Хмельницький національний університет
Факультет інформаційних технологій
Кафедра інженерії програмного забезпечення

КВАЛІФІКАЦІЙНА РОБОТА

Методи реалізації мікросервісних архітектур при розробці програмних застосунків: впровадження та тестування

Назва теми

Рівень вищої освіти Другий (магістерський)

Галузь знань 12 «Інформаційні технології»

Спеціальність 121 «Інженерія програмного забезпечення»

Освітня програма Освітньо-професійна програма «Інженерія програмного забезпечення»

Шифр КвРІПЗ.2301111.01.04.ПЗ

Виконав студент 2 курсу, група ІПЗм-23-1 _____ С. А. Грига
Підпис Ім'я, ПРІЗВИЩЕ

Керівник канд. пед. наук, доцент _____ Н. І. Праворська
Науковий ступінь, звання Підпис Ім'я, ПРІЗВИЩЕ

Нормоконтролер канд. тех. наук, доцент _____ О. М. Яшина
Підпис Ім'я, ПРІЗВИЩЕ

До захисту допускаю:
Завідувач кафедри
інженерії програмного забезпечення _____ Л. П. Бедратюк
Підпис Ім'я, ПРІЗВИЩЕ

Дата 28.11.2024

Хмельницький 2024

ХМЕЛЬНИЦЬКИЙ НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ

Факультет Інформаційних технологій
Кафедра Інженерії програмного забезпечення
Рівень вищої освіти Другий (магістерський)
Галузь знань 12 «Інформаційні технології»
Спеціальність 121 «Інженерія програмного забезпечення»
Освітня програма Освітньо-професійна програма «Інженерія програмного забезпечення»

ЗАТВЕРДЖУЮ
Завідувач кафедри ІПЗ
Л. П. Бедратюк
02.09.2024 р.

ЗАВДАННЯ
НА КВАЛІФІКАЦІЙНУ РОБОТУ

Григи Сергія Андрійовича
Прізвище, ім'я, по батькові здобувача

1. Тема роботи Методи реалізації мікросервісних архітектур при розробці програмного забезпечення: впровадження та тестування

Керівник роботи канд. пед. наук, доцент Праворська Н. І.
Прізвище, ім'я, по батькові, науковий ступінь, вчене звання

Затверджена наказом ректора університету від 26.08.2024 р. № 60
2. Строк подання студентом роботи на кафедру 02.12.2024 р.
3. Вихідні дані до роботи Матеріали науково-дослідної практики
4. Зміст пояснювальної записки (перелік питань, які потрібно розробити)
1. Дослідження предметної області та постановка задачі
2. Підходи та методи вирішення поставлених задач
3. Проектування програмного забезпечення
4. Реалізація та тестування програмного забезпечення
5. Перелік графічного матеріалу (із зазначенням обов'язкових креслень)
Презентаційні матеріали (слайди)

6. Консультанти розділів кваліфікаційної роботи

Розділ	Прізвище, ініціали та посада консультанта	Підпис, дата	
		завдання видав	завдання прийняв
Антиплагіат	Форкун Ю. В., к.т.н, доцент		
Нормоконтроль	Яшина О. М., к.т.н, доцент	23.11.24	23.11.24

7. Дата видачі завдання « 02 » вересня 2024 р.

КАЛЕНДАРНИЙ ПЛАН

Назва етапів (розділів) кваліфікаційної роботи	Строк виконання етапів роботи	Примітка
1 Вивчення предметної області, формування мети та задач дослідження, визначення об'єкта та предмета дослідження	02.09-10.09.2024	
2 Робота над розділом 1 – вивчення джерел, аналіз сучасних підходів, методів і засобів за темою роботи, визначення задач, висновки	11.09-25.09.2024	
3 Робота над розділом 2 – розробка підходів дослідження поставленої задачі, висновки	26.09-10.10.2024	
4 Робота над науковими статтями	11.10-30.10.2024	
5 Робота над розділом 3 – аналіз вимог до програмного застосунку, розробка структури та вибір технологій, висновки	11.10-26.10.2024	
6 Робота над етапом 4 – реалізація та тестування програмного забезпечення	27.10-17.11.2024	
7 Попередній захист дипломної роботи	Листопад (згідно графіку)	
8 Узгодження постановки задач, отриманих результатів та висновків, написання вступу, загальних висновків, оформлення джерел та додатків, оформлення записки та графічних матеріалів відповідно до вимог	18.11-30.11.2024	
9 Перевірка роботи на наявність плагіату, нормоконтроль, брошурування пояснювальної записки, підготовка супровідних документів	01.12-04.12.2024	
10 Підготовка до захисту кваліфікаційної роботи	з 01.12.2024	

Студент _____ Грига. С. А.
Підпис Ім'я, ПРІЗВИЩЕ

Керівник роботи _____ Праворська. Н. І.
Підпис Ім'я, ПРІЗВИЩЕ

АНОТАЦІЯ

Тема кваліфікаційної роботи: Методи реалізації мікросервісних архітектур при розробці програмних застосунків: впровадження та тестування.

Автор роботи: Грига Сергій Андрійович.

Керівник роботи: Праворська Наталія Іванівна.

Пояснювальна записка 94 с., 33 рис., 5 табл., 3 дод., 31 джерел.

Об'єктом даного дослідження є мікросервісна архітектура, її особливості реалізації та способи організації міжсервісної комунікації.

Предметом дослідження даної кваліфікаційної роботи є методи передачі даних між мікросервісами, їх ефективність, інтеграція та вплив на продуктивність.

Вперше було запропоновано підхід до одночасного використання синхронних та асинхронних методів комунікації між мікросервісами, що забезпечує підвищення ефективності роботи програмного забезпечення при обробці запитів із різним об'ємом даних.

Головною метою даної роботи є дослідження, аналіз та реалізація мікросервісної архітектури з використанням різних методів міжсервісної комунікації для визначення найбільш ефективних сценаріїв їх використання залежно від обсягів даних та інших параметрів.

Для досягнення поставленого завдання було проведено детальний аналіз предметної області та проаналізовано сучасні дослідження. На основі отриманої інформації було визначено технології та методи, розроблено програмне забезпечення з допомогою якого було проведено тестування.

Отримані результати дослідження демонструють, що найбільш ефективними методами комунікації виявились ті комбінації, що використовують асинхронний обмін даними, що пов'язано з особливостями їхньої роботи та реалізації.

28.11.2024
Дата

Підпис

ЗМІСТ

ВСТУП

На сьогоднішній день розробники програмного забезпечення активно використовують широкий спектр різноманітних архітектур. Їх коректний вибір та реалізація є одним із найважливіших етапів розробки, що у подальшому буде впливати на всі етапи життєвого циклу та ефективність роботи кінцевого програмного продукту.

У зв’язку з цим надзвичайно важливим є детальний аналіз технічного завдання, на основі якого здійснюється вибір архітектури з врахуванням її переваг та недоліків. Наприклад, монолітну архітектуру доцільно використовувати для невеликих за розміром проектів, де критичним аспектом є швидкість розробки. Чи клієнт-серверна архітектура, що використовується для створення веб-застосунків, де функціонал можна чітко розмежувати на дві частини.

Дані архітектури вже давно використовуються розробниками програмного забезпечення і за багато років їх активного використання є відточеними майже до досконалості. Але, і досі деякі архітектури є важкими для впровадження з різних причин, а їх некоректна реалізація може призвести до критичних наслідків, що в свою чергу можуть спричинити повну неможливості використання розробленого програмного продукту.

До таких типів можна віднести мікросервісну архітектуру. Її концепція існує відносно давно, проте вона почала набувати популярність нещодавно завдяки своїм перевагам, що змогли вдало використати великі компанії у своєму програмному забезпеченні, до яких можна віднести Amazon, Uber, Netflix та Etsy.

Але разом з цим, дана архітектура має ряд серйозних недоліків, наприклад, висока складність розробки та менша швидкість виконання коду, що може призвести до зменшення ефективності роботи програмного продукту чи навіть сильно сповільнити процес розробки, що потягне за собою ряд інших проблем включно з повним провалом проекту.

Одним із найкритичніших аспектів розробки мікросервісної архітектури є підхід до комунікації мікросервісів. На ефективність обміну даними між сервісами може впливати багато факторів, до яких можна віднести вірність розбиття системи на мікросервіси, підходи до реалізації каналів зв’язку між ними, їхня кількість та довжина ланцюга мікросервісів, що беруть участь у виконанні запитів, та навіть вибраних спосіб розгортання системи.

На сьогоднішній день вичерпних матеріалів даної тематики дуже мало, а більшість з них обмежуються використанням певного одного підходу в контексті вирішення однієї конкретної задачі.

З даних причин доцільним є проведення дослідження сучасних методів реалізації комунікацій в мікросервісній архітектурі, що напряму впливають на ефективність роботи програмного продукту та обміну даними між сервісами, з врахуванням сучасних підходів до розгортання таких систем.

Об’єктом даного дослідження є мікросервісна архітектура, її особливості реалізації та способи організації міжсервісної комунікації.

Предметом дослідження даної кваліфікаційної роботи є методи передачі даних між мікросервісами, їх ефективність, інтеграція та вплив на продуктивність.

Головною метою виконання даної роботи було детальне дослідження методів реалізації комунікацій у мікросервісній архітектурі та ефективності їхнього спільного застосування.

Наукова новизна полягає у тому, що вперше було запропоновано підхід до одночасного використання синхронних та асинхронних методів комунікації між мікросервісами, що забезпечує підвищення ефективності роботи програмного забезпечення при обробці запитів із різним об’ємом даних.

Для досягнення поставлених цілей роботи було вирішено наступні завдання:

– здійснено аналіз предметної області

– проаналізовано ряд сучасних досліджень у сфері мікросервісної архітектури;

– визначено сучасні методи комунікації мікросервісів між собою та підходи їхнього розгортання;

– сформовано методи дослідження;

– сформовано структуру програмного забезпечення, на основі якого було розроблено застосунок;

– визначено ряд технологій та інструментів, що були використані для розробки програмного забезпечення та його тестування;

– проведено ряд тестів для створеного програмного забезпечення на основі яких здійснено аналіз результатів та зроблено висновки щодо ефективності розроблених методів комунікації між мікросервісами і можливостей їхнього подальшого застосування.

1 ХАРАКТЕРИСТИКА ПРЕДМЕТНОЇ ОБЛАСТІ ТА ПОСТАНОВКА ЗАДАЧІ

1.1 Аналіз предметної області

Мікросервісна архітектура – це архітектурний стиль розробки програмного забезпечення в якому єдина система складається із сукупності невеликих та незалежних сервісів, що виконують конкретно визначену бізнес функцію. Дані сервіси взаємодіють між собою за допомогою простих протоколів передачі даних, наприклад, HTTP для виконання більш складних задач.

Перші базові концепції, що призвели до появи мікросервісної архітектури, зородились у 1980 роках. Програмісти намагались розробити децентралізовану архітектуру для розподілу завдань між декількома різними серверами чи сервісами, що призвело до появи сервісно-орієнтованої архітектури.

Сервісно-орієнтована архітектура була популярною у двохтисячних роках і активно використовувалась для розробки розподілених систем. Їх концепція передбачала створення бізнес-сервісів, що могли взаємодіяти один з одним через різноманітні стандартні протоколи, наприклад, SOAP. Але, попри всі наявні переваги, сервісно-орієнтованої архітектури вона мала ряд критичних проблем, до яких відносили високу складність впровадження та управління масштабними системами, велику залежність від централізованих серверів, що призводило до великих витрат на координацію сервісів.

Для вирішення даних проблем було розроблено мікросервісну архітектуру як еволюція сервісно-орієнтованої архітектури. Вона акцентувала увагу на менші та більш автономні сервіси, що можуть незалежно створюватись, розгортатись та модифікуватись.

Мікросервісна архітектура почала набирати популярність у 2010 роках завдяки її використанню великими компаніями, наприклад, Netflix, Amazon, Google та інших, розробники почали проводити власні дослідження та розробку інструментарію, що і до сьогодні продовжує розвиватись.

Приклад спрощеної схеми сучасної мікросервісної архітектури зображено на рисунку 1.1.

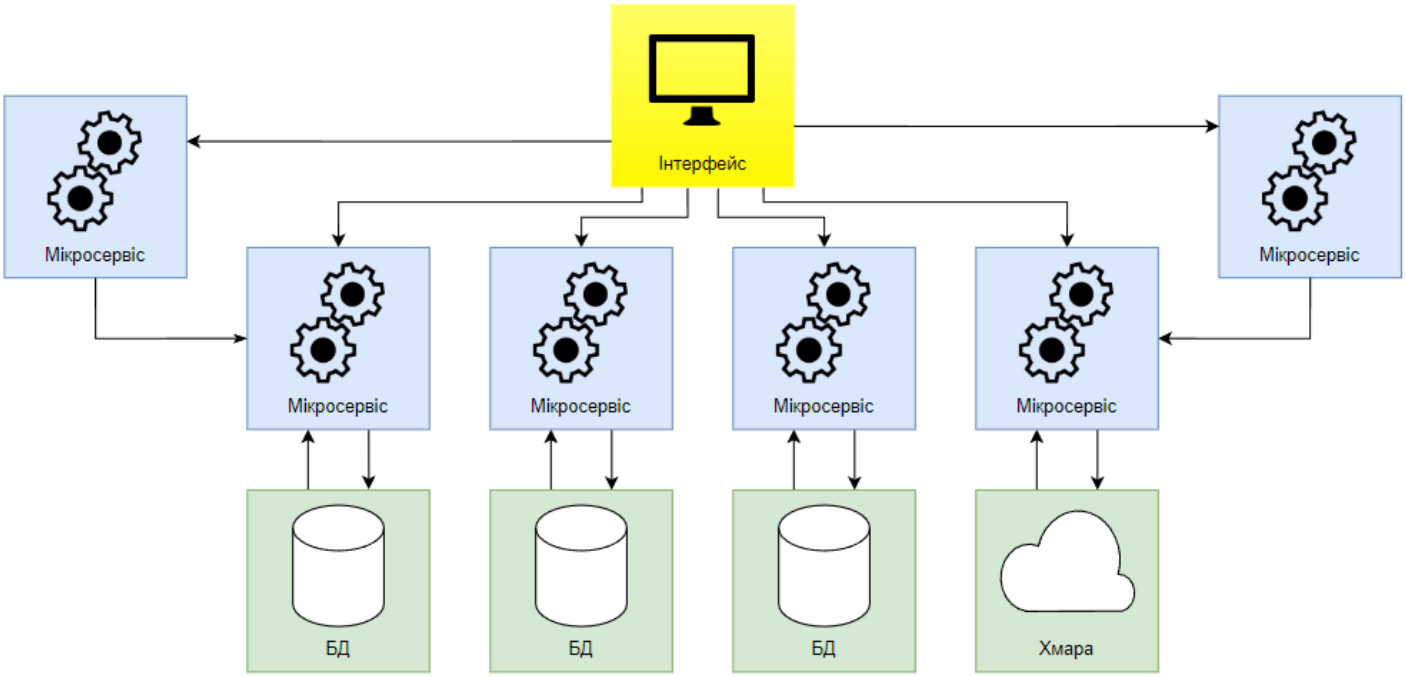

Рисунок 1.1 – Схема мікросервісної архітектури

З даного рисунка можна зрозуміти, що кожен мікросервіс, що безпосередньо взаємодіє з певними даними, має власну базу даних, що можуть бути розміщені як локальними, так і у хмарі. За потреби один сервіс може отримувати дані, що відносяться до іншого сервісу шляхом прямої передачі інформації чи виклику методів одного мікросервісу в іншому.

Наприклад, в інтернет-магазині один сервіс відповідає за обробку даних клієнтів та зберігає їх у відповідну базу даних, другий сервіс працює з даними про різноманітні товари магазину, третій відповідає за обробку покупок при цьому він може використовувати дані як про клієнта, так і про товари, що він хоче придбати, отримавши їх від інших сервісів.

До головних переваг мікросервісної архітектури відносять:

– можливість незалежного масштабування, розгортання та оновлення кожного мікросервісу;

– команди, що розробляють сервіси, можуть працювати незалежно один від одного та можуть використовувати різний набір технологій, мови програмування, тощо;

– можливість швидкої розробки за допомогою залучення великої кількості команд розробників;

– можливість повторного використання коду;

– підвищена стійкість системи до збоїв та кращий загальний рівень надійності;

– високий рівень ізоляції неполадок, що при збої роботи одного сервісу дозволяє продовжувати працювати іншим елементам систем;

– чітке розмежування відповідальності кожного мікросервісу відповідно до наявних бізнес-функцій;

– ефективне використання ресурсів.

На перший погляд можна сказати, що ці переваги є серйозними, але дана архітектура також має ряд серйозних недоліків, що можуть бути набагато критичнішими і повністю перекрити всі переваги її застосування. До основних недоліків можна віднести:

– висока складністю управління та підтримки великої кількісті сервісів, що потребує використання додаткових інструментів;

– є повільнішою у порівнянні з багатьма іншими архітектурами через необхідність постійної взаємодії різних мікросервісів між собою та передачі даних між ними;

– необхідність забезпечення узгодженості даних між різними мікросервісами, що може бути складним завданням у випадках використання розподілених транзакцій;

– потребує використання ряду потужних інструментів для управління контейнерами, логування, моніторингу та забезпечення безпеки всієї системи, що може збільшити витрати на розробку та підтримку;

– розробники повинні мати високий рівень знань та навичок у різних сферах;

– висока вартість розробки.

Особливу увагу в контексті мікросервісної архітектури необхідно акцентувати на доцільність її використання. Деякі сучасні команди розробників та замовники акцентують увагу на використанні саме мікросервісної архітектури через її відносну новизну та ефективність використання, орієнтуючись на приклади інших раніше розроблених застосунків, що є критично не вірним підходом.

Мікросервісна архітектура є досить специфічною та має ряд критичних недоліків у зв'язку з чим її застосування сильно обмежується досить невеликим переліком можливих програмних продуктів. З даних причин її доцільно використовувати лише у випадках, коли майбутня система відповідає хоча б одній із наступних вимог:

– коли команда розробників, що розроблює проект, використовують лише мови програмування з динамічною типізацією, а сам проект є досить масштабним за своїми розмірами;

– коли команда розробників з тих чи інших причин використовує декілька різних мов програмування для реалізації проекту;

– коли до розробки проекту залучено декілька різних команд розробників, що не можуть активно взаємодіяти між собою;

– коли проект є об'ємним за своїм розміром та існують плани на його майбутній розвиток, що потребуватиме незалежного масштабування його окремих компонентів.

Якщо жоден із наведених пунктів не відноситься до майбутнього програмного забезпечення, то використання мікросервісної архітектури не є доцільним і кращим варіантом буде розгляд інших архітектур. В іншому випадку її використання може призвести до критичних проблем на різних етапах розробки системи чи навіть спричинити провал проекту.

Також, в контексті мікросервісної архітектури, надзвичайно важливим постає завдання на етапі проектування. Під час нього проводиться створення сервісів та можливих шляхів їхньої комунікації на основі розподілу між ними всієї наявної бізнес-логіки майбутнього програмного продукту.

