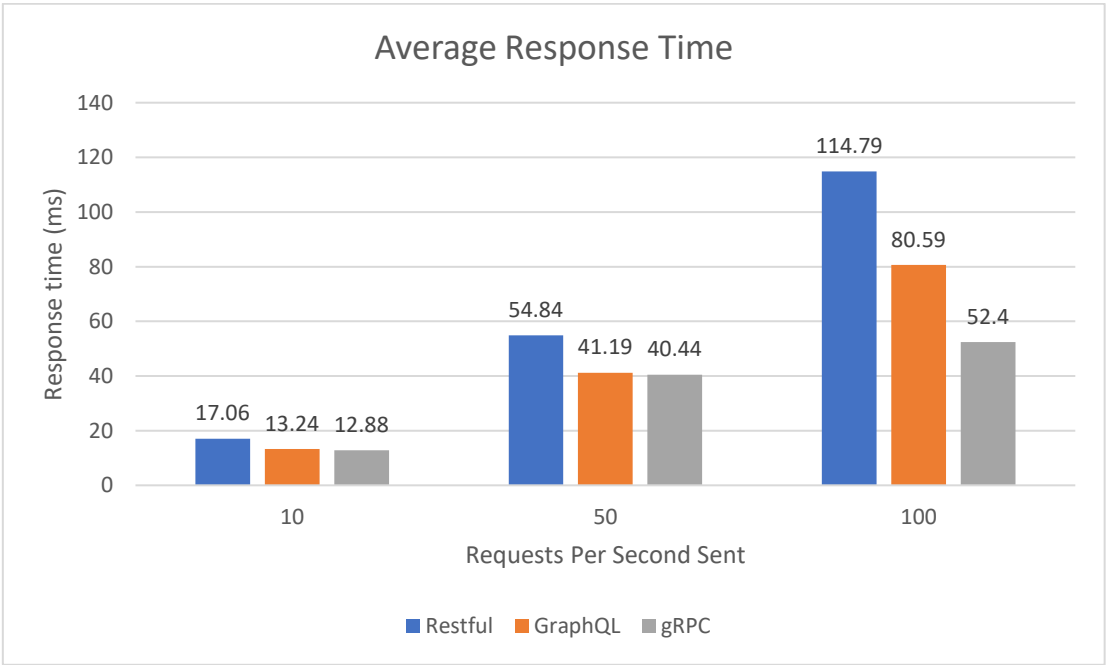

Рисунок 4.6 - Середній час відповіді у другому тестовому сценарії.

Налаштування тестування було ідентичним до першого сценарію, за винятком кінцевих точок для API Restful і служби для API gRPC, які були змінені на отримати часткові дані.

Під час дослідження продуктивності системи або додатку, важливо вимірювати Throughput, оскільки це дає інформацію про кількість запитів, які можуть бути оброблені за певний проміжок часу. Це дозволяє оцінити ефективність системи при роботі з різними об’ємами робочих навантажень.

Вимірювання Throughput є важливим, коли необхідно підвищити продуктивність системи або забезпечити її стабільну роботу при збільшенні кількості запитів. Рисунок 4.7 містить дані про пропуску здатність у другому тестовому сценарії.