Під час виконання даного завдання важливо коректно визначити розмір майбутніх сервісів. Занадто малі сервіси призводять до того, що структура системи стає занадто складною та заплутаною, а швидкість передачі інформації зменшується, наприклад, для кожної сторінки сайту існує окремий мікросервіс. Дана проблема вирішується об’єднанням декількох малих мікросервісів в один з врахуванням їхнього функціоналу.

Іншим можливим варіантом є занадто великі сервіси. Це є надзвичайно негативним рішенням, адже це порушує суть мікросервінної архітектури, що передбачає розбиття програмного продукту на невеликі за розміром та незалежні елементи, а також може призводити до надмірного навантаження одного мікросервісу. Поява таких великих мікросервісів вказує на допущену помилку при проектування, а тому доцільним є проведення аналізу отриманих результатів та розбиття відповідного сервісу на декілька менших.

Для вирішення ряду проблем пов’язаних з проектуванням мікросервісів, а також, з метою спрощення проектування, розробники програмного забезпечення часто практикують використання різних підходів розбиття системи на сервіси. Нижче наведено п’ять основних сучасних підходів:

- принцип дизайну, орієнтованого на домени;
- декомпозиція за бізнес можливостями;
- декомпозиція за потоками даних ;
- декомпозиція за командними структурами;
- патерн Strangler.

Одним із основних та найбільш популярних сучасних підходів для проектування мікросервісів є дизайн, орієнтований на домени чи DDD.

Дизайн, орієнтований на домени – це підхід до проектування програмного забезпечення, що акцентує увагу на моделі предметної області та тісній взаємодії з працівниками та експертами предметної області, що дозволяє забезпечити краще розуміння бізнес-вимог. Основною метою використання даного підходу є створення системи, що максимально можливо відповідає реальним бізнес-процесам та здатна адаптуватись під їхні зміни.

Використання даного підходу дозволяє забезпечити високий рівень деталізації структури програмного продукту, що допомагає зменшити кількість випадків появи занадто великих чи малих сервісів, а також дозволяє визначити канали комунікації.

До основних принципів дизайну орієнтованого на домени можна віднести:

– детальне моделювання предметної області, що точно відображає наявні бізнес-процеси;

– використання єдиної термінології, що є зрозумілою для розробників програмного забезпеченням те експертам галузі;

– відокремлення різних частин системи на окремі елементи;

– система повинна розроблятись з акцентом на готовність до будь-яких змін, але при цьому працювати стабільно.

Даний підхід структурує всю систему навколо доменів та субдоментів, що дозволяє полегшити розуміння і впровадження складних вимог. Домен являє собою певну область діяльності, що має власні завдання та правила. Наприклад, для поштового сервісу доменом є «доставка посилок», а в банківського застосунку доменами можуть виступати «кредитування» та «управління платежами».

Кожен домен складається із субдоменів, що є меншими за розмірами доменами, але описані більш детально та відповідають за виконання конкретних завдань чи бізнес-аспектів, наприклад, домен «доставки посилок» може складатись з субдоменів для оформлення посилки, відслідковування її місця перебування, побудова маршруту доставки, тощо.

Всі субдомени можуть бути розбиті на наступні елементи:

– основні субдомени – це найбільш важливі для бізнес-логіки домени, що відповідають за виконання критично важливих для бізнесу функцій, які становлять основну цінність для клієнтів та потребують максимальної уваги при проектуванні;

– допоміжні субдомени – це домени, що вміщують у собі набір різноманітних функцій, що призначені для підтримки коректної роботи основних субдоменів, але при цьому не є критично важливими;

– загальні субдомени – це домени, що зберігають в собі набір часто використовуваних функцій іншими елементами системи, що не є унікальними.

На основі всіх наведених доменів та субдоменів може відбуватись створення мікросервісів з орієнтуванням на конкретну поставлену задачу.

Приклад використання можна детально продемонструвати на основі системи для компанії, що займається перевезенням вантажів. Головною задачею такої системи є доставка вантажів, що є одним із основних доменів. До них також можна віднести функціонал, що відповідає за формування документації, функціонал для клієнтів компанії та менеджер, що використовується при формуванні способу доставки для кожного окремого замовлення.

Також даний підхід дозволяє ще на етапі проектування включати функціонал, що буде реалізовуватись у подальших версіях програмного забезпечення. В даному випадку такими можливостями є оренда техніки, коли та не використовується, та можливість контролювати клієнтам етапи доставки. На основі наявної інформації було створено відповідно модель, що зображена на рисунку 1.2.

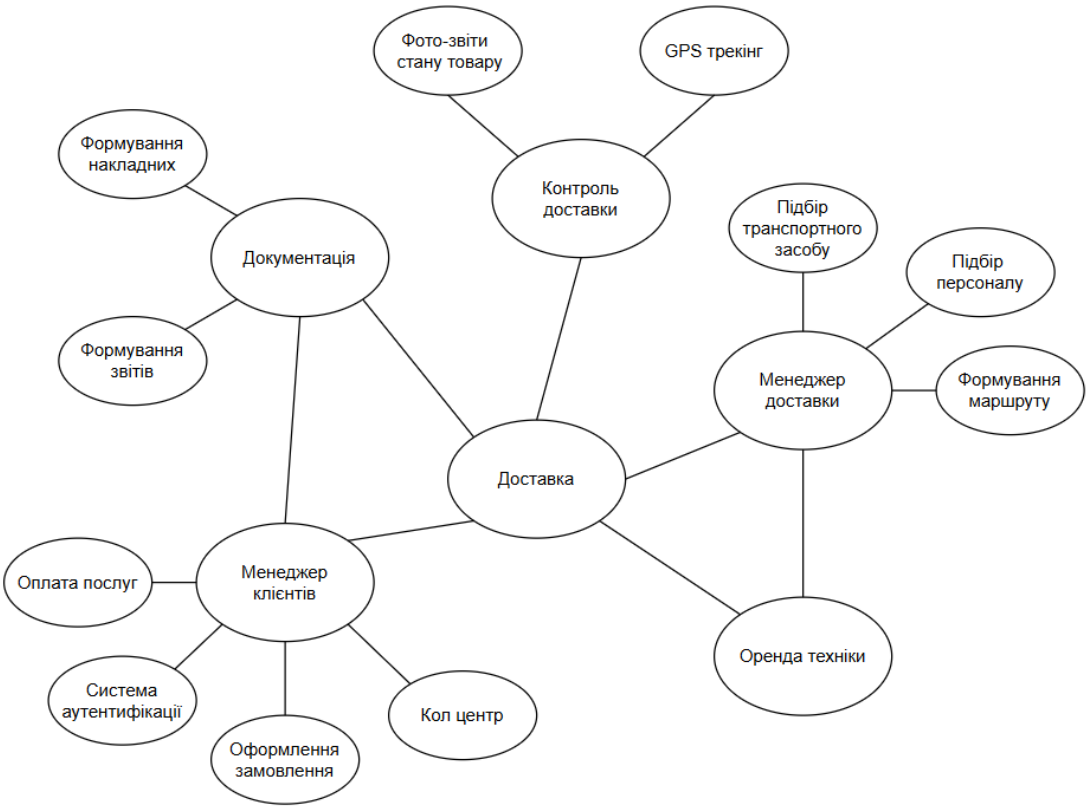

Рисунок 1.2 – Модель розбиття системи для вантажних перевезень на домени

Важливо розуміти, що лише одне розбиття всієї системи на домени та субдомени не є ідеальним підходом. Це пов’язано з тим, що використання одних лише доменів не дозволяє встановити чітких меж відповідальності.

Для вирішення даного недоліку в дизайні, орієнтованому на доменів використовується концепція граничних контекстів.

Граничний контекст – це чітко визначена логічна межа, що встановлює область відповідальності для кожної моделі в межах системи. Іншими словами це ізольована частина системи, що має власні правила і обмеження. В межах граничного контексту використовується власна термінологія, що служить інструментом для опису бізнес-логіки, а терміни мають чітке і незмінне значення, але в інших контекстах ідентичні терміни можуть мати інше визначення.

У випадках використання дизайну орієнтованого на домени визначення граничних контекстів відбувається на основі встановлення функціональних відмінностей кожного домену. Визначені таким чином граничні контексти використовуються як основа для побудови мікросервісів, але з врахуванням можливих випадків їхньої взаємодії.

Якщо ж розглядати весь процес визначення граничних контекстів, то коротко його можна описати наступним чином – спочатку проводиться детальний аналіз предметної області та визначаються ключові елементи бізнес-логіки, що можуть бути відокремлені. Для таких елементів встановлюється як вони можуть взаємодіяти між собою, що допомагає визначити можливі шляхи комунікації між сервісами та уникнути дублювання коду. Далі виконується визначення чітких меж кожного граничного контексту, що орієнтуються на ізольовані моделі та їхні відмінності.

Даний підхід можна застосувати на минулого прикладу. У його випадку розбиття на граничні контексти відбувається на основі відмінностей між завданнями, а саме їхній основній цілі та призначенням. Отриманий результат можна побачити на рисунку 1.3.

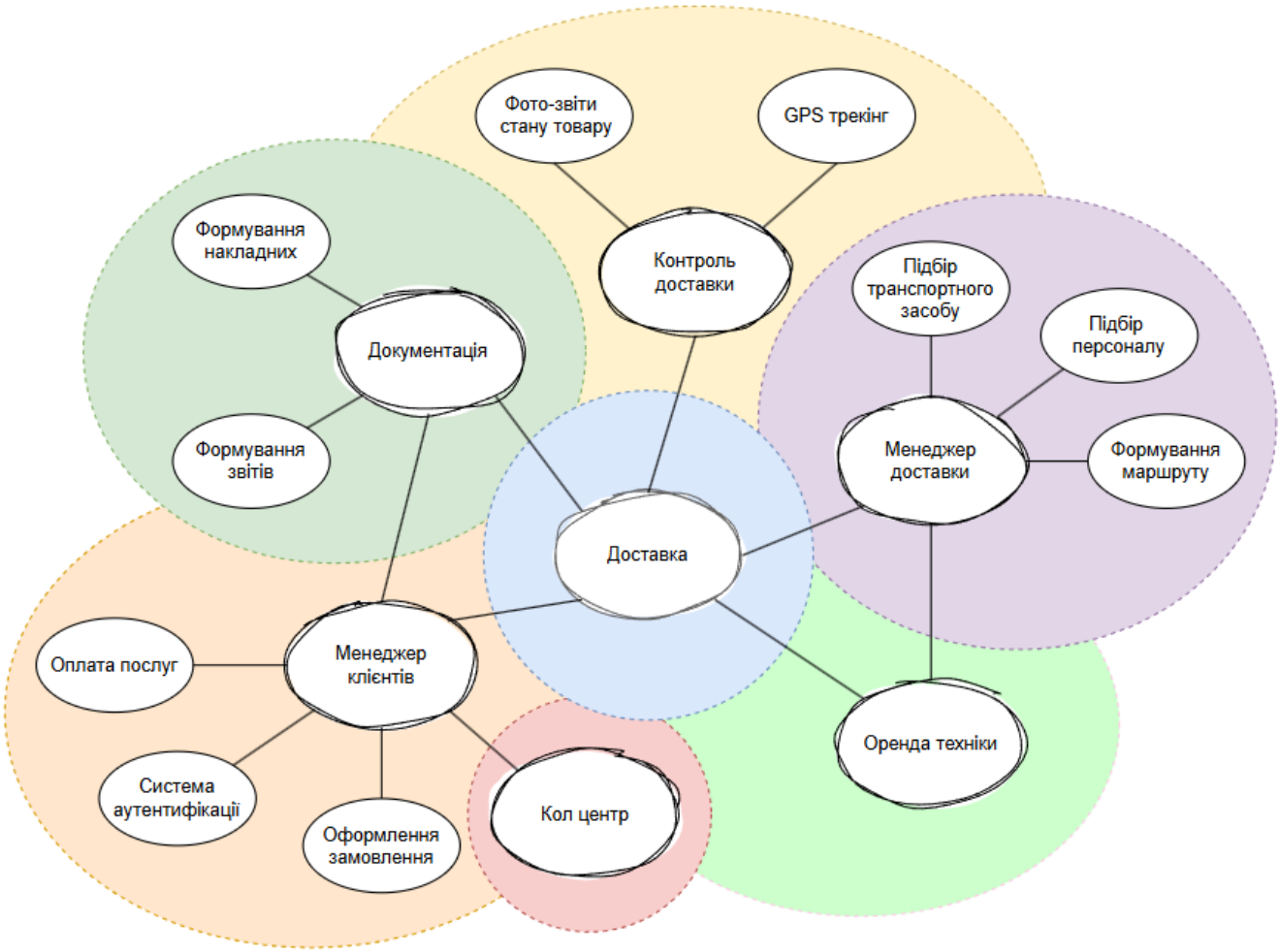

Рисунок 1.3 – Визначення граничних контекстів для наявних доменів

На рисунку видно, що поточна система складається з семи граничних контекстів. Кожен з них відповідає за виконання певних задач, що відрізняються від інших, а тому на їхній основі можуть бути сформовані незалежні мікросервіси. При цьому варто відмітити, що під час визначення граничних контекстів було враховано який функціонал буде розроблений пізніше і чи доцільно його вводити у вже наявні контексти, чи варто виділити в окремий з метою уникнення необхідності проведення недоречної модифікації системи.

Найчастіше дані підходи використовуються при проектуванні складних систем, де важливі точні правила та вимоги, наприклад, фінансові, юридичні чи медичні системи. Також він чудово підходить у випадках, коли необхідно враховувати велику кількість різних аспектів в бізнес-логіці, що вирішується побудовою детальних моделей.

Вище описаний підхід розбиття системи на домени та визначення на їхній основі граничних контекстів при одночасному використанні називають методом стратегічного дизайну.

Подібних методів існує велика кількість. Їхня основна ціль – спросити вирішення певної задачі чи групи задач використовуючи підхід, що довів свою ефективність при розробці інших подібних проектів.

Чудовою альтернативою декомпозиції, орієнтованій на домени є декомпозиція за бізнес можливостями. Як і минулий підхід, він використовується для розбиття системи на мікросервіси орієнтуючись на функціональні можливості. Проте, в даному випадку, він акцентує увагу на виділення модулів, що створюються на основі ключових функціональних можливостей системи.

Проте, на відміну від дизайну орієнтованого на домени, декомпозиція за бізнес можливостями оперує більш широкими категоріями. Він концентрує увагу необхідності охоплення всіх бізнес-процесів для кожного окремого модуля.

Принцип визначення мікросервісів є подібним до визначення доменів у дизайні, орієнтованому на доменах – аналіз предметної області, визначення окремих модулів на основі їхніх функціональних можливостях, та розбиття їх на підмодулі, що відображають менші завдання. Після цього кожен окремий модуль може бути реалізований як окремий мікросервіс.

Даний підхід найкраще зарекомендував себе при розробці програмних продуктів для підприємств, що акцентують увагу на великих бізнес-процесах, що мають підтримувати високий рівень автономності та в компаніях з великою кількістю різноманітних підрозділів.

Варто відмітити, у зв’язку з тим, що даний підхід є менш деталізованим можуть виникати суперечності щодо границь мікросервісів. Способом вирішення даної проблеми може бути використання даного підходу у поєднанні із граничними контекстами, що допоможуть встановити чіткі межі відповідальності.

Наступним методом є декомпозиція за потоками даних, що розбиває систему на мікросервіси на основі потоків даних між різними компонентами. Основною метою підходу є виокремлення мікросервісів на основі взаємодії з даними – тобто

мікросервіси створюються залежно від того, які дані вони отримують, трансформують і передають.

Такий підхід найдоцільніше застосовувати у системах, що працюють з великим обсягом даних, що потребують обробки, трансформації чи аналітики, системах повністю побудованих на асинхронній обробці подій чи повідомлень, а також у випадках, коли необхідна моментальна реакцій на зміни даних.

Однак слід враховувати, що при використанні даного методу виникають питання пов’язані з узгодженням даних та координацією транзакцій для забезпечення цілісності інформації, що є надзвичайно критичним у розподілених системах. Також він може ускладнювати обмін даних, оскільки зростання кількості потоків призводить до збільшення частоти передачі даних між сервісами, що може спричинити затримки та підвищить навантаження на системи.

Теоретично, цей підхід можна ефективно застосовувати у поєднанні з іншими методами декомпозиції, адже його самостійне використання не завжди можливе, а у поєднанні з іншими він може допомогти розділити систему на більш ефективні сервіси, що одночасно можуть працювати над виконанням однієї задачі.

Наступним підходом до визначення мікросервісів є декомпозиція за командними структурами. Даний підхід ґрунтується на тому, що різні команди розробників відповідають за створення, тестування, розгортання та підтримку власного мікросервісу.

Зазвичай, такий підхід до декомпозиції орієнтується на закон Конвеє, що звучить наступним чином: «Будь-яка організація, яка створює систему, (несвідомо) розробляє проекту структуру, що є копією організаційної структури».

Таким чином кожен мікросервіс програмного продукту буде відповідати за виконання задач, що покладені на конкретний відділ компанії, а їхня розробка, зазвичай, здійснюється окремими командами, що концентрують увагу на вирішенні задач відповідного відділу.

Такий підхід дозволяє виконувати одночасну розробку багатьох мікросервісів без необхідності підтримки комунікації між різними командами, що є корисним для великих компаній, але дані переваги призводять до появи ряду серйозних недоліків.

Через низький рівень комунікації між командами розробників та за відсутності чітких правил певні функції можуть дублюватись у різних мікросервісах, а також це може призвести до появи проблем із синхронізацією та узгодженістю даних.

Для вирішення недоліків даного підходу можна провести об’єднання моделей отриманих різними командами та виділити спільні функціональні можливості, що можуть бути винесені в окремі мікросервіси, що будуть спільні для різних команд.

І останнім важливим підходом до визначення мікросервісів є патерн Strangler. На відміну від попередніх підходів, що орієнтовані на розробку нових мікросервісів, даний патерн необхідний для покрокової міграції від монолітної до мікросервісної архітектури для вже створених застосунків.

Для переходу від моноліту до мікросервісів патерн передбачає наявність проміжного етапу, що можна назвати модульним монолітом. Він передбачає створення модулів, що являють собою певну незалежну частину функціоналу готового програмного продукту.

На наступному кроці кожен незалежний модуль поетапно трансформується в окремі мікросервіси. Вибір порядку переносу найчастіше залежить від частоти використання конкретного модуля чи від складності його переносу, де перевага надається простішим модулям. Схематичну демонстрацію такого переносу можна побачити на рисунку 1.3.

При такому підході до створення нового мікросервісу йому передаються всі запити, що до цього виконував відповідний модуль, а сам модуль може бути видалений із системи. По перетворенню всіх моделей на мікросервіси можна стверджувати, що програмний продукт було перенесено із монолітної архітектури на мікросервісну.

Детальний аналіз даних підходів демонструє, що кожен може ефективно використовуватись за певних умов для різних типів програмних продуктів, але варто відмітити, що їх використання може відбуватись одночасно при наявності для цього відповідних факторів. Такі способи поєднання, теоретично, можуть покращити підхід до проектування мікросервісних систем, хоча і не зможуть виступати в ролі універсального методу.

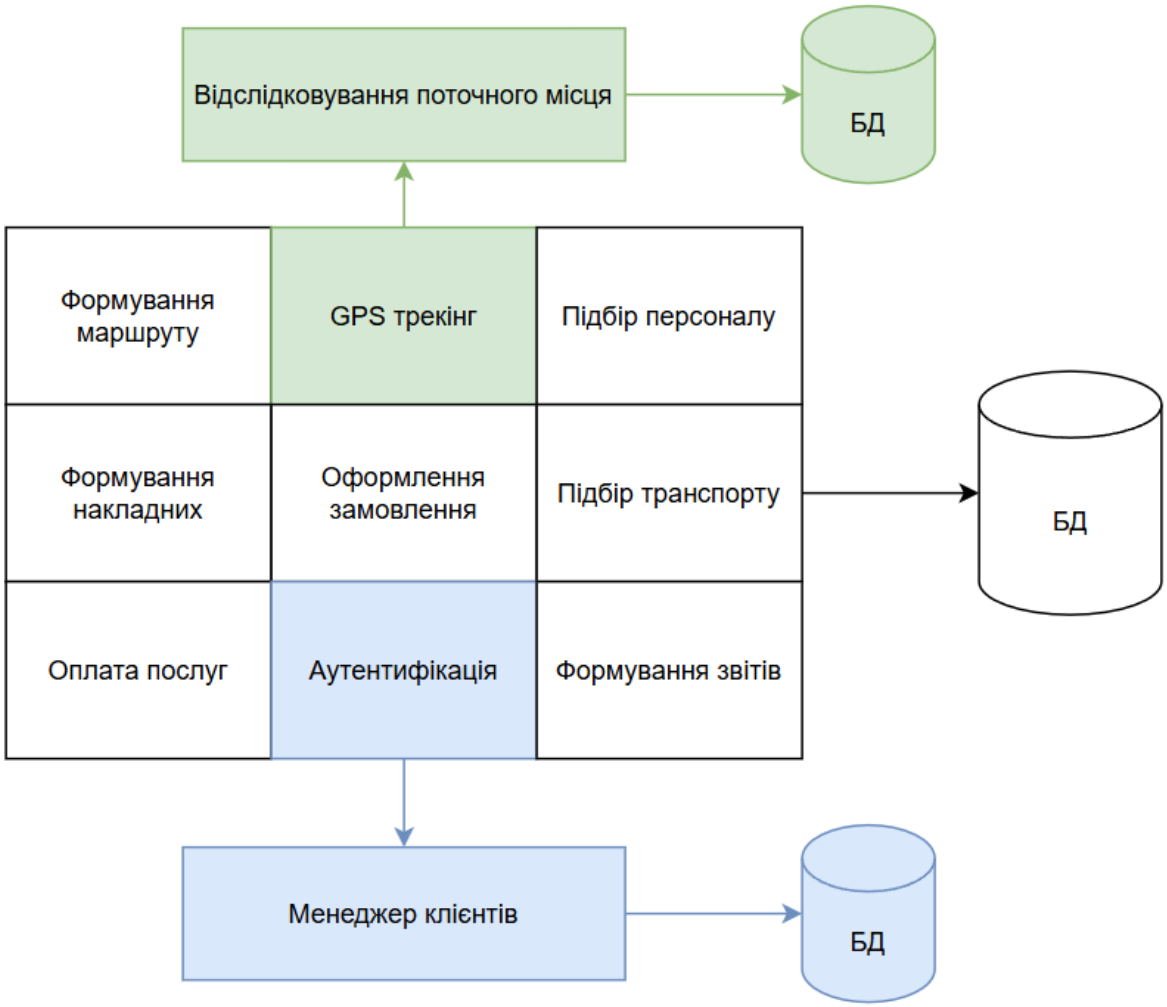

Рисунок 1.3 – Приклад використання патерна Strangler для покрокового перетворення моделей у мікросервіси

На основі всього сказаного можна зробити висновок, що мікросервісна архітектура є досить специфічною та складною для реалізації, а на ефективність її роботи можуть впливати різноманітні фактори.

1.2 Аналіз сучасних досліджень

Впродовж багатьох років активного використання мікросервісної архітектури, у проектах різної складності, розробники програмного забезпечення змогли

накопити широкий спектр знань у даному напрямку, що допомагають вирішувати велику кількість задач.

Не зважаючи на це, мікросервісна архітектура продовжує мати ряд критичних недоліків, що обмежує можливості її використання невеликим спектром задач. Сучасні дослідження акцентуються на спробах вирішення проблем, що виникають через недоліки архітектури.

Кожен дослідник намагається вирішити можливі проблеми різними способами. Одні намагаються передбачити можливі проблеми у розробці за допомогою аналізу технічного завдання та завчасно визначити можливі шляхи їхнього вирішення, тоді як інші пробують вирішити конкретні проблеми шляхом використання різних методів та технологій як окремо, так і разом.

Першим дослідженням, на яке було звернено особливу увагу є спроба систематизації оглядів літератури у напрямку вирішення викликів при розробці мікросервісної архітектури з назвою «Challenges and solution directions of microservice architectures: a systematic literature review» [1]. Автори дослідження проаналізували більше 3800 сторінок різного виду літератури пов’язаної з розробкою мікросервісної архітектури, що була опублікована починаючи з 2014 року.

На основі проведеного аналізу матеріалів дослідники змогли визначити дев’ять основних типів викликів, що можуть виникати при впровадженні архітектури. Дані виклики було розбито на сорок менших підвикликів, що являють собою конкретну проблему.

До основних типів викликів було віднесено:

- виявлення служб;
- управління даними та узгодженість;
- тестування;
- прогнозування, вимірювання та оптимізація продуктивності;
- комунікація та інтеграція;
- оркестрування сервісів;

– безпека;

– моніторинг, відстеження та логування;

– декомпозиція.

Автори дослідження також навели та коротко описали можливі варіанти вирішення проблем на основі проаналізованої літератури, але разом з цим вони наполягають на важливості продовження досліджень у напрямку мікросервісних архітектур.

З даної роботи можна зробити висновок, що перед початком розробки мікросервісної архітектури надзвичайно важливим є аналіз можливих викликів ще на етапі проектування та планування потенційних варіантів їхнього вирішення. Це може допомогти уникнути чи сильно зменшувати негативний вплив ряду проблем, що можуть виникати на різних етапах життєвого циклу.

Наступним розглянутим дослідженням було «On the definition of microservice bad smells» [2] у якому проводиться спроба визначити «погані запахи» специфічні для мікросервісної архітектури, що можуть негативно впливати на якість програмного забезпечення. Термін «поганий запах» використовується для опису певних специфічних ознак у коді, які можуть вказувати на потенційні проблеми чи недоліки у дизайні архітектури, що у подальшому можуть призвести до різного виду викликів при підтримці чи масштабуванні програмного продукту.

Автори дослідження провели опитування серед сімдесяти двох розробників, що мають досвід у створенні програмних продуктів на основі мікросервісної архітектури. Проаналізувавши результати вони змогли створити каталог із одинадцяти «поганих запахів», які зазвичай вважаються шкідливими практиками, та ряд практичних рекомендацій, що повинні допомогти уникнути чи вирішити проблеми даних практик. До основних «поганих запахів» відносять:

– жорстко закодовані кінцеві точки – використання жорстко закодованих URL-адрес, що використовуються для взаємодії між різними мікросервісами ускладнює зміну конфігурації системи;

– циклічна залежність – циклічна залежність між різними мікросервісами може сильно ускладнити їхнє розуміння, підтримку та модифікацію;

– спільна цілісність – спільне використання баз даних між різними мікросервісами, що призводить до зменшення їхньої незалежності;

– версії API – API мікросервісів повинні мати ідентичні версії для коректної взаємодії один з одним;

– використання ESB – взаємодія мікросервісів через корпоративну сервісну шину додає складності у реєстрації нових сервісів;

– відсутність API шлюзу – відсутність шлюзу призводить до того, що мікросервіси безпосередньо взаємодіють один з одним, а в гіршому випадку користувачі безпосередньо взаємодіють з мікросервісами, що сильно збільшує складність системи та ускладнює її підтримку;

– надмірна взаємодія між мікросервісами – надлишкова взаємодія мікросервісів між собою може ускладнювати тестування та розгортання, що може вказувати на некоректне розбиття задач по мікросервісам;

– спільні бібліотеки – використання спільних бібліотек між різними мікросервісами зменшує їхню незалежність;

– використання багатьох стандартів – використання занадто великої кількості різних мов програмування, стандартів, протоколів, технологій, тощо призводить до надмірного ускладнення системи;

– жадібні мікросервіси – розробники часто створюють окремі мікросервіси для кожної функції, навіть якщо вони не є доцільними, наприклад, для обслуговування однієї HTML сторінки може існувати окремий мікросервіс. Але бувають і зворотні ситуації – один мікросервіс відповідає за виконання надлишкової кількості різного функціоналу.

Здійснивши аналіз даного дослідження можна зробити висновок, що для забезпечення високого рівня якості програмного продукту варто уникати «поганих запахів», а з використанням створеного каталогу можна ще на ранніх етапах розробки мікросервісної архітектури уникнути ряду критичних проблем, що допоможе зменшити матеріальні та часові витрати на розробку, впровадження та підтримку систем.

Ще одним важливим дослідженням є спроба систематичного картографування наукових публікацій, що присвячені мікросервісній архітектурі з акцентуванням уваги на сучасні тенденції та дослідження можливостей для промислового провадження з назвою «Architecting with microservices. A systematic mapping study» [3]. За мету даної роботи було поставлено оцінку та класифікацію сучасних наукових підходів в даній галузі, визначення провалин у знаннях та оцінку перспектив впровадження у промисловість.

В даному дослідженні автори проаналізували п’ятсот тридцять дві публікації та за допомогою чітких критерій відібрали із них сто три роботи, на основі яких було проведено подальший аналіз наступних параметрів:

– тенденції у публікаціях – як змінювалась кількість наукових публікацій та їхні типи класифіковані по роках;

– цілі дослідження – які проблеми в архітектурі досліджувались та як оцінювались її характеристики;

– можливості впровадження – оцінка рівня зрілості технологій.

В результаті розробники відмітили, що до 2014 року публікації на тему мікросервісної архітектури майже повністю відсутні, але з 2015 року відбувся різний ріст її популярності, що тривав аж до 2017 року. Більшість досліджень публікувались в наукових журналах та на конференціях, тому було зроблено висновок, що наукова спільнота була зацікавлена темою мікросервісів.

У проаналізованих дослідженнях акцентувалась увага на гнучкості систем, можливостях ефективного і швидкого масштабування, а також складності розподілених систем у зв’язку з чим виникали виклики, що пов’язані з підтримкою надійної та коректної роботи сервісів. Окрім цього у роботі відмічається, що теми пов’язані з безпекою та тестування мікросервісної архітектури є недослідженими та потребують більшої уваги, а проблеми взаємодії різних сервісів та консистентність даних потребують більш детального розкриття.

Якщо ж говорити за можливості впровадження, то після проведення аналізу на технологічну готовність рішень для промислового впровадження, та було встановлено, що більшість досліджень перебувають на початкових стадіях. Це

означає, що їхні результати не можуть бути швидко впроваджені без проведення ряду додаткових розробок.

Проаналізувавши дану роботу можна стверджувати, що мікросервісна архітектура потребує збільшення кількості досліджень у різних напрямках та може ефективно виконувати поставлені на неї задачі, проте мікросервіси досі знаходяться на ранніх етапах розвитку.

1.3 Постановка задач та вимог

У попередніх розділах було встановлено, що розробка мікросервісної архітектури є надзвичайно складним та комплексним завданням, що потребує високого рівня знань та вмінь розробників архітектури. Навіть маленька помилка допущена на етапі розробки може призвести до критичних проблем, а враховуючи особливості архітектури, їх пошук та виправлення може потребувати великої кількості часових та матеріальних ресурсів.

Найчастіше неефективність програмного продукту на основі мікросервісної архітектури пов’язане з декількома факторами, до яких відносяться:

– невірно визначені мікросервіси та шляхи їхньої комунікації;

– вибір неефективного способу комунікації між мікросервісами чи інших пов’язаних технологій;

– некоректне розгортання системи.

На основі наявної інформації було зроблено висновок, що комунікація між різними сервісами є одним із ключових факторів, що впливають на швидкість роботи застосунків на базі мікросервісної архітектури.

На ефективність розроблених комунікацій може впливати багато факторів, наприклад, кількість мікросервісів, що беруть участь у виконанні запиту, методи комунікації, тощо. З даних причин можна стверджувати, що реалізації мікросервісної архітектури в контексті методів ефективної комунікації потребує проведення додаткових досліджень.

Для виконання даного завдання доцільним є проведення дослідження та тестування можливості одночасного застосування різних методів комунікації мікросервісів, що теоретично можуть перекривати недоліки один одного. Для цього доречним є проведення ряду тестів, що передбачають можливість комунікації ідентичних мікросервісів за допомогою різних підходів та дослідження їхньої ефективності.

Для цього необхідно передбачити ряд тестових сценаріїв у яких мікросервіси будуть виконувати ідентичне завдання, але комунікація між ними відбувається за допомогою різних підходів. Також важливим аспектом такого виду тестування є дослідження можливості одночасного застосування декількох підходів, що можуть автоматично змінювати спосіб комунікації.

Окремо варто акцентувати увагу на ефективності таких підходів при розгортанні архітектури за допомогою сучасних підходів. Використання додаткових інструментів та технологій можуть сильно впливати як на ефективність роботи мікросервісів, так і на їхню комунікацію.

1.4 Висновки

На першому кроці виконання даної роботи було проведено аналіз предметної області. В ньому було ознайомлено з мікросервісною архітектурою, коротко розглянуто історію її розвитку, принципи роботи. Також було визначено основні переваги та недоліки архітектури та встановлено основні випадки коли доцільно використовувати мікросервіси та сучасні підходи до визначення сервісів.

Наступним кроком було проведено аналіз сучасних досліджень. На основі розглянутих результатів було зроблено висновок, що через особливості мікросервісної архітектури її реалізація є досить складним завданням, а поява помилок може призвести до критичних наслідків, але її використання надає ряд переваг, що може включати зменшення витрат на підтримку стабільної роботи серверів та ряду інших факторів.

Для уникнення появи різного виду помилок чи їхнього негативного вплину на систему доцільним є проведення аналізу можливих викликів, що можуть виникати на різних етапах розробки, та визначення декількох потенційних варіантів їх вирішення. Також важливим є постійна перевірка та контроль проекту на наявність певного виду специфічних маркерів, що можуть вказувати на недоліки у реалізації та різного типу проблемні місця у системі.

На основі наявної інформації було зроблено висновок, що одним із критичних аспектів, що надзвичайно сильно впливає на ефективність роботи застосунків на основі мікросервісної архітектури є шляхи комунікації між сервісами. Від них напряму залежить як швидко користувач зможе отримати відповідь. З даних причин було визначено, що доцільним є проведення дослідження можливості одночасного використання різних способів комунікації між сервісами з врахуванням сучасних методів розгортання такого виду систем.

2 ПІДХОДИ ТА МЕТОДИ ВИРІШЕННЯ ПОСТАВЛЕНИХ ЗАДАЧ

2.1 Технології комунікації мікросервісів

Як вже було згадано у попередніх розділах, впродовж багатьох років активного використання мікросервісної архітектури, розробники програмного забезпечення змогли сформувати базовий набір технологій та підходів, проте навіть такі сучасні методи розробки продовжують мати критичні недоліки та мають певні складності у впровадженні.

Одним із найважливіших завдань при проектуванні мікросервісної архітектури є реалізація способу передачі даних. На швидкість та ефективність обміну інформації між сервісами впливає велика кількість факторів, наприклад, коректне визначення мікросервісів та формування зв’язків між ними, об’єм даних та кількість сервісів, що задіяні для виконання запиту, тощо. Проте, позбутись більшості таких факторів або надзвичайно складно, або повністю неможливо, тому розробники програмного забезпечення акцентують свою увагу на вибір та реалізацію способу обміну інформацією.

Для передачі даних між різними мікросервісами зазвичай прийнято використовувати синхронну комунікацію. Вона передбачає, що виконання запиту відбувається в режимі реального часу де один мікросервіс очікує відповідь від іншого сервісу і лише після цього може продовжити свою роботу.

Головною причиною активного застосування синхронного обміну повідомленнями є те, що при відправці запиту клієнт отримає негайну відповідь на нього, а також можливість легкого керування транзакціями для перевірки успішного виконання операції.

Зазвичай, для синхронного обміну інформації між мікросервісами використовується класичний текстово-орієнтований протокол комунікації HTTP, хоча в певних випадках перевага може надаватись використанню HTTPS. При його застосуванні обмін даними між сервісами відбувається у форматі JSON чи XML.

Протокол HTTP активно використовується при реалізації міжсервісної комунікації з наступних причин:

– HTTP є універсальним стандартом для веб-комунікацій, що активно підтримується різними браузерами, клієнтськими бібліотеками та веб-серверами;

– HTTP дозволяє підтримувати достатній рівень безпеки за допомогою стандартних протоколів шифрування SSL та TLS;

– підтримка HTTP більшістю моніторингових систем, тестових утиліт та IDE;

– формати JSON та XLM підтримується багатьма мовами програмування, а налаштування REST API є відносно простим.

Проте, не зважаючи на всі переваги, протокол HTTP не можна назвати універсальним в контексті різного типу програмного забезпечення. Даний протокол працює в форматі «запитання-відповідь», що є незручним у випадках, коли застосунок передбачає двонаправлений обмін даними у режимі реального часу. Також варто відмітити, що HTTP працює лише з методами GET, PUT, DELETE, POST, а отже не завжди є ефективним для складних чи високоефективних засобів.

Вирішити даний недолік можна використанням протоколу високоефективної передачі процедур GRPC. Він заснований на HTTP/2, що дозволяє виконувати кілька запитів як одне з’єднання, стискати заголовки та контролювати потік. Також даний протокол використовує Protobuf як мову визначення інтерфейсу для формування сервісів та структури повідомлень, що дозволяє виконувати бінарну серіалізацію даних.

Протокол GRPC підтримує чотири віддалені процедури виклику:

– односторонні виклики – це прості виклики по типу «запит-відповідь»;

– стрімінговий виклик з боку сервера – клієнт відправляє запит та отримує відповідь у вигляді потоку для читання послідовності повідомлень;

– стрімінговий виклик з боку клієнта – клієнт надсилає послідовність повідомлень та отримує одну відповідь;

– двосторонній стрімінговий виклик – обидві сторони відправляють послідовність повідомлень та використовують потік для читання та запису.

Якщо порівнювати GRPC з HTTP, то необхідно відмітити, що він має кращу продуктивність та менші затримки, а бінарна серіалізація дозволяє зменшити розмір надісланих даних.

Проте, як і HTTP, даний протокол не є універсальним рішенням. Головним недоліком GRPC є складність його впровадження та управління. Це пов’язано з тим, що даний протокол працює на HTTP/2, який, хоча і є ефективнішим, але складніший для налаштування та реалізації, а сервери та системи не завжди мають його повну підтримку, що може ускладнювати процес інтеграції. Також при його впровадженні можуть виникати інші труднощі, наприклад, з бінарним форматом даних складніше працювати, а підтримка протоколу в різних браузерах є обмеженою, тощо.