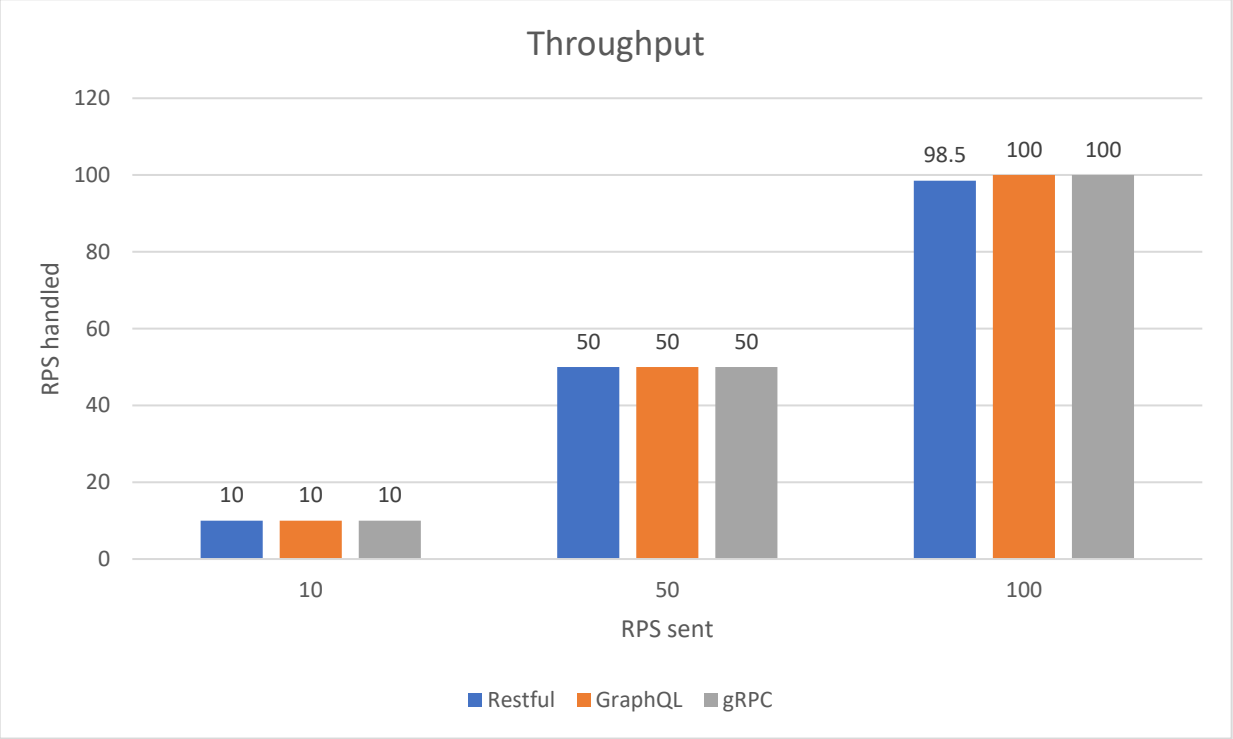

Рисунок 4.7 – Пропускна здатність у другому тестовому сценарії.

Окрім того, важливо визначити, які додаткові чинники можуть впливати на продуктивність системи, такі як обсяг даних, що передається, розмір запитів та кількість паралельних запитів. Отже, виміряємо останню метрику для другого

тестового сценарію, щоб оцінити кількість переданих даних. Результат цього вимірювання наведено на рисунку 4.8.

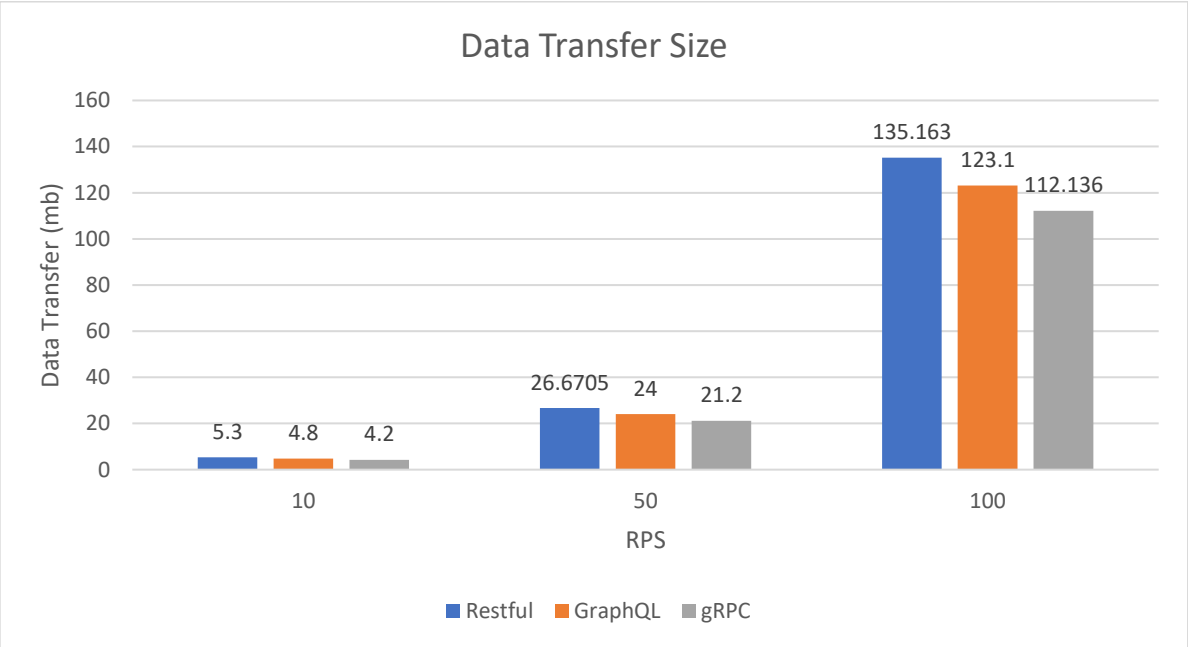

Рисунок 4.8 – Розмір переданих даних для другого тестового сценарію

Таким чином, було оцінено, як працюють API під час отримання лише підмножини доступних даних, що є типовим випадком використання в реальних програмах.

4.7 Аналіз результатів експерименту

У дослідженні було порівняно продуктивність трьох популярних технологій для реалізації API: gRPC, Restful і GraphQL. На основі зібраних метрик було побудовано діаграми, які були проаналізовані, для того, щоб прийти висновку, яка технологія API в яких сценаріях працює найкраще. Нижче наведено детальний аналіз зібраних у минулому пункті даних для порівняння продуктивності трьох API.

У першому сценарії тестування було виміряно середній час відповіді для кожного API. На рисунку 4.3 видно, що gRPC перевершує по ефективності Restful і GraphQL, особливо за умов збільшення кількості запитів на секунду. Час відповіді для gRPC залишався стабільним навіть при високих рівнях навантаження, тоді як у Restful і GraphQL виникали складності та середній час відповіді збільшився. Restful і GraphQL мали схожий середній час обробки запиту в цьому сценарії, при цьому gRPC продемонстрував найнижчий загальний час відгуку. Однак результати також показали, що GraphQL працює трохи гірше, ніж Restful, у сценарії де випробується масштабованість.

На рисунку 4.4 наведено, що всі три API достатньо добре впоралися з високими рівнями завантаження, причому кожен API продемонстрував лише незначне зниження пропускної здатності при збільшенні кількості запитів на секунду. Однак gRPC зміг обробити найбільшу кількість запитів на секунду, досягаючи 100 PRS, тоді як Restful зміг обробити лише 97,5% запитів, а GraphQL — 95,5%. Таким чином, можемо дійти висновку, що gRPC є найефективнішою технологією API з точки зору пропускної здатності, а підходи Restful і GraphQL хоч і не сильно, але відстають у цьому параметрі.

Далі було виміряно розмір переданих даних для першого сценарію, а результати представлені на рисунку 4.5. На графіку показано, що gRPC передає менше даних через мережу у порівнянні з Restful і GraphQL. Функція компресії gRPC зменшує кількість даних, які потрібно передати. При цьому, Restful і GraphQL мають схожі результати щодо обсягу передачі даних. Таким чином, можна зробити висновок, що gRPC є більш ефективним, ніж Restful і GraphQL з точки зору кількості переданих по мережі даних, що напряму впливає на ефективність користування мережею та загальну ефективність системи.

Переходячи до аналізу результатів другого тестового сценарію, я хотів би зазначити, що мені не потрібно було вносити жодних змін у функціональність GraphQL API, щоб забезпечити нову функцію з частковим отриманням даних, що я вважаю плюсом.

Другий тестовий сценарій зосереджувався на отриманні часткових даних сутності, що є частим сценарієм у веб-застосунках. На рисунку 4.6 зображено середній час відповіді для цього сценарію. Результати демонструють, що існує помітна зміна продуктивності GraphQL API у порівнянні з минулим сценарієм, завдяки його функції вибору полів для отримання, що робить його більш вдалим вибором для сценаріїв цього типу. Однак gRPC все ще перевершує GraphQL за середнім часом відповіді. Хоча GraphQL може краще підходити для часткового пошуку об’єктів, він не може перевершити gRPC з точки зору продуктивності. Тому вибір між цими API зрештою залежить від конкретних вимог системи та пріоритетів команди розробників.

На рисунку 4.7 можна побачити, що пропускна здатність API у випадку часткових даних майже однакова: gRPC і GraphQL демонструють чудові результати, а Restful API добре справляється з навантаженням, але з трохи гіршими показниками. На малюнку 4.8 видно, що розмір переданих даних для API GraphQL значно менший за попередній тестовий сценарій, при цьому обсяг даних, що передаються, майже такий самий, як і скомпресовані дані з gRPC API.