На основі наявної інформації можна зробити висновок, що визначення ефективних технологій обміну даними між мікросервісами є складним та надзвичайно важливим завданням. Підібрати ідеальний варіант реалізації комунікацій між мікросервісами дуже складно та, зазвичай, можливо лише для специфічного програмного забезпечення. Наприклад, для реалізації стандартного інтернет-магазину доцільним є використання протоколу HTTP, що дозволить у повній мірі реалізувати весь функціонал. Якщо ж розглядати програмне забезпечення, що працює з великими об’ємами даних, доцільно використовувати більш ефективні протоколи.

Проте, хоча і синхронний спосіб комунікації дозволяє швидко обмінюватись інформацією, не залежно від вибраного протоколу, даний підхід має загальні недоліки, що дуже складно вирішити. У зв’язку з тим, що синхронна комунікація передбачає пряму взаємодію між двома мікросервісами, а отже вони повинні знати про існування один одного. Через такий підхід до реалізації обміну даними схема взаємодії різних мікросервісів між собою може розростатись до великих масштабів, що призводить до появи посередників. Приклад варіанту такої структури можна побачити на рисунку 2.1.

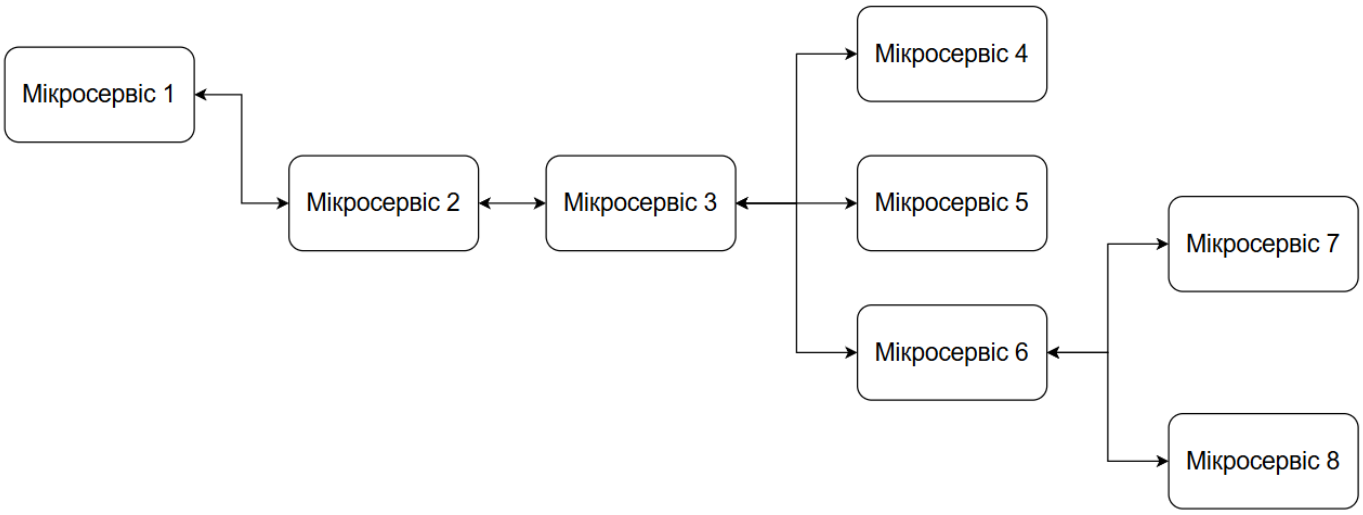

Рисунок 2.1 – Приклад можливої схеми шляхів комунікації між різними сервісами

На перший погляд схема є цілком логічною – за потреби перший мікросервіс звертається до другого, другий до третього і так далі. Коли останній у ланцюгу сервіс обробить запит він поверне відповідь попередньому, що буде відбуватись до моменту досягнення першого мікросервісу, що надіслав початковий запит.

Реалізація подібних схем комунікації на практиці, зазвичай, не є складним завданням, але такий підхід несе в собі великі ризики. Його головним недоліком є те, що у випадку виходу із ладу будь-якого одного мікросервісу, що залучений до виконання завдання, запит не зможе бути виконаний, а це може призвести навіть до повної зупинки роботи всієї системи.

Можливим варіантом вирішення даної проблеми може стати використання асинхронного способу передачі повідомлень. Основна ідея даного підходу полягає в тому, що сервіс може відправляти повідомлення одному або декільком сервісам без необхідності очікування відповіді. Таким чином сервіс, що надсилає повідомлення може продовжувати свою роботу, а сервіс, що отримав повідомлення зможе обробити його у зручний для себе час.

При асинхронному обміні повідомлень основними компонентами є:

– відправники – сервіси, що генерують та надсилають повідомлення до брокера;

– отримувачі – сервіси, що отримують і обробляють повідомлення;

– черга – структура, де брокер тимчасово зберігає отримані повідомлення у порядку їх надходження;

– топік – багаторівневий канал, де кожен підписник на топік отримує копію повідомлення.

Для забезпечення обміну повідомленнями між мікросервісами в асинхронний спосіб застосовується шина чи брокер повідомлень. Вони являють собою програмний компонент, що відповідає за обробку, маршрутизацію та зберігання повідомлень. При такій реалізації мікросервіси не взаємодіють напряму один з одним. В даному випадку брокер виступає як посередник між всіма мікросервісами. Це призводить до того, що кожному окремому мікросервісу необхідно знати лише про існування брокера без необхідності надання інформації про будь-які інші мікросервіси.

При асинхронному обміні повідомленнями відправник надсилає запит до брокера після чого він зберігає отримане повідомлення у чергу чи топік. Відразу після цього відправник звільняється і може виконувати роботу не очікуючи відповідь. У зручний для себе час отримувач забирає повідомлення з черги чи отримує його з топіку. Варто відмітити, що повідомлення зберігається до тих пір, поки його не отримає відповідний отримувач, що гарантує доставку повідомлення. На рисунку 2.2 можна ознайомитись із схемою роботи відправки асинхронних повідомлень через чергу та топік.

Такий підхід до обміну повідомленнями вирішує проблему стійкості до збоїв, адже компоненти працюють незалежно один від одного, що дозволяє системі продовжувати роботу при поломці одного чи декількох сервісів. Також такий підхід зменшує навантаження на мікросервіси, адже їм не потрібно очікувати відповідь.

Проте, даний спосіб комунікації позбавлений основних переваг синхронних методів. Через наявність посередника у вигляді брокера та через необхідність очікування поки отримувач одержить дані може збільшуватись час виконання запиту. Крім того даний підхід підвищує складність архітектури, а також ускладнює узгодженість даних, адже різні сервіси можуть бути не синхронізованими.

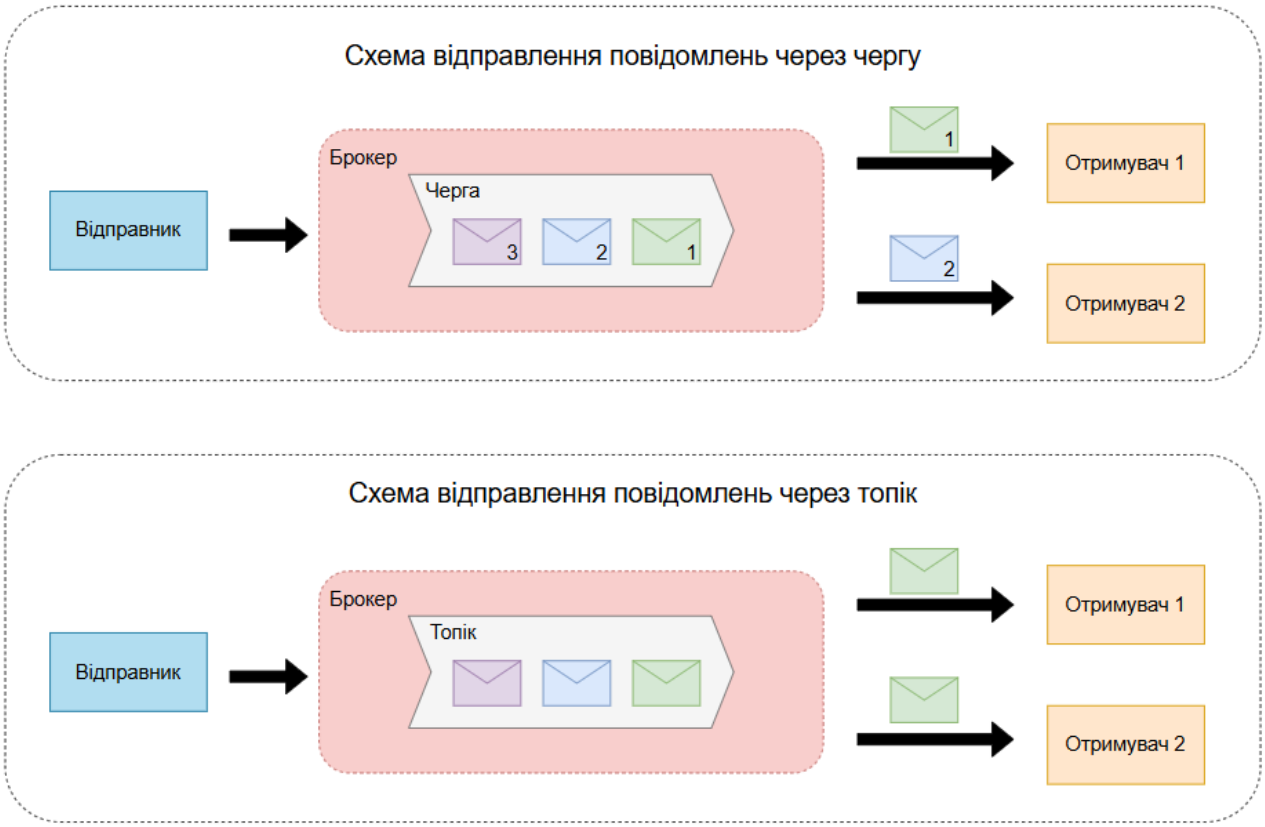

Рисунок 2.2 – Схема відправки асинхронних повідомлень через чергу та топік

На основі наведених даних було зроблено висновок, що як синхронні, так і асинхронні методи обміну повідомленнями мають свої переваги та недоліки, а також часто застосовуються у різних ситуаціях.

Окремо варто відмітити те, що дані методи перекривають недоліки один одного, а отже було зроблено припущення, що їхнє одночасне використання є потенційно корисним та може надати більше переваг ніж недоліків, що можуть виникнути при їхній реалізації та впровадження.

З даних причин доречним є проведення дослідження щодо можливості одночасного використання синхронних та асинхронним методів обміну повідомлення між сервісами. Детальне тестування та аналіз отриманих результатів допоможе визначити доцільність використання такого підходу та можливі виклики, що можуть виникати при розробці програмного забезпечення.

Ще однією важливою проблемою у мікросервісній архітектурі є процес розгортання програмного забезпечення. Трьома найпоширенішими підходами до

розгортання є традиційний метод, архітектура на основі віртуалізації та підхід з використанням контейнерів.

При використанні традиційного підходу всі застосунки розгортають безпосередньо на операційній системі одного фізичного сервера. Таким чином застосунки ділять між собою спільні ресурси системи, наприклад, пам’ять, дисковий простір, можливості процесора, тощо.

Такий спосіб розгортання легко використовувати, адже він не потребує додаткових слоїв абстракцій, забезпечує швидкий доступ до апаратних ресурсів, адже має прямий доступ до них, та є дешевим у реалізації, проте у даному підході повністю відсутня ізоляція між застосунками, а ресурси системи можуть використовуватись неефективно.

Інший підхід до розгортання базується на віртуалізації. Даний підхід передбачає, що на одному фізичному сервері встановлюється гіпервізор – програмне забезпечення, що створює та керує віртуальними машинами. Кожна віртуальна машина має власну операційну систему, що може запускати окремі застосунки.

Даний спосіб розгортання дозволяє забезпечити максимальний рівень ізоляції, адже кожна віртуальна машина ізольована від інших. Крім того такий підхід дозволяє ефективно розподіляти ресурси між різними сервісами відповідно до їхніх потреб, а можливість вибору операційної системи може допомогти зменшити ресурсні затрати без порушення вимог.

Проте, використання гіпервізора призводить до необхідності залученням додаткових ресурсів. Окрім того кожна віртуальна машина має власну операційну систему, що вимагає виділення додаткових потужностей у вигляді оперативної пам’яті, дискового простору та можливостей обчислювальної машини, а між самими віртуальними машинами можуть виникати високі затримки через необхідність використання посередника.

Третій спосіб розгортання програмного продукту передбачає застосування контейнерів, що являє собою легковажні та ізольовані середовища запуску програмного забезпечення, що забезпечують достатній рівень ізоляції, хоча і не є повністю ізольованими, у порівнянні з використанням віртуальних машин. Кожен

контейнер зберігає файли, бібліотеки, конфігурації та залежності, що необхідні для коректної роботи сервісу.

Для роботи контейнерів використовується лише одна операційна система, а це означає, що всі контейнери ділять між собою ресурси системи. Для управління різними контейнерами використовується контейнерний рушій на рівні операційної системи. Загальні схеми роботи кожного з наведених способів розгортання зображено на рисунку 2.3.

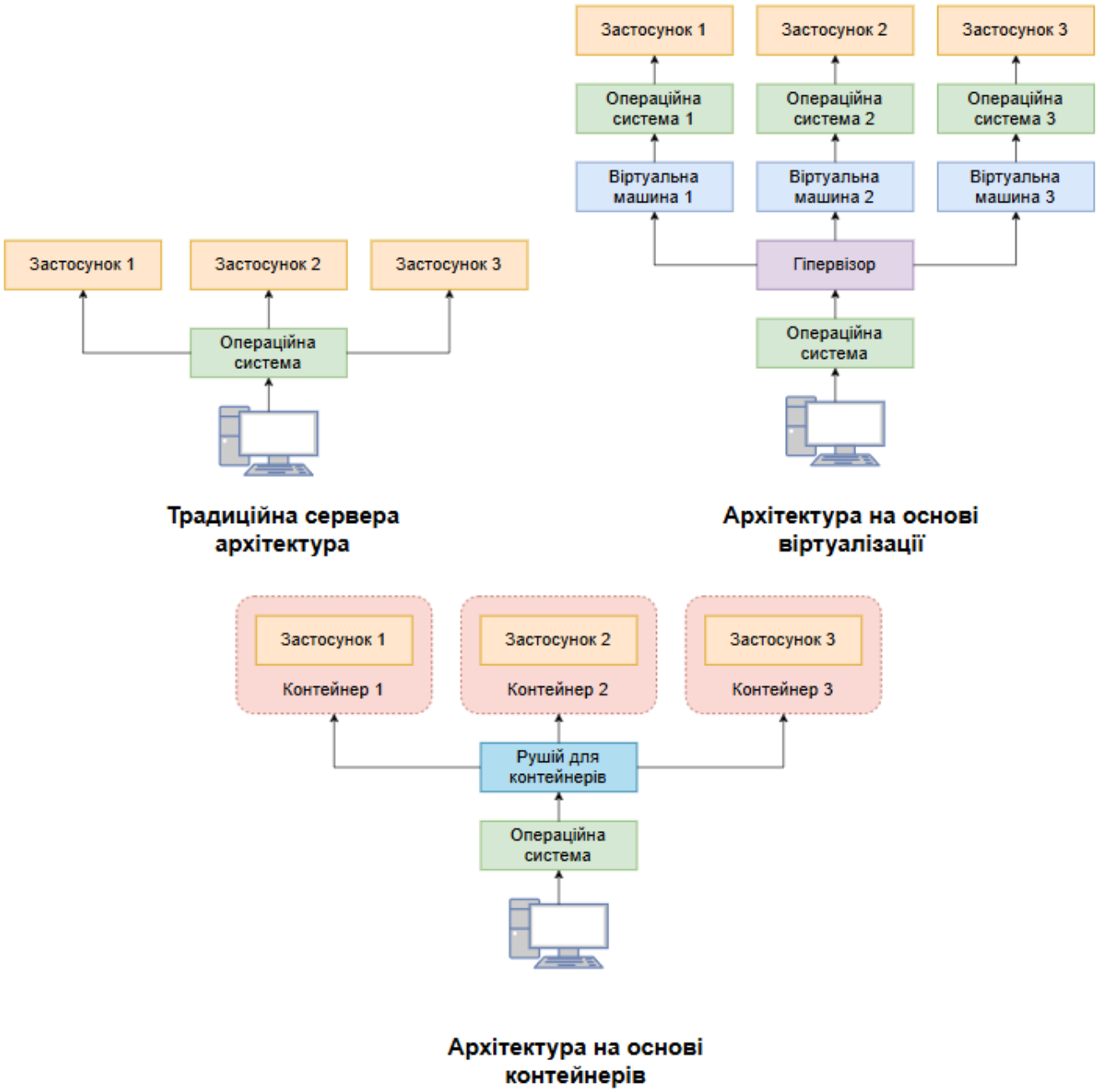

Рисунок 2.3 – Сучасні схеми розгортання програмного забезпечення

Будь-який із наведених способів розгортання системи можуть надзвичайно сильно впливати на її продуктивність. Першочергово це пов’язано з тим, що різні підходи можуть передбачати необхідність використання додаткових компонентів чи технологій, що потребують виділення додаткових ресурсів, що впливають на шляхи комунікації мікросервісів.

При коректно проведеному процесі розгортання покращити ефективність роботи системи дуже складно. Зазвичай, найпростішим способом для виконання даної задачі є покращення апаратно-технічних характеристик серверу на якому розгорнуто програмне забезпечення.

Проведення такого виду покращень потребує великих матеріальних витрат, що не завжди може бути можливим. З даних причин доцільним є розгляд того як різні способи розгортання впливають на ефективність роботи мікросервісної архітектури та їхній вплив на їхні комунікаційні зв’язки.

2.2 Методологічні підходи до дослідження способів комунікації мікросервісів

У попередньому розділі було встановлено, що основними сучасними підходами до реалізації комунікації між мікросервісами є синхронні та асинхронні методи. Кожен з них має власні переваги, недоліки та складності у впровадженні. Зазвичай в рамках одного програмного продукту використовується лише один спосіб комунікації між сервісами, проте комбінація різних підходів до обміну даними може покращити швидкість та ефективність застосовуваного програмного забезпечення.

З даних причин доцільним є проведення тестування даного підходу з метою дослідження доцільності та ефективності використання різних методів комунікації як окремо, так і одночасно в рамках одного програмного продукту.

Для виконання даного завдання було визначено ряд синхронних та асинхронних методів міжсервісної комунікації, що необхідно дослідити в контексті даної роботи. До них відносяться:

– протокол HTTP – найбільш поширений синхронний протокол передачі даних, що підтримується більшістю систем;

– фреймворк GRPC – віддалена система виклику процедур, що використовує протокол HTTP/2. У порівнянні з HTTP є більш швидкою для обміну великих об’ємів даних, проте більш складний для реалізації;

– брокер повідомлень – використовується як інструмент для асинхронної передачі даних, що виступає як посередник між всіма мікросервісами та дозволяє зменшити залежність сервісів від часу відповіді.

Для дослідження даних методів комунікації достатньо було реалізувати три мікросервіси, що в повній мірі дозволять дослідити ефективність роботи різним методів комунікації в різних ситуаціях та з різними даними. Початковий запит буде надходити до першого мікросервісу. Для виконання отриманого завдання він буде надсилати його далі до другого сервісу, а вже він далі перенаправлятиме його до третього мікросервісу.

При такому підході до реалізації важливим є розмір даних, що передається між сервісами, адже залежно від їхнього об’єму буде змінюватись ефективність комунікації з врахуванням поточного способу передачі інформації.

Таким чином принцип роботи мікросервісів може бути наступним: перший сервіс отримує запит на відправку певних даних заданого розміру та передає їх до другого сервісу. Другий мікросервіс за потреби змінює розмір отриманого файлу та передає його далі третьому сервісу, що формує звіт щодо виконаного завдання та повертає результат.

З допомогою такого підходу можна оцінити як кількість та формат обмінюваних даних впливають на ефективність роботи кожного підходу як окремо, так і разом із іншими методами комунікації, що допоможе максимально детально дослідити дане питання.

Для проведення даного дослідження було заплановано серію наступних тестів, що включають:

– вимірювання продуктивності HTTP, GRPC та брокеру повідомлень у ізольованому режимі роботи;

– тестування комбінацій HTTP та GRPC, HTTP і брокера повідомлень, а також GRPC та брокера повідомлень в різному порядку їхнього застосування при обміні даними.

На основі отриманих результатів тестування з врахуванням обсягу даних та частоти запитів можна оцінити ефективність використання як одного, так і декількох методів комунікації сервісів та дозволить провести їхнє детальне порівняння з визначенням найбільш ефективних підходів враховуючи швидкість обміну даними.

Загальні схеми тестування таких комбінацій комунікацій зображені на рисунках 2.4 та 2.5.

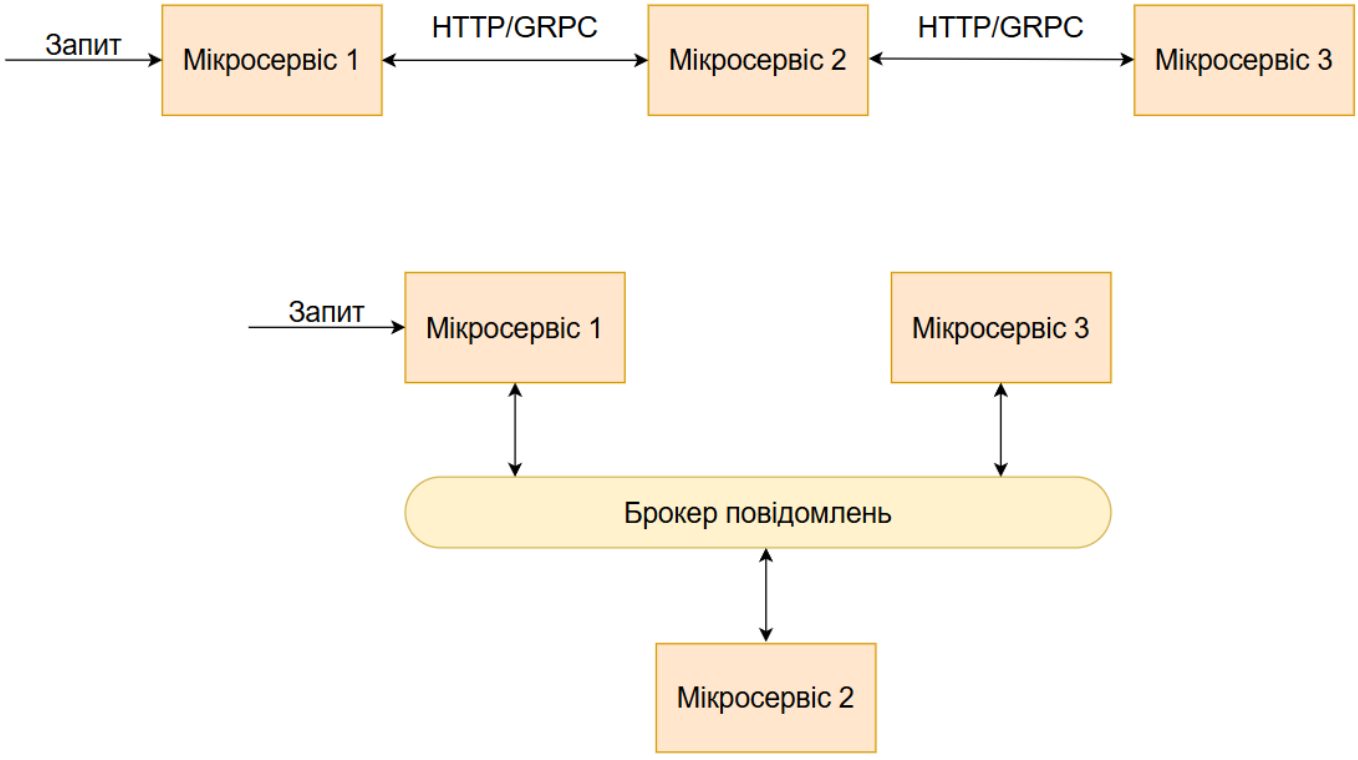

Рисунок 2.4 – Схеми комунікації сервісів при тестуванні ізольованого режиму роботи

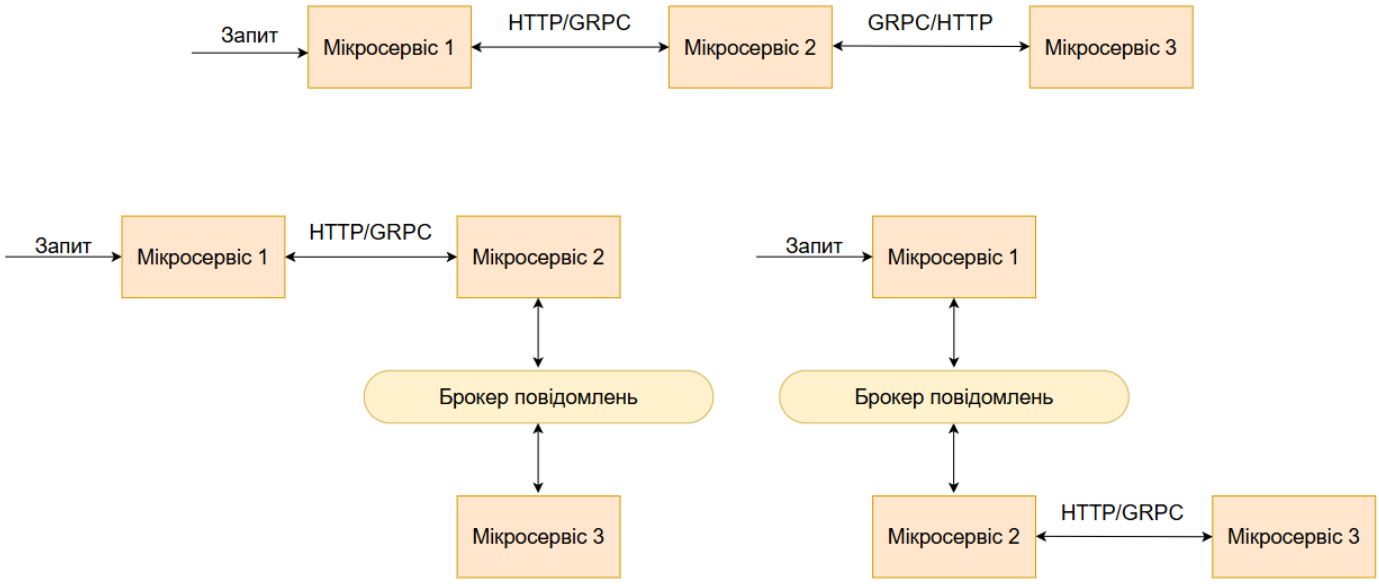

Рисунок 2.5 – Схеми комунікації сервісів при одночасному тестування декількох методів комунікації

В контексті даних досліджень необхідно акцентувати увагу на те, що спосіб розгортання програмного забезпечення може надзвичайно сильно впливати на швидкість обміну даними між мікросервісами та ефективність роботи всієї системи. З даних причин, для наведених вище видів тестувань, необхідно дослідити вплив способу розгортання програмного продукту на швидкість комунікації між мікросервісами та ефективність їхньої роботи.

Для даного дослідження найменш доцільною до використання є архітектура на основі віртуалізації. Хоча вона і має ряд важливих переваг над іншими методами розгортання, проте використання даного підходу може включати в себе велику кількість сторонніх факторів, що впливатимуть на ефективність комунікації між сервісами.

Це пов’язано з тим, що кожен сервіс повинен працювати на власній операційній системі, що розміщена на віртуальній машині та управляється за допомогою гіпервізора. Таким чином у схемі комунікації сервісів з’являються додаткові проміжні компоненти, що призводить до збільшення затримок. Також, в

даному випадку, необхідно враховувати ряд факторів, що включають версію операційних систем, що встановлені на віртуальних машинах, параметри їхніх налаштувань, тощо. Це призводить до того, що на результат тестування можуть впливати фактори, які не можна ефективно контролювати чи відслідковувати.

Кращим варіантом є використання традиційного підходу до розгортання. Даний підхід позбавлений недоліків попереднього, адже у ньому відсутні будь-які додаткові компоненти, що ускладнюють комунікацію. Він чудово підходить для тестування розробленого програмного забезпечення, проте через повну відсутність ізоляції один сервіс може впливати на інший, а отже отримані результати можуть виявитись не релевантними.

Враховуючи дані фактори доцільним є проведення тестувань з використанням традиційного підходу та контейнеризації. Таке подвійне тестування дозволить оцінити вплив різних підходів до розгортання програмного продукту.

Для оцінки ефективності комбінованого використанні різних методів комунікації та того як вони впливають на продуктивність роботи всієї системи. Першим типом тестування є інтеграційне. Його ціллю є оцінка взаємодії мікросервісів з використанням різних підходів до розгортання. Даний вид тестування передбачає:

– розгортання всіх мікросервісів за допомогою традиційного підходу та контейнерів;

– створення тестового сценарію в якому один мікросервіс відправляє запит іншому з використанням синхронних та асинхронних методів;

– проведення тестів та оцінку отриманих результатів.

Для вимірювання часу, який необхідний для виконання запиту було застосовано тестування затримок. З його допомогою можна визначити найкращі та найгірші комбінації методів комунікацій. Для його проведення використовується спеціальні інструменти, що дозволяють виміряти час обробки запитів та затримки.

2.3 Висновки

У даному розділі було проведено аналіз сучасних методів реалізації комунікацій між мікросервісів, їхні переваги та недоліки. Серед основних способів комунікації було розглянуто синхронні та асинхронні методи у контексті використання HTTP, GTPC та брокера повідомлень.

Перевагою синхронних методів є те, що вони є більш простими у реалізації та дозволяють швидко отримувати відповіді на запити, проте при такій реалізації комунікації мікросервіси повинні знати про існування один одного, адже вони взаємодіють напряму. Також при розробці великих проектів шляхи комунікації можуть сильно розростатись, а при поломці одного із сервісів, що включені у шлях виконання запиту, може відбутись здій, що призведе до неможливості отримання доступу до інших мікросервісів.

У випадку асинхронного обміну повідомленнями навіть при поломці одного чи декількох з мікросервісів система буде працювати стабільно, а запит буде збережений в брокері повідомлення до моменту відновлення роботи відповідного мікросервісу та отримання відповідного запиту.

Хоч даний підхід і є надійнішим, але час затримки для виконання запиту є більшим, що може бути критичним у деяких випадках. Також такий підхід створює вузьке місце у системі у вигляді брокера. При його поломці один сервіс не зможе виконувати жодних запитів до інших сервісів, а отже програмний продукт не зможе коректно працювати.

На основі даної інформації було зроблено висновок, що одночасне використання різних методів передачі даних може покращити швидкість виконання запитів та підвищити стабільність роботи всієї системи, але при цьому необхідно враховувати спосіб розгортання сервісів, адже він може впливати на різноманітні аспекти системи.

Для дослідження даного аспекту було визначено ряд можливих варіантів комунікацій між сервісами та методи їхнього тестування, що включають

інтеграційне та навантажувальне тестування, тестування затримок та відмовостійкості, що дозволять у повному розмірі оцінити ефективність роботи системи та її залежність до обраних методів обміну даними.

3 ПРОЕКТУВАННЯ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ

3.1 Аналіз вимог до програмного забезпечення

На основі отриманої інформації у попередніх розділах та з врахуванням визначених методів дослідження ефективності комунікації між мікросервісами, залежно від способу передачі даних та їхнього об’єму, з врахуванням вибраного методу розгортання, було сформовано розширені вимоги до розроблюваного програмного продукту.

Даний застосунок повинен складатись мінімум із трьох основних мікросервісів, кожен з яких має відповідати ряду наступних спільних для них всіх вимог до реалізації:

– мати можливість взаємодіяти з іншими сервісами в рамках визначених можливостей, що може відбуватись як за допомогою синхронних, так і асинхронних методів комунікації;

– до синхронних методів передачі даних необхідних для реалізації відносяться протокол HTTP та GRPC. Асинхронний метод комунікації повинен реалізовуватись на основі брокера повідомлень з використанням черги повідомлень для зберігання та відправки інформації;

– у випадках неможливості виконання запиту з допомогою брокеру повідомлень комунікація між мікросервісами повинна автоматично перенаправлятись на використання доступного на момент виконання запиту синхронного метода;

– підтримувати CRUD операції (створення, читання, оновлення видалення) для даних з якими вони працюють;

– наявність окремої бази даних для кожного мікросервісу для забезпечення незалежності та ізоляції даних;

– під час виконання завдання сервіс повинен збирати інформацію про час виконання всіх можливих задач;

– система повинна підтримувати традиційний спосіб розгортання та розгортання на основі контейнерів;

– використання контейнерів повинно бути максимально можливо оптимізовано для мінімізації споживання ресурсів та впливу на продуктивність виконання запитів.

Оскільки основною метою створення даного програмного продукту є тестування швидкості комунікації між мікросервісами з врахуванням різних параметрів, їхній функціонал необхідно обмежити до максимально можливого мінімуму, що обмежується задачами для роботи з обробки отриманих даних, їхню подальшу відправку та зберіганням даних щодо швидкості виконання кожного окремого запиту.

У зв’язку з даним фактором перший мікросервіс повинен виконувати лише наступні задачі:

– підтримувати CRUD операцій для роботи з даними, що використовуються для тестування швидкості обміну даних між сервісами;

– створення запитів до наступних мікросервісів із врахуванням обраного методу комунікації та необхідного для тестування об’єму даних;

– збереження інформації, що стосується швидкості виконання задач відповідним сервісом

– повернення результату виконання запиту у вигляді інформації щодо часу виконання обміну даними та мітками, що дозволять встановити у яких мікросервісах було проведено маніпуляції.

Другий мікросервіс повинен мати можливості для:

– виконання CRUD операцій для роботи з даними;

– збереження даних про швидкість виконання критичних задач;

– обробка отриманих даних від першого мікросервісу залежно від отриманих параметрів запиту, що може передбачає збільшення чи зменшення їхнього об’єму до заданого розміру;

– повернення отриманого результату з третього мікросервісу до першого сервісу.

Третій сервіс повинен виконувати наступні задачі:

– виконання CRUD операцій;

– збереження інформації про швидкість виконання критичних задач;

– обробка отриманих даних;

– формування звіту щодо швидкості виконання критичних завдань та повернення результату до другого мікросервісу.

У зв’язку з тим, що даний програмний продукт реалізується для тестування ефективності одночасного використання різних методів комунікації в рамках одного застосунку, тип та структура даних, що передається між різними мікросервісами повинні бути приведені до одного спільного формату.

Такий підхід до реалізації унеможливить появу нерелевантних результатів, що пов’язані через зміну формату даних, що в свою чергу може призвести до зміни об’єму обмінюваної інформації чи ряду інших її параметрів, що певним чином можуть впливати на роботу розробленого застосунку.

Для створення даних необхідного об’єму перший та другий мікросервіс повинні мати ідентичний метод для створення об’єкту заданого розміру, що визначається у мегабайтах, з можливістю зміни їхнього об’єму при появі подібної необхідності.

Окремо варто акцентувати увагу на виборі бази даних. У зв’язку з роботою із великими об’ємами інформації під час проведення тестування швидкість доступу до інформації відіграє ключову роль та безпосередньо впливає на швидкість виконання запитів. Для уникнення потенційних проблем пов’язаних з даним аспектом доцільним є вибір бази даних, що дозволить максимально швидко та ефективно отримувати доступ до інформації з мінімально можливими затримками та ресурсними витратами.

Для взаємодії з мікросервісами необхідно передбачити єдину точку входу у вигляді API Getaway для організації управління трафіком. Важливо врахувати, що дана точка доступу стане ще одним компонентом у ланцюгу через який необхідно

пройти для виконання поставленого завдання. З даних причин обраний спосіб реалізації API Getaway повинен мінімально впливати на швидкість виконання запитів. Таким чином його впровадження може допомогти покращити балансування навантаження між різними сервісами та дозволить отримати додаткові дані про запити, що проходили крізь нього.

Як вже було визначено раніше, доцільним є проведення дослідження впливу способу розгортання на ефективність роботи мікросервісної архітектури. Для цього необхідно передбачити можливість використання традиційного способу розгортання системи, а також підходу, що передбачає можливість впровадження контейнерів для ізоляції мікросервісів.

Для цього інструментарій, що використовується для розгортання системи, повинен дозволяти ефективно управляти, розгортати і масштабувати контейнери з необхідністю залучення мінімальної кількості додаткових ресурсів, що повинно мінімально впливати на ефективність роботи системи.

Ще одним важливим аспектом вимог до даного програмного забезпечення є необхідність моніторингу та логування. Моніторинг дозволяє відстежувати продуктивність та доступність кожного мікросервісу, що є критично важливим для дослідження ефективності комбінацій різних методів комунікації. Логування, в свою чергу, допомагає детально аналізувати роботу кожного мікросервісу, збираючи інформацію про всі запити. Основою цілю впровадження логування є відслідковування помилок та їхніх причин під час роботи програмного продукту, що повинно допомогти визначити найбільш слабкі місця проекту на етапі тестування та швидко виправити їх.

Таким чином використання моніторингу та логування мають забезпечити можливість ефективної оцінки часу відгуку та затримок різних методів комунікацій та виявити вузькі місця в архітектурі, а при появі поломок дані інструменти повинні вказувати на можливі аномалії.

3.2 Розробка структури програмного забезпечення

Як вже було визначено у попередніх розділах, для проведення тестувань ефективного використання різних підходів до організації обміну даними між мікросервісами, необхідно передбачити та реалізувати коректні і максимально ефективні шляхи комунікації.

Для виконання даного завдання було вирішено розробити діаграму архітектури програмного продукту, з врахуванням всіх наявних вимог та обмежень. Варто зазначити, що діаграма була створена для системи, розгорнутої на базі контейнерів, а у випадку локального розгортання єдиним відсутнім елементом системи будуть ці ж контейнери. Ознайомитись із отриманою діаграмою можна на рисунку 3.1.