Експеримент показав, що gRPC перевершує Restful і GraphQL за середнім часом відповіді, пропускною здатністю та розміром переданих даних, що робить її найефективнішою технологією для побудови міжсерверної взаємодії. У першому тестовому сценарії, де вимірювався середній час відповіді, час відповіді gRPC залишався стабільним навіть при високих рівнях навантажень, тоді як Restful і GraphQL продемонстрували значне збільшення часу відгуку. У першому тестовому сценарії, де вимірювалася пропускна здатність, gRPC міг обробити найбільшу кількість запитів на секунду, тоді як Restful і GraphQL не змогли обробити стільки запитів. Однак, експеримент також показав, що функція GraphQL для часткового отримання полів об’єктів робить її більш придатною для цього конкретного сценарію. Хоча gRPC перевершує GraphQL з точки зору часу відповіді, GraphQL краще підходить для часткового отримання об’єктів, що підходить краще для сценаріїв такого типу.

4.8 Проблеми та рекомендації до використання

Якщо головною проблемою є максимізація продуктивності та пропускної здатності, gRPC буде найкращий варіантом. Його підтримка передачі бінарних даних і ефективне стиснення даних роблять його ідеальним для сценаріїв де вимагається висока пропускна здатність та низький час відповіді. Однак, варто зазначити, що gRPC може потребувати більше досвіду у розробці та знань у галузі налаштування та підтримки інфраструктури, оскільки він передбачає використання бінарних протоколів і вимагає додаткової конфігурації для використання веб-сервісів.

З іншого боку, якщо основною вимогою є легкість розробки та гнучкість, Restful буде кращим вибором. Використання стандартизованих протоколів HTTP та підтримка широким спектром мов програмування та фреймворків роблять його легким і простим у роботі навіть для розробників, які можуть бути не знайомі зі специфікою розробки та реалізації API. Крім того, Restful API мають високу масштабованість і можуть бути легко інтегровані з різними іншими системами та службами.

Для сценаріїв, де важливий часткове отримання даних і гнучкі запити, GraphQL буде кращим варіантом. Його здатність вибірково отримувати лише ті дані, які потрібні клієнту, може значно зменшити обсяг даних, які необхідно передати через мережу, що призведе до підвищення продуктивності та зменшення навантаження на мережу. Легкість впровадження та факт, що не треба робити ніяких змін на серверному боці для реалізації нових способів отримання даних також є відчутним плюсом. Крім того, його декларативна та строго типізована мова схем може спростити процес розробки API та полегшити підтримку та розвиток API у перспективі.

ВИСНОВКИ

В ході виконання магістерської кваліфікаційної роботи було досліджено та проаналізовано сучасні тенденції у способах побудови міжсерверного зв’язку. В результаті аналізу було знайдено проблемні місця існуючих підходів та надані рекомендації щодо використання у реальних застосунках.

В результаті розробки було створено 3 серверних застосунки з різними реалізаціями веб-інтерфейсів, 1 серверний застосунок з реалізацією архітектурного шаблону BFF для налагодження міжсерверного зв’язку та клієнтський застосунок для тестування навантаженням зазначених веб-сервісів.

Для точного та справедливого збору та подальшого аналізу метрик серверні застосунки було розгорнуто як хмарні сервіси Azure. Були написані та програмно реалізовані тестові сценарії. У результаті порівняння зроблено висновки про можливості використання різних підходів для реалізації міжсерверної взаємодії між мікросервісами. Результати роботи опубліковано у науково-технічній конференції «Collins 2023».

Робота дає цінну інформацію про ефективність gRPC, Restful і GraphQL і може допомогти командам розробників вибрати найкращу технологію API для їхніх конкретних вимог. Однак наше дослідження має деякі обмеження. По-перше, було проаналізовано лише три зазначені вище популярні технології побудови API і в аналіз не були включені інші підходи. По-друге, тестові сценарії були обмежені й можуть не повністю відображати всі можливі сценарії використання. Майбутні дослідження можуть зосередитися на тестуванні інших технологій побудови API і розширенні тестових сценаріїв, щоб отримати більш повне розуміння щодо ефективної реалізації міжсерверної взаємодії.

Результати дослідницької роботи задовольняють вимогам винесеним до неї у постановці задачі.

ПЕРЕЛІК ДЖЕРЕЛ ПОСИЛАННЯ

1. Most popular social networks worldwide as of January 2022, ranked by number of monthly active users [Електронний ресурс] – Режим доступу до ресурсу: https://www.statista.com/statistics/272014/global-social-networks-ranked-by-number-of-users (дата звернення: 19.12.2022)

2. Sean Whitesell. Pro Microservices in .NET 6: With Examples Using ASP.NET Core 6. Apress. 2022 – 320 с.

3. Sam Newman. Building Microservices: Designing Fine-Grained Systems. O`Reilly Media. 2015 – 280 с.

4. Rob Ford. Web design. The evolution of the digital world 1990-today. Taschen. 2016 – 640 с.

5. Martin Fowlers Microservices Guide [Електронний ресурс] – Режим доступу до ресурсу: https://martinfowler.com/microservices (дата звернення: 19.12.2022)