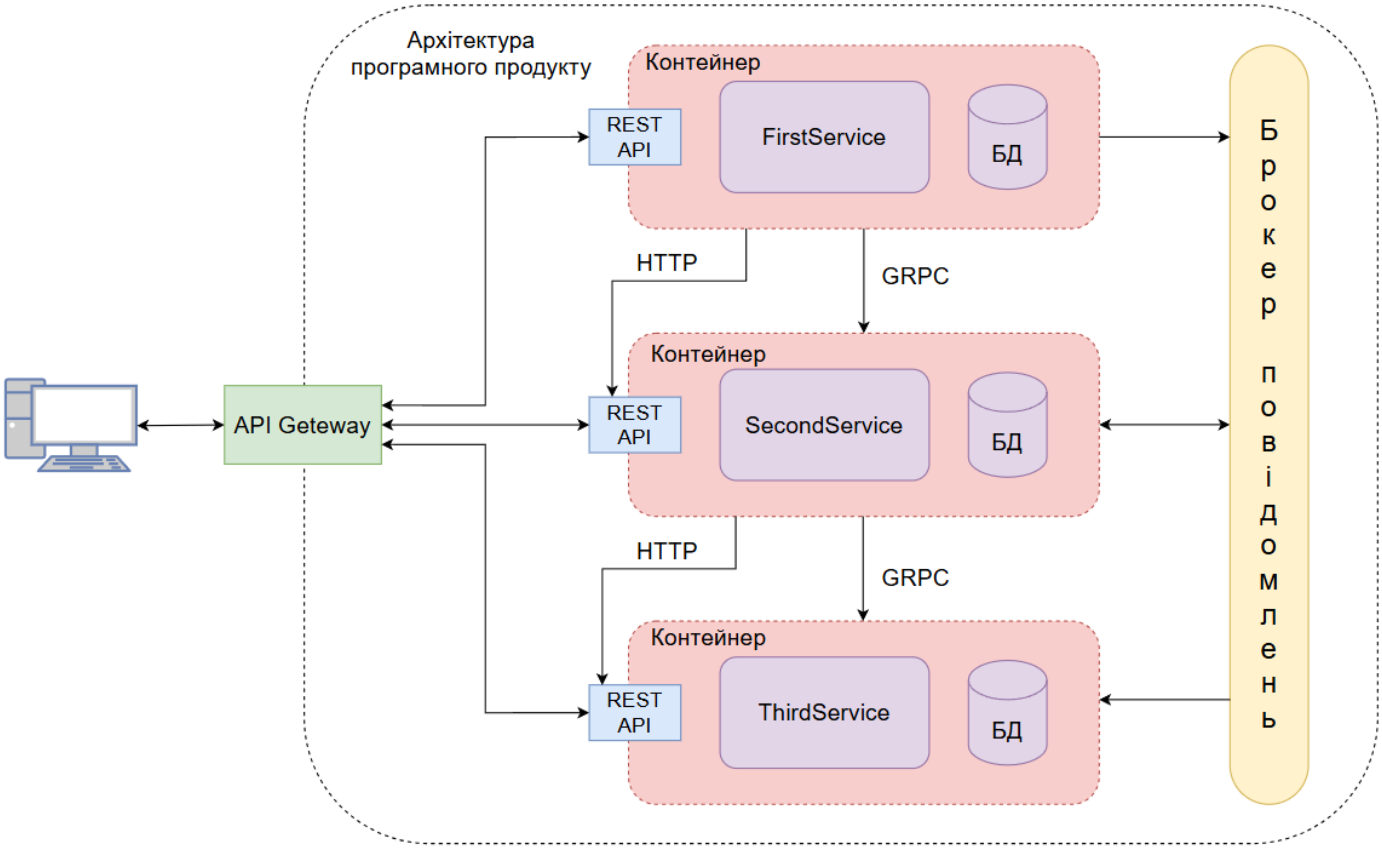

Рисунок 3.1 – Архітектура розроблюваного програмного забезпечення

На даному рисунку можна побачити, що кожен мікросервіс розміщений у власному ізольованому контейнері. Для виконання запитів зовні використовується API Gateway, що являє собою єдину точку входу всіх клієнтських запитів до будь-якого з наявних мікросервісів. Для комунікації між ним та мікросервісами використовується REST API.

Крім того, структура програмного продукту передбачає використання як синхронних, так і асинхронних методів обміну даними між різними сервісами. Це може включати синхронні HTTP чи GRPC запити, а також асинхронні повідомлення через системи черг.

Щодо структури звернень, то можливість комунікації між мікросервісами організована таким чином, що перший мікросервіс може звернутися до другого, але другий не може звернутися до першого. Це забезпечує чітку структуру взаємодії і допомагає уникати циклічних залежностей.

Більш детально варто розглянути внутрішню структуру кожного мікросервісу. У зв’язку з тим, що перший мікросервіс призначений для відправлення запитів до другого мікросервісу та не отримує запитів від інших сервісів, у ньому наявний лише один контролер, що містить методи для обробки даних та вибору способу передачі даних.

Для відправлення даних до наступного сервісу передбачена наявність трьох клієнтів кожен з яких відповідає за свій метод передачі даних, а саме HTTP, GRPC чи через брокер повідомлень. Їхня задача полягає у встановлені кінцевої адреси відправки інформації, підготовці даних, що будуть надіслані наступному мікросервісу, надсилання інформації та обробці з повернення отриманого результату до метода з якого було здійснено надсилання даних. Ознайомитись із структурою першого мікросервісу можна на рисунку 3.2.

На даному рисунку можна побачити, декілька додаткових компонентів, що не були описані раніше. Першим із них є моделі. Вони являюсь собою репрезентацію даних, що зберігають у базі даних, до яких можна віднести інформацію, що обмінюється між мікросервісами, та дані про виконання запиту.

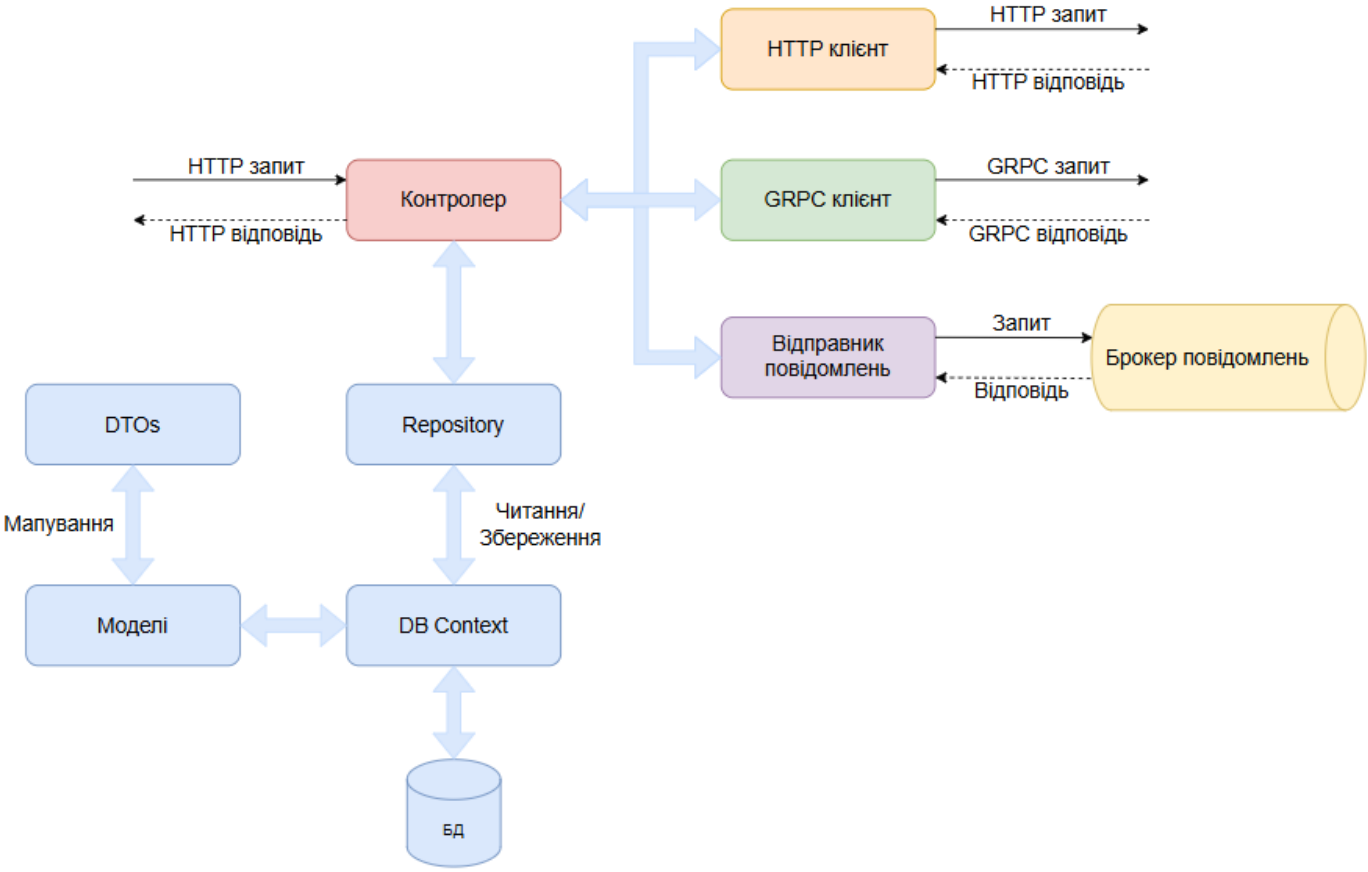

Рисунок 3.2 – Структура першого мікросервісу

Наступним елементом є DTOs, що представляє собою спрощену модель даних, що використовується для передачі інформації від одного мікросервісу до іншого.

Для отримання доступу до бази даних використовується об’єкт DB Context, що забезпечує зв’язок між моделями та реальною базою даних та дозволяє виконувати запити та управляти інформацією збереженою в базі даних.

Останнім елементом є репозиторій, що являє собою шаблон, що забезпечує абстракцію для роботи з базою даних та виступає посередником між контекстом даних та бізнес-логікою програмного забезпечення. Основною ціллю його використання є забезпечення можливості читання, запису, оновлення та видалення записів із бази даних.

Всі описані раніше елементи є базовими для кожного мікросервісу та виконують ідентичні завдання.

Наступним доцільно розглянути структуру другого мікросервісу. Він є подібним до першого, проте отриманий запит може оброблятись в одному із трьох компонентів. Якщо передачу даних було здійснено за допомогою протоколу HTTP, то дані передаються до контролера, де обробляються та передаються до третього мікросервісу відповідним методом.

При передачі даних за допомогою GRPC запит буде надходити на відповідний GRPC сервіс, що обробить отриманий результат та здійснить передачу до наступного мікросервісу. Ідентичним чином працює виконання запиту через брокер повідомлень, проте в даному випадку сам брокер виступає додатковим проміжним елементом між клієнтом та отримувачем. Ознайомитись із структурою другого мікросервісу можна на рисунку 3.3.

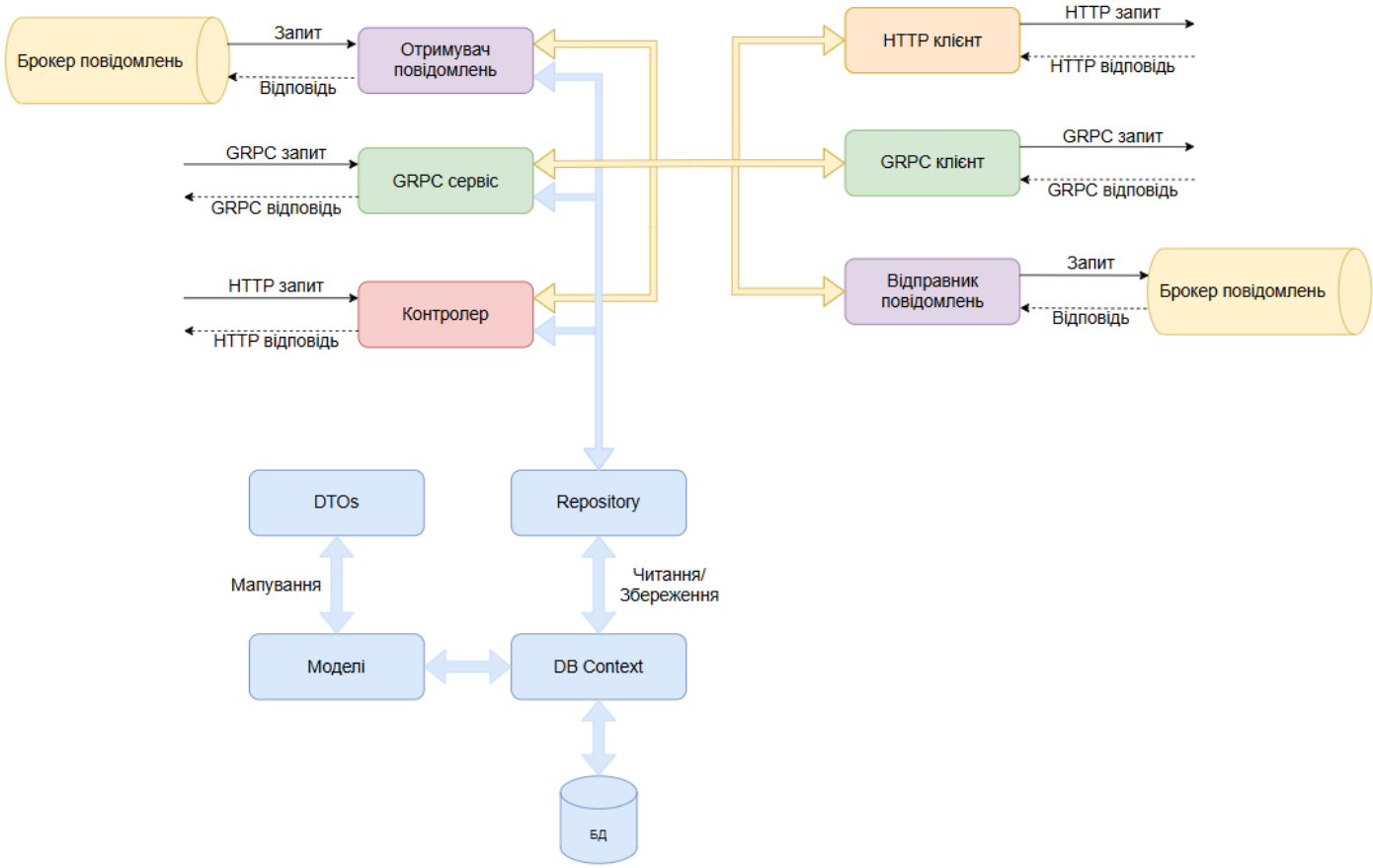

Рисунок 3.3 – Структура другого мікросервісу

Далі було розглянуто структуру третього мікросервісу. У зв’язку з тим, що він є останнім у ланцюгу виконання отриманого запиту, його основною задачею є отримання запиту, формування фінального результату та повернення його до першого мікросервісу. З даних причин в третьому мікросервісі наявні лише компоненти, що необхідні для отримання та обробки запиту, що було отримано від другого сервісу. Ознайомитись із структурою третього мікросервісу можна на рисунку 3.4.

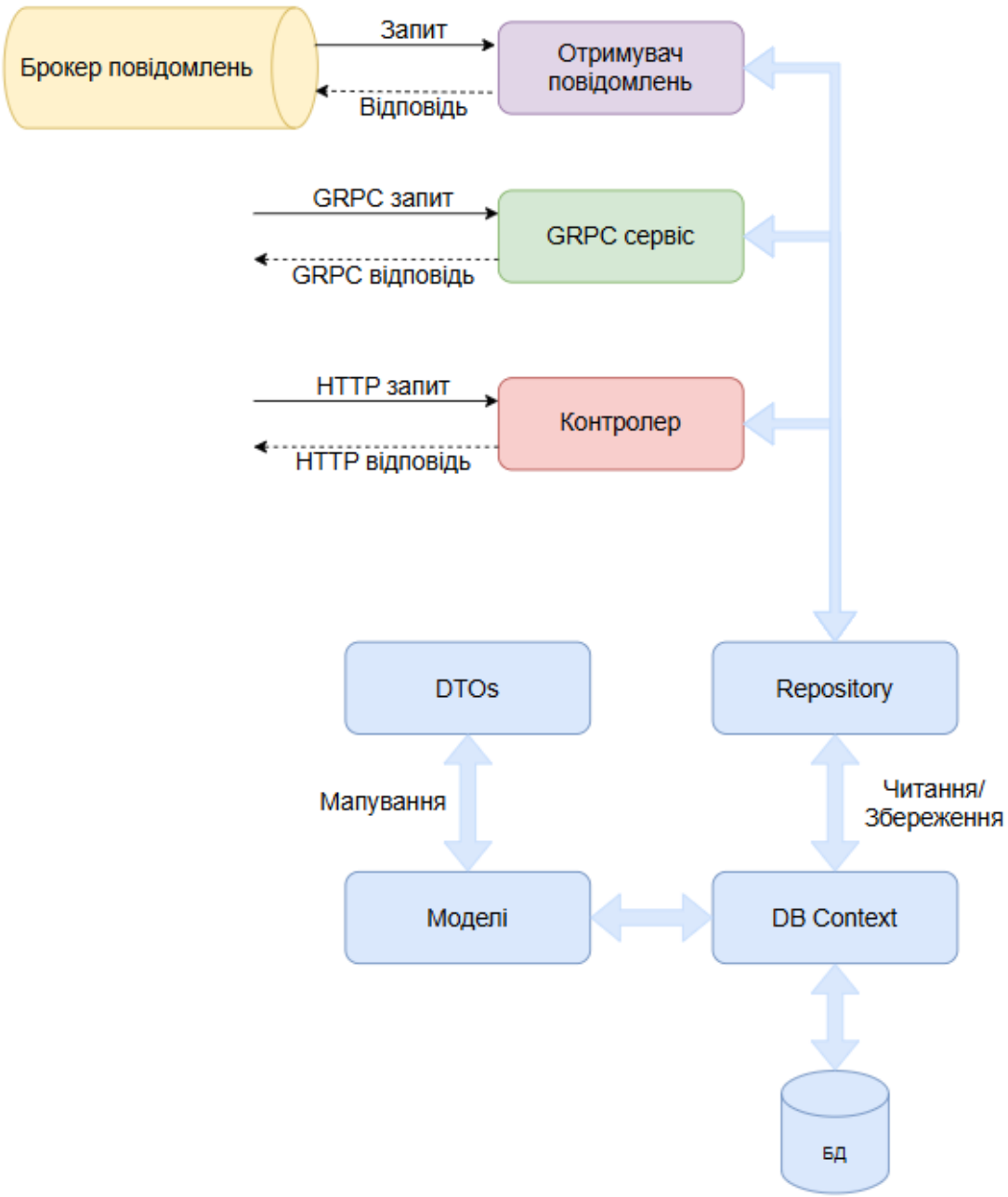

Рисунок 3.4 – Структура третього мікросервісу

Наступним важливим аспектом даного програмного продукту є підхід до комунікації мікросервісів. Як вже було сказано раніше, для виконання запиту перший сервіс відправляє дані до другого, а другий до третього мікросервісу після чого результат їхньої роботи повертається назад до першого мікросервісу.

Для кращого розуміння процесу комунікації мікросервісів було вирішено розробити діаграму послідовності, що демонструє яким чином відбувається процес взаємодії мікросервісів. Ознайомитись із відповідною діаграмою можна звернувшись до рисунку 3.5.

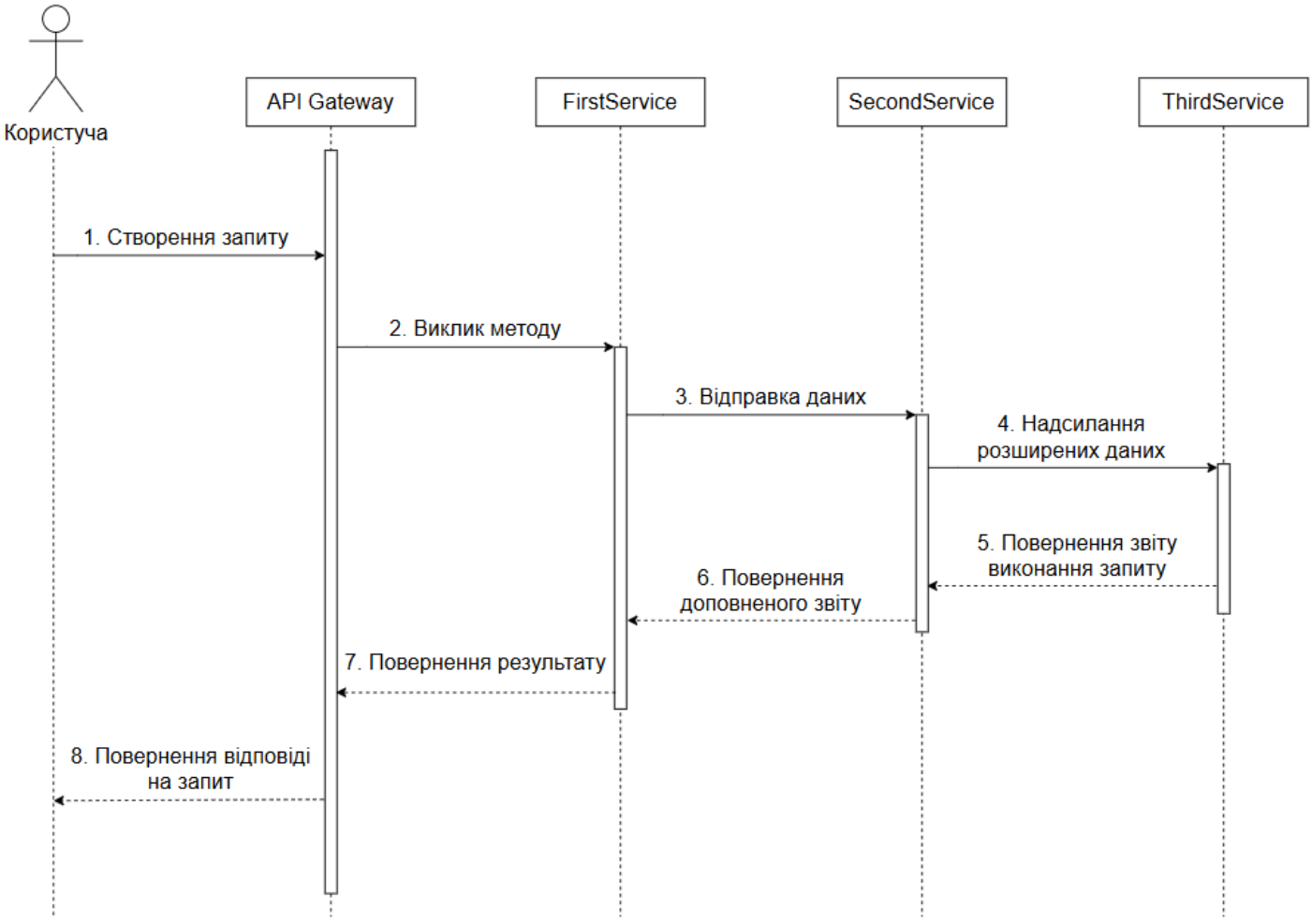

Рисунок 3.5 – Діаграма послідовності

З рисунку можна зрозуміти, що всі запити, що надходять до мікросервісів проходять через API Getaway, що являє собою єдину точку входу, що дозволяє здійснювати балансування навантажень, кешувати відповіді для часто запитуваних ресурсів та виконувати маршрутизацію запитів, що допомагає ізолювати клієнтів від деталей реалізації внутрішньої архітектури.

Окремо варто відмітити, що діаграма відображає процес комунікації при використанні синхронних підходів. Це можна зрозуміти поглянувши на слуги активації. Вони вказують на те, що елемент існує до того моменту, поки не відправить відповідь назад.

Наступним важливим завданням було визначення даних, які будуть передаватись між різними мікросервісами. Для тестування ефективності обміну даними з допомогою різних методів було створено клас ToSendData, який являє собою модель даних, що зберігається в базі даних, та містить інформацію, що необхідно передавати у текстовому форматі та ряд додаткових змінних, що містять інформацію про ці дані та спосіб їхньої передачі між сервісами.

Оскільки в процесі виконання запиту об’єм даних, що передається між різними мікросервісами може сильно змінюватись, тому повертати їх як фінальний результат виконання запиту не є доцільним адже це призвело б до необхідності більш складного контролю та могло б вплинути на результати всіх наявних видів тестування.

Для вирішення даного виклику було передбачено, що кожен мікросервіс буде повертати інформацію про те який спосіб передачі даних було проведено, чи були доставлені дані до кожного з мікросервісів, час відправки та отримання інформації кожним сервісом. Для цього було створено клас ResponeData, який являє собою тип даних для повернення. Ознайомитись із даними, що вони зберігають, можна на рисунку 3.6.

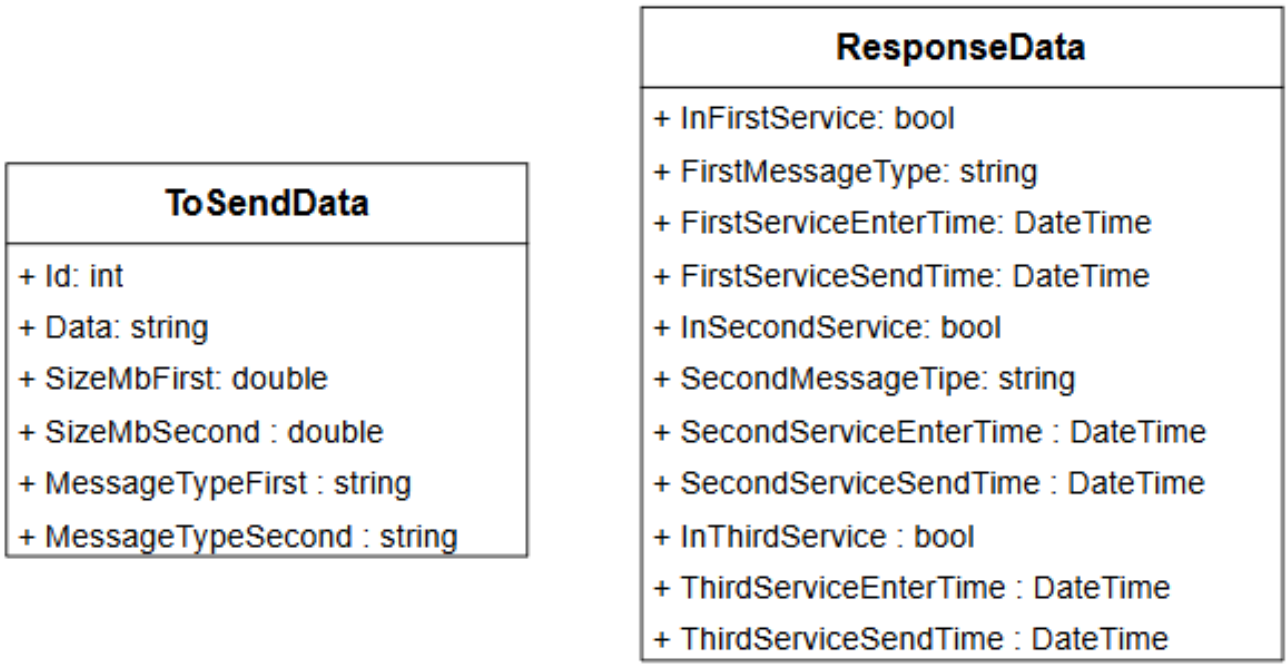

Рисунок 3.6 – Моделі даних

При такій реалізації відбувається тестування швидкості відправки даних заданого об’єму лише в одну сторону, а в результаті повертатиметься лише інформація про виконання запиту дуже малого розміру, що мінімальним чином впливає на швидкість роботи програмного продукту.

3.3 Вибір технологій розробки

На даному етапі, коли всі основні вимоги до програмного забезпечення сформовані, а структура застосунку визначена, необхідно було визначитись із технологіями потрібними для розробки, що дозволять всі необхідні види тестувань з мінімальною кількістю чинників, що впливатимуть на його роботу.

Для розробки програмного забезпечення на основі мікросервісної архітектури найкращим варіантом для даної роботи є використання мови програмування C#. Дана мова характеризується своєю високою продуктивністю та кросплатформеністю, що є ключовими факторами при розробці мікросервісної

архітектури, що може працювати на різних середовищах. Крім того, C# пропонує потужний фреймворк ASP.NET Core, що чудово оптимізований для роботи з HTTP запитами, що дозволить забезпечити високий рівень швидкості та надійності комунікації між сервісами.

Разом із цим, C# підтримує протокол GRPC, що дозволяє досягти високої швидкості передачі даних у системі, а завдяки розвинутій екосистемі готових бібліотек і компонентів, ця мова надає всі необхідні інструменти для ефективної реалізації, тестування та моніторингу мікросервісів. Це сприяє створенню надійної, масштабованої та продуктивної системи, що повністю відповідає всім поставленим вимогам.

На основі даної інформації для синхронної комунікації між мікрсервісами було передбачено використання HTTP та GRPC.

Для реалізації асинхронної комунікації між мікросервісами найкращим варіантом є використання RabbitMQ. Він являється брокером повідомлень, що підтримує обмін даними за допомогою протоколу AMQP, що робить його чудовим рішенням для обміну повідомленнями між сервісами без жорсткої прив’язки між ними та забезпечує достатній рівень надійності та гнучкості .

Для роботи з базами даних було обрано фреймворк Entity Framework Core, що підтримує інтеграцію з багатьма типами баз даних та надає можливості по використанню мови запитів LINQ, а також забезпечує ряд додаткових зручних та ефективних інструментів для роботи з даними.

У зв’язку з тим, що основним завданням даного проекту є тестування методів передачі даних надзвичайно важливим постає проблема зменшення впливу інших чинників на систему, що можуть призводити до сповільнення її роботи. Так як дане дослідження передбачає роботу з відносно великим об’ємом даних при зчитуванні та записі інформації можуть відбуватись досить великі затримки, що можуть негативно впливати на швидкість виконання запитів.

Для вирішення даного недоліка доцільно використовувати технологію InMemory. Вона дозволяє зберігати дані у оперативній пам’яті замість використання дискового простору. Хоча такий підхід не можна ефективно застосувати для

зберігання даних у реальних проектах, проте він чудово підходить для проведення різного виду тестувань. Це пов’язано з тим, що оперативна пам’ять є набагато швидшою у роботі ніж жорсткі диски чи твердотілий накопичувач, що допоможе зменшити вплив аспектів, що пов’язані з роботою з базою даних, на швидкість виконання запитів.

Для забезпечення ефективного розгортання, масштабування та управління мікросервісами необхідно було обрано технології контейнеризації та оркестрації. Серед найбільш ефективних сучасних варіантів було обрано Docker та Kubernetes, як найбільш доцільні до використання.

Docker дозволяє ізолювати кожен мікросервіс у власному контейнері, що забезпечує його незалежність та послідовність роботи в різних середовищах. Це означає, що будь-який мікросервіс може працювати ідентичним чином на локальному комп'ютері розробника, тестовому сервері чи в хмарному середовищі, що спрощує процес розгортання та значно знижує ризики пов’язані із несумісністю. Крім того, використання Docker забезпечує контроль над залежностями та конфігурацією кожного окремого сервісу, що є критично важливим у мікросервісній архітектурі, де кожен компонент системи розвивається та розгортається окремо один від одного.

Kubernetes відповідає за оркестрацію контейнерів, дозволяючи автоматично розподіляти навантаження між ними, масштабувати додаток на основі кількості отриманих запитів та забезпечувати високу доступність системи. Також Kubernetes забезпечує автоматичне відновлення контейнерів, що виходять з ладу не залежно від причин поломки, а також підтримує автоматичне оновлення і розгортання сервісів, що дозволяє мінімізувати час простою ресурсів системи та спрощує підтримку і оновлення програмного продукту.

У контексті даного проекту, поєднання Docker та Kubernetes дозволить не лише спростити розгортання мікросервісів на різних середовищах, а й допоможе у проведенні експериментів із вимірюванням ефективності різних комунікаційних методів при змінних умовах навантаження. Це також дозволить забезпечити достатньо високий рівень гнучкості та відмовостійкості під час тестування

різноманітних сценаріїв, що передбачають роботу із великими об’ємами даних та дозволить налаштувати різноманітні параметри масштабування на основі потреб кожного окремого експерименту.

При використанні Kubernetes в даному проекті для реалізації API Getaway найкращим інструментом є Ingress NGINX. Він являє собою контролер, що приймає вхідні запити та перенаправляє їх до необхідного сервера. З його допомогою можна ефективно управляти трафіком для різних мікросервісів в рамках одного кластеру та балансувати навантаження.

Для проведення різного виду тестувань доцільним є використання k6, що являє собою сучасний інструмент для тестування продуктивності та стійкості до навантажень, що використовується для веб-додатках, API та мікросервісах. З його допомогою будуть проводитись усі необхідні види тестувань та надає всі необхідні метрики для аналізу, що можна зберігати у зручному форматі.

3.4 Висновки

Для написання даного розділу було визначено основні вимоги до програмного продукту з врахування поставлених до реалізації задач. На основі отриманих даних було визначено які задачі має виконувати кожен мікросервіс та яких обмежень потрібно дотримуватись при проектуванні застосунку, що дозволить зменшити вплив різних аспектів системи на результати тестування.

Використовуючи сформовані вимоги та обмеження було проведено розробку структури програмного забезпечення. Для виконання даного завдання було створено декілька діаграм, що демонструють архітектуру програмного продукту при розгортанні з використанням контейнерів, детально розглянуто структуру кожного із передбачених у системі мікросервісів та розроблено діаграму послідовності, що демонструє яким чином мікросервіси можуть обмінюються між собою даними для виконання поставлених задач.

Здійснивши подальший аналіз отриманих даних було визначено перелік основних технологій, що є необхідними для виконання поставлених до реалізації задач та допоможуть провести максимально релевантні тестування.

Для проведення розробки було обрано мову програмування C# з використанням ASP.NET Core. Для взаємодії з базою даних було передбачено використання Entity Framework та технологію InMemory, що дозволять ефективно працювати з даними.

Для розгортання системи на основі контейнерів було обрано Docker у комбінації з Kubernetes з впровадженням Ingress NGINX у ролі API Getaway.

4 РЕАЛІЗАЦІЯ ТА ТЕСТУВАННЯ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ

4.1 Програмна реалізація

Одним із заключних етапів виконання даної роботи була реалізація програмного забезпечення, за допомогою якого проведено всі подальші тестування. На основі обраних технологій було створено три мікросервіси, що будуть обмінюватись інформацією між собою. Відповідно до вимог до кожного з них були підключені наступні пакети:

– Entity Framework Core та ряд додаткових пакетів для спрощення роботи з базами даних;

– база даних InMemory для зберігання інформації в оперативній пам’яті;

– розширення AutoMapper з системою впровадження залежностей для спрощення перетворення об’єктів в межах сервісу;

– Protobuf від компанії Google для обміну даними, що передаються між мікросервісами за допомогою GRPC;

– ряд пакетів, що дозволяють впровадити до проекту GRPC сервісів та клієнтів, а також інструментарій необхідний для компіляції файлів формату proto для визначення сервісів та повідомлень;

– клієнт RabbitMQ, що забезпечує інтерфейс для взаємодії з сервером для відправки та прийому повідомлень.

Варто також розглянути метод для створення даних, що передаються між мікросервісами. Швидкість виконання запитів напряму залежить від швидкості обміну даними між сервісами, а на даний аспект сильно впливає об’єм інформації, що надсилаються.

Таким чином, для можливості тестування різних методів передачі інформації та визначення їхньої ефективності роботи залежно від об’єму даних, було розроблено метод, що створює новий об’єкт класу заданого розміру, який у подальшому може бути використаний для виконання запитів. Ознайомитись із кодом даного методу можна на рисунку 4.1.

```
6 references
public ToSendData CreateNewToSendData(CreateToSendDataDto createToSendDataDto)
{
    int sizeInBytes = (int)createToSendDataDto.SizeMbFirst * 1024 * 1024;

    if (sizeInBytes > int.MaxValue / sizeof(char))
    {
        sizeInBytes = 1000;
    }

    var data = new string('X', sizeInBytes);
    var toSendData = new ToSendData { Data = data,
        SizeMbFirst = createToSendDataDto.SizeMbFirst,
        SizeMbSecond = createToSendDataDto.SizeMbSecond };

    _context.ToSendDatas.Add(toSendData);
    _context.SaveChanges();

    return toSendData;
}
```

Рисунок 4.1 – Метод створення об’єкта

Варто звернути увагу, що у багатьох методах результат може повертатись у форматі JSON. За замовчування у ASP.NET Core HTTP-сервер Kestrel, що є вбудованим та автоматично використовується при запуску додатку, обмеження на розмір тіла запиту встановлено у розмірі 30 мегабайт для всіх типів даних, а подібні обмеження передбачені і для інших серверів, хоча можуть відрізнятись розміром та іншими факторами.

Подібні параметри за замовчуванням надзвичайно сильно обмежують обсяг даних, що можуть передавитися між мікросервісами. Це призводить до того, що у випадках коли розмір відправленого об’єкту перевищує максимально дозволений ліміт запит перестає виконуватись.

Також, у зв’язку з постійним обміном великої кількості даних можуть виникати проблеми коли, очікування відповіді відбувається занадто довго у зв’язку з необхідністю обробити отриману інформацію чи надіслати її далі, що може призвести до зупинення виконання задачі.