6. Number of internet users worldwide from 2005 to 2021 [Електронний ресурс] – Режим доступу до ресурсу: https://www.statista.com/statistics/273018/number-of-internet-users-worldwide (дата звернення: 19.12.2022)

7. Think with Google. Bounce rate statistics. [Електронний ресурс] – Режим доступу до ресурсу: https://www.thinkwithgoogle.com/marketing-strategies/app-and-mobile/page-load-time-statistics (дата звернення: 19.12.2022)

8. Heikki Laaki, Comparison of REST and GraphQL Interfaces for OPC UA, Bachelor`s Thesis, Aalto University, Finland, 2022.

9. Pablo Fernandez, GraphQL or REST for Mobile Applications, Conference Paper, University of Seville, Spain, 2022.

10. Jeremy Wagner, Web Performance in Action: Building Fast Web Pages, Simon and Schuster, NY, 2016, C. 314-316.

11. K. Smelyakov, M. Hvozdiev, A. Chupryna, D. Sandrkin and V. Martovytskyi, "Comparative Efficiency Analysis of Gradational Correction Models of Highly Lighted Image," 2019 IEEE International Scientific-Practical Conference Problems of Infocommunications, Science and Technology (PIC S&T), 2019, pp. 703-708, doi: 10.1109/PICST47496.2019.9061356.

12. K. Smelyakov, A. Chupryna, M. Hvozdiev and D. Sandrkin, "Gradational Correction Models Efficiency Analysis of Low-Light Digital Image," 2019 Open Conference of Electrical, Electronic and Information Sciences (eStream), 2019, pp. 1-6, doi: 10.1109/eStream.2019.8732174.

13. K. Smelyakov, A. Chupryna, D. Sandrkin and M. Kolisnyk, "Search by Image Engine for Big Data Warehouse," 2020 IEEE Open Conference of Electrical, Electronic and Information Sciences (eStream), Vilnius, Lithuania, 2020, pp. 1-4, doi: 10.1109/eStream50540.2020.9108782.

14. I. Ruban, O. Makoveichuk, V. Khudov, I. Khizhnyak, H. Khudov, I. Yuzova, and Y. Drob. The Method for Selecting the Urban Infrastructure Objects Contours, in Intern. Scient.-Pract. Conf. Problems of Infocommunications. Science and Technology (PIC S&T), 2019, pp. 689–693. DOI: https://doi.org/10.1109/infocommst.2018.8632045.

15. Рой Філдінг. Architectural Styles and the Design of Network-Based Software Architectures. Ph.D. Thesis, University of California, Irvine, CA, USA, 2000.

16. Microsoft. Web API Design [Електронний ресурс] – Режим доступу до ресурсу: https://docs.microsoft.com/en-us/azure/architecture/best-practices/api-design (дата звернення: 19.12.2022).

17. Webber, J.; Parastatidis, S.; Robinson, I. REST in Practice: Hypermedia and Systems Architecture. O'Reilly Media. Sebastopol, CA, USA, 2010.

18. GraphQL Foundation. Introduction to GraphQL [Електронний ресурс] – Режим доступу до ресурсу: https://graphql.org/learn/ (дата звернення: 19.12.2022).

19. GraphQL: A Data Query Language [Електронний ресурс] – Режим доступу до ресурсу: https://engineering.fb.com/2015/09/14/core-data/graphql-a-data-query-language/ (дата звернення: 19.12.2022).

20. Глеісон Бріто. REST vs GraphQL: A Controlled Experiment. Conference Paper, International Conference on Software Architecture, 2020.

21. Introducing gRPC, a new open source HTTP/2 RPC Framework [Електронний ресурс] – Режим доступу до ресурсу: https://developers.googleblog.com/2015/02/introducing-grpc-new-open-source-http2.html (дата звернення: 19.12.2022).

22. gRPC FAQ [Електронний ресурс] – Режим доступу до ресурсу: https://grpc.io/docs/what-is-grpc/faq (дата звернення: 19.12.2022).

23. Azure SQL Database [Електронний ресурс] – Режим доступу до ресурсу: https://azure.microsoft.com/en-us/products/azure-sql/database (дата звернення: 25.03.2023).

24. Entity Framework Core [Електронний ресурс] – Режим доступу до ресурсу: https://learn.microsoft.com/en-us/ef/core/ (дата звернення: 25.03.2023).