Для усунення даних обмежень у кожному проекті було проведено ряд налаштувань, що збільшують максимальний розмір повідомлення та часові рамки очікування для отримання та відправлення їх серверами, що включають GRPC-сервіс, сервери Kestrel, IIS та базові параметри обробки JSON файлів у контролерах. Ознайомитись із встановленням відповідних параметрів можна на рисунку 4.2.

```
builder.Services.AddControllers()
    .AddJsonOptions(options =>
    {
        options.JsonSerializerOptions.MaxDepth = 1000;
        options.JsonSerializerOptions.DefaultBufferSize = 2000 * 1024 * 1024;
    });

builder.Services.Configure<IISServerOptions>(options =>
{
    options.MaxRequestBodySize = 2147483648;
});

builder.Services.Configure<KestrelServerOptions>(options =>
{
    options.Limits.MaxRequestBodySize = 2147483648;
    options.Limits.KeepAliveTimeout = TimeSpan.FromMinutes(2);
    options.Limits.RequestHeadersTimeout = TimeSpan.FromMinutes(2);
});

builder.Services.AddGrpc(options =>
{
    options.MaxReceiveMessageSize = 2000 * 1024 * 1024;
    options.MaxSendMessageSize = 2000 * 1024 * 1024;
});

builder.WebHost.ConfigureKestrel(options =>
{
    options.Limits.Http2.KeepAlivePingDelay = TimeSpan.FromSeconds(60);
    options.Limits.Http2.KeepAlivePingTimeout = TimeSpan.FromMinutes(2);
});


builder.Services.AddHttpClient<HttpSecondServiceDataClient>( client =>
 {
    client.Timeout = TimeSpan.FromMinutes(2);
});
```

Рисунок 4.2 – Параметри для збільшення максимального розміру файлів, часу очікування та відправлення даних

Після завершення всіх попередніх налаштувань мікросервісів можна стверджувати, що етап підготовки завершено і було розпочато створення основного функціоналу застосунку.

Як вже було визначено у попередніх розділах, даний програмний продукт передбачає наявність трьох мікросервісів, а отже, між ними відбуватиметься дві передачі даних. Цього цілком достатньо для перевірки можливості ефективного використання різних методів передачі даних. Таким чином у даному програмно продукті було реалізовано наступні методи для обміну повідомленнями від першого до другого та від другого до третього сервісу:

- HTTP та HTTP;
- HTTP та GRPC;
- HTTP до RabbitMQ;
- RabbitMQ та RabbitMQ;
- RabbitMQ та GRPC;
- GRPC та GRPC.

Варто відмітити, що використання двох ідентичних методів передачі даних між різними сервісами необхідно для порівняння ефективності їхньої роботи як самостійно, так і в комбінації з іншими. Окрім цього у другому мікросервісі існує ідентичний метод як у першому що дозволяє збільшити чи зменшити дані, що використовують для обміну залежно від отриманих параметрів. Це дозволить протестувати різні комбінації обміну інформацією, наприклад, при першій передачі даних відправляється 10 мегабайт, а при другій – 30 мегабайт, чи навпаки.

Для реалізації відповідних комбінацій обміну повідомлень у першому сервісі було створено ряд методів, що зображені на рисунку 4.3, та використовуються для виконання відповідних запитів. Усі вони отримують на вхід дані, що вказують розмір даних, який повинен передаватись між різними мікросервісами, на основі яких буде створюватись відповідний об’єкт. Ознайомитись із повним кодом контролера можна у додатку А.1.

```
[Route("api/[controller]")]
[ApiController]
3 references
public class FirstServiceController : ControllerBase
{
    private readonly AppDbContext _context;
    private readonly ISecondServiceDataClient _secondServiceDataClient;
    private readonly ILogger<FirstServiceController> _logger;
    private readonly IGrpcFirstServiceDataClient _grpcFirstServiceDataClient;
    private readonly IMessageBusClient _messageBusClient;

    0 references
    public FirstServiceController(AppDbContext context,
        ISecondServiceDataClient secondServiceDataClient,
        ILogger<FirstServiceController> logger,
        IGrpcFirstServiceDataClient grpcFirstServiceDataClient,
        IMessageBusClient messageBusClient)
    {
        _context = context;
        _secondServiceDataClient = secondServiceDataClient;
        _logger = logger;
        _grpcFirstServiceDataClient = grpcFirstServiceDataClient;
        _messageBusClient = messageBusClient;
    }

    [HttpPost("HttpToHttpById/{id}")]
    0 references
    public async Task<IActionResult> HttpToHttpById(int id)...

    [HttpPost("HttpToGrpcById/{id}")]
    0 references
    public async Task<IActionResult> HttpToGrpcById(int id)...

    [HttpPost("HttpToBusById/{id}")]
    0 references
    public async Task<IActionResult> HttpToBusById(int id)...

    [HttpPost("GrpcToGrpcById/{id}")]
    0 references
    public async Task<IActionResult> GrpcToGrpcById(int id)
    {
        var toSendData = await _context.ToSendDatas.FirstOrDefaultAsync(lo => lo.Id == id);
```

Рисунок 4.3 – Методи контролера першого сервісу

Першим методом передачі даних, який було вирішено реалізувати в рамках проекту, стала комунікація через протокол HTTP. Він був обраний з кількох ключових причин. Насамперед, він є надзвичайно легким у реалізації, оскільки не потребує використання будь-яких додаткових інструментів, бібліотек чи спеціалізованих пакетів. HTTP давно зарекомендував себе як загальноприйнятий

стандарт для обміну інформацією в середовищі веб-комунікації. Більше того, його широке використання та підтримка забезпечують простоту інтеграції, а також високу сумісність із різними технологіями та платформами.

Для виконання даного завдання у першому та другому міколсеовісах було реалізовано клієнт, що використовується для відправки інформації за допомогою HTTP-запиту до наступного мікросервісу. Його алгоритм роботи можна описати наступним чином:

– відповідно до вмісту запиту та його кінцевого місця призначення встановлюється адреса для відправки запиту, після чого здійснюється перевірка її коректність;

– дані для відправки сформовані у вигляді об’єкта серіалізуються в JSON та встановлюється у потік для читання;

– клієнт виконує відправку відповідного запиту за вказаною адресою та очікує відповідь;

– у випадку успішної відповіді результат десеріалізується та повертається у метод з якого було здійснено запит;

– у разі виникнення помилки інформація про неї заноситься до журналу логів для подальшого аналізу, після чого повертається пусте значення або значення за замовчуванням.

Дані клієнти у першому та другому мікросервісів виконують ідентичні задачі, проте адреса відправки запиту у першому сервісі обирається залежно від того який подальший метод передачі даних буде застосовано. У випадку другого сервісу, клієнт має лише одну адресу, що використовується для передачі даних до третього мікросервісу.

Ознайомитись із фрагментом коду даного клієнта можна звернувшись до рисунка 4.4. Весь код клієнта розміщено у додатку А.2.

```
4 references
public async Task<ResponseData> SendDataToSecondService(ToSendData toSendData)
{
    string? secondServiceUrl = null;
    switch (toSendData.MessageTypeSecond)
    {
        case "HTTP":
            secondServiceUrl = _configuration["HttpToHttp"];
            break;
        case "GRPC":
            secondServiceUrl = _configuration["HttpToGRPC"];
            break;
        case "BUS":
            secondServiceUrl = _configuration["HttpToBus"];
            break;
    }

    if (string.IsNullOrEmpty(secondServiceUrl))
    {
        _logger.LogError("URL для SecondService не заданий у конфігурації.");
        return null;
    }

    using (var memoryStream = new MemoryStream())
    {
        await JsonSerializer.SerializeAsync(memoryStream, toSendData);

        memoryStream.Seek(0, SeekOrigin.Begin);

        var streamContent = new StreamContent(memoryStream)
        {
            Headers =
            {
                ContentType = new System.Net.Http.Headers.MediaTypeHeaderValue("application/json")
            }
        };

        try
        {
            var response = await _httpClient.PostAsync(secondServiceUrl, streamContent);

            if (response.IsSuccessStatusCode)
            {

                var responseContent = await response.Content.ReadAsStringAsync();
                var toSendDataResponse = JsonSerializer.Deserialize<ResponseData>(
                    responseContent,
                    new JsonSerializerOptions { PropertyNameCaseInsensitive = true });
```

Рисунок 4.4 – Фрагмент коду HTTP-клієнта

Наступним реалізованим методом передачі даних був GRPC. Даний фреймворк заснований на використанні протоколу HTTP/2 та використовує мову серіалізації даних Protobuf як основний формат передачі даних, що є альтернативою JSON. Його особливістю є те, що серіалізація та десеріалізація відбувається досить швидко, а дані зберігаються у бінарному форматі, що займає менше місця ніж будь-які інші текстові формати.

Проте, важливо врахувати те, QRPC використовує протокол HTTP/2, що може конфліктувати у випадках використання HTTP з’єднання. Для вирішення даної

проблеми у конфігурації Kestrel необхідно передбачити кінцеві точки для HTTP та GRPC з’єднання з різними адресами.

Першим етапом реалізації комунікації за допомого GRPC є розробка структури запитів та відповідей у відповідному файлі Protobuf, де описуються відповідні дані, методи та типи повідомлень. Ознайомитись із створеним файлом даного формату, що використовується всіма сервісами можна ознайомитись на рисунку 4.5.

```
syntax = "proto3";

option csharp_namespace = "Services.Protos";

import "google/protobuf/timestamp.proto";

service GrpcFirstService{
    rpc SendDataToSecondService (ToSendData) returns (ResponseData);
}

service GrpcSecondService{
     rpc SendDataToThirdService (ToSendData) returns (ResponseData);
}

message ToSendData {
    int32 id = 1;
    string data = 2;
    double sizeMbFirst = 3;
    double sizeMbSecond = 4;
    string messageTypeFirst = 5;
    string messageTypeSecond = 6;
}

message ResponseData {
    bool inFirstService = 1;
    string firstMessageType = 2;
    google.protobuf.Timestamp firstServiceEnterTime = 3;
    google.protobuf.Timestamp firstServiceSendTime = 4;
    bool inSecondService = 5;
    string secondMessageType = 6;
    google.protobuf.Timestamp secondServiceEnterTime = 7;
    google.protobuf.Timestamp secondServiceSendTime = 8;
    bool inThirdService = 9;
    google.protobuf.Timestamp thirdServiceEnterTime = 10;
    google.protobuf.Timestamp thirdServiceSendTime = 11;
}
```

Рисунок 4.5 – Налаштування різних адрес для різних протоколів передачі даних

В даному файлі наявні два методи, що використовуються для обробки відповіді на отриманий запит, та дві моделі даних, що є ідентичними до вже існуючих моделей. В даному контексті варто відмітити, що хоч моделі і є ідентичними, проте вони не можуть автоматично бути перетворені один в одного.

Для вирішення даної проблеми було реалізовано автомапер з детальними налаштуваннями, що дозволяє автоматично переносити дані між об’єктами без необхідності ручного написання коду.

Далі, на основі даного файлу та з використанням відповідних інструментів GRPC було згенеровано код, що містить клієнтську та серверну логіку на основі яких було створено відповідний клієнт та сервер.

Клієнтська частина представляє собою клас, що містить мінімум один метод, що визначає адресу для відправки запиту та його вміст. Після цього запит та його вміст можуть бути відправлені до серверної частини, а отриманий результат відповідним чином оброблений та повернений.

Серверна частина працює подібним чином, проте вона відразу обробляє дані після отримання запиту з клієнтської частини та повертає назад отриманий результат. Ознайомитись із фрагментом коду сервера GRPC для другого мікросервісу можна на рисунку 4.6 чи звернувшись до додатку А.4.

Варто відмітити, що для обробки та передачі даних до наступного мікросервісу через GRPC було створено окремий клас, що відповідає за даний функціонал. Це пов’язано з тим, що контролери у ASP.NET Core використовують HTTP запити, тоді як GRPC використовує HTTP/2, що може призводити до різних конфліктів при спробі їхньої взаємодії та може призвести до зменшення ефективності обробки запиту.

Останнім методом передачі даних, що був передбачений до розробки в рамках даного проекту є реалізація асинхронного типу обміну даними через брокер повідомлення. У попередніх розділах було визначено, для її реалізації використовуватиметься RabbitMQ. Перед початком розробки клієнтської та

серверної частини RabbitMQ було встановлено та запущено у Docker, що дозволило відразу протестувати його роботу.

```
4 references
public class GrpcSecondService : GrpcFirstService.GrpcFirstServiceBase
{
    private readonly IMapper _mapper;
    private readonly ILogger<GrpcSecondService> _logger;
    private readonly ISecondServiceProcessor _processor;

    0 references
    public GrpcSecondService(IMapper mapper, ILogger<GrpcSecondService> logger, ISecondServiceProcessor processor)
    {
        _mapper = mapper;
        _logger = logger;
        _processor = processor;
    }

    3 references
    public override async Task<ResponseData> SendDataToSecondService(ToSendData request, ServerCallContext context)
    {
        ResponseData res = new ResponseData
        {
            InSecondService = true
        };

        var memoryStream = new MemoryStream(request.ToByteArray());
        var deserializedRequest = Services.Protos.ToSendData.Parser.ParseFrom(memoryStream);

        Models.ToSendData toSendData = _mapper.Map<Models.ToSendData>(deserializedRequest);

        var result = await _processor.ProcessDataAsync(toSendData);
        var mappedResult = _mapper.Map<ResponseData>(result);

        return res;
    }
```

Рисунок 4.6 – Фрагментом коду сервера GRPC для другого мікросервісу

Для реалізації клієнтської частини було створено клас, що використовується для відправлення запиту до брокера. У конструкторі класу відбувається налаштування підключення до брокера повідомлень, що включає створення каналу та обробника подій.

Також у даному класі реалізовано ряд методів. Перший метод приймає дані для відправки, серіалізує їх у формат JSON та за допомогою іншого методу відправляє до брокера повідомлень. Для випадків, коли результат має повертатись, використовується слухач. Він очікує відповідь з черги брокера, а при отримані результату десеріалізує його. З фрагментом коду клієнта для RabbitMQ можна ознайомитись на рисунку 4.7 чи у додатку А.5.

```
3 references
public Task<ResponseData> SendDataToThirdService(ToSendData toSendData)
{
    if (_connection == null || _channel == null)
    {
        _logger.LogError("--> RabbitMQ connection is not initialized.");
        throw new InvalidOperationException("RabbitMQ: Connection is not initialized.");
    }

    var message = JsonSerializer.Serialize(toSendData);
    var body = Encoding.UTF8.GetBytes(message);

    if (_connection.IsOpen)
    {
        _channel.BasicPublish(
            exchange: "trigger",
            routingKey: "",
            basicProperties: _props,
            body: body);

        _logger.LogInformation("Message sent");

        return _responseTcs.Task.ContinueWith(t =>
        {
            if (t.IsFaulted)
            {
                _logger.LogError("Error in receiving response");
                throw new Exception("Error in receiving response");
            }

            return JsonSerializer.Deserialize<ResponseData>(t.Result);
        });
    }
    else
    {
        _logger.LogWarning("--> RabbitMQ connection is closed, not sending.");
        throw new InvalidOperationException("RabbitMQ connection is closed.");
    }
}
```

Рисунок 4.7 – Фрагмент коду клієнта для RabbitMQ

Сервіс для брокера повідомлень виглядає подібним чином. У конструкторі він налаштовує підключення до RabbitMQ, а також створює чергу та прив’язує її до обміну. Замість методу для передачі даних іншому сервісу він може повертати до брокера відповідь.

Також варто врахувати, що при спробі передачі запиту до брокера повідомлень та отримання помилки у випадках його поломки чи неможливості прийняти запит виконання автоматично переходить до передачі інформації через

HTTP протокол. Таким чином мікросервіс зможе продовжити виконувати поставлене завдання та можна оцінити можливість комбінації різних методів передачі даних та доцільність їхнього комбінування у подібних ситуаціях.

На даному етапі можна стверджувати, що розробка основного функціоналу програмного продукту завершена і необхідно провести розгортання системи..

Для виконання даного завдання до кожного із мікросервісів було додано підтримку Docker. Відразу після цього до кожного з них було підключено Dockerfile, що являє собою текстовий файл з інструкціями для створення образу. Приклад даного файлу для першого мікросервісу зображено на рисунку 4.8.

```
FROM mcr.microsoft.com/dotnet/sdk:8.0 AS build-env
WORKDIR /app

COPY *.csproj ./
RUN dotnet restore

COPY . ./
RUN dotnet publish -c Release -o out

FROM mcr.microsoft.com/dotnet/aspnet:8.0
WORKDIR /app
COPY --from=build-env /app/out .
RUN dotnet --info
ENTRYPOINT ["dotnet", "FirstService.dll"]
```

Рисунок 4.8 – Dockerfile для першого мікросервісу

Для розгортання контейнерів за допомогою Kubernetes було створено ряд конфігураційних файлів, що визначає яким чином та де мають бути запущені застосунку з їхніми даними та встановлює доступ до кластерів та рядом додаткової

інформації, наприклад, порти чи тип сервісу. Ознайомитись даним файлом налаштувань можна на рисунку 4.9.

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: firstservice-depl
spec:
  replicas: 1
  selector:
    matchLabels:
      app: firstservice
  template:
    metadata:
      labels:
        app: firstservice
    spec:
      containers:
        - name: firstservice
          image: sergiy1111/firstservice:latest
---
apiVersion: v1
kind: Service
metadata:
  name: firstservice-clusterip-srv
spec:
  type: ClusterIP
  selector:
    app: firstservice
  ports:
  - name: firstservice
    protocol: TCP
    port: 8080
    targetPort: 8080
```

Рисунок 4.9 – Файл конфігурації для першого мікросервісу

На даному етапі можна стверджувати, що програмний продукт є розробленим і розгорнутим у кластері Kubernetes та у подальшому може використовуватись для проведення тестування та дослідження.

4.2 Вибір інструментів тестування

Для перевірки коректної роботи програмного забезпечення доцільним є використання unit-тестів. У зв’язку з тим, що перший мікросервіс залучений до виконання всіх завдань у системі, основну увагу при тестуванні необхідно надати йому. Таким чином перевіривши основні методи першого мікросервісу можна дізнатись чи вірно працюють всі компоненти системи, включно з іншими сервісами, а у випадках появи помилок можна швидко оцінити на якому із етапів виникають проблеми та оперативно їх вирішити.

Через використання платформи .NET для розробки програмного забезпечення доцільним є використання фреймворку NUnit, що дозволяє створювати і виконувати автоматизовані тестові сценарії.

У зв’язку з тим, що даний програмний застосунок не передбачає наявність інтерфейсу для взаємодії з мікросервісами необхідно було передбачити створення запитів до мікросервісів для тестування коректності їхньої роботи. Для вирішення даного завдання було використано Insomnia, що дозволяє провести тестування різних сценаріїв взаємодії з сервісами, перевірити коректність роботи запитів та результати їхньої роботи.

Основним завданням даної роботи є тестування ефективності різних методів передачі даних та перевірку їхньої стійкості до навантажень. Одним із найкращих інструментів, що може вирішити дане завдання, є k6, що являє собою інструмент для тестування API та веб-сервісів. Основою його перевагою є те, що він дозволяє створювати, запускати на аналізувати різного види сценарії, а по їхньому завершенню надає детальні звіти продуктивності, що включають різні метрики та показники.

4.3 Розробка тестових сценаріїв

Першим етапом тестування розробленого програмного забезпечення була розробка юніт тестів. Для цього до поточного рішення було підключено проект відповідного типу, додано посилання на перший мікросервіс та створено ряд тестів, що дозволять перевірити коректність роботи основних методів контролера. Ознайомитись із результатами даного тестування можна на рисунку 4.10.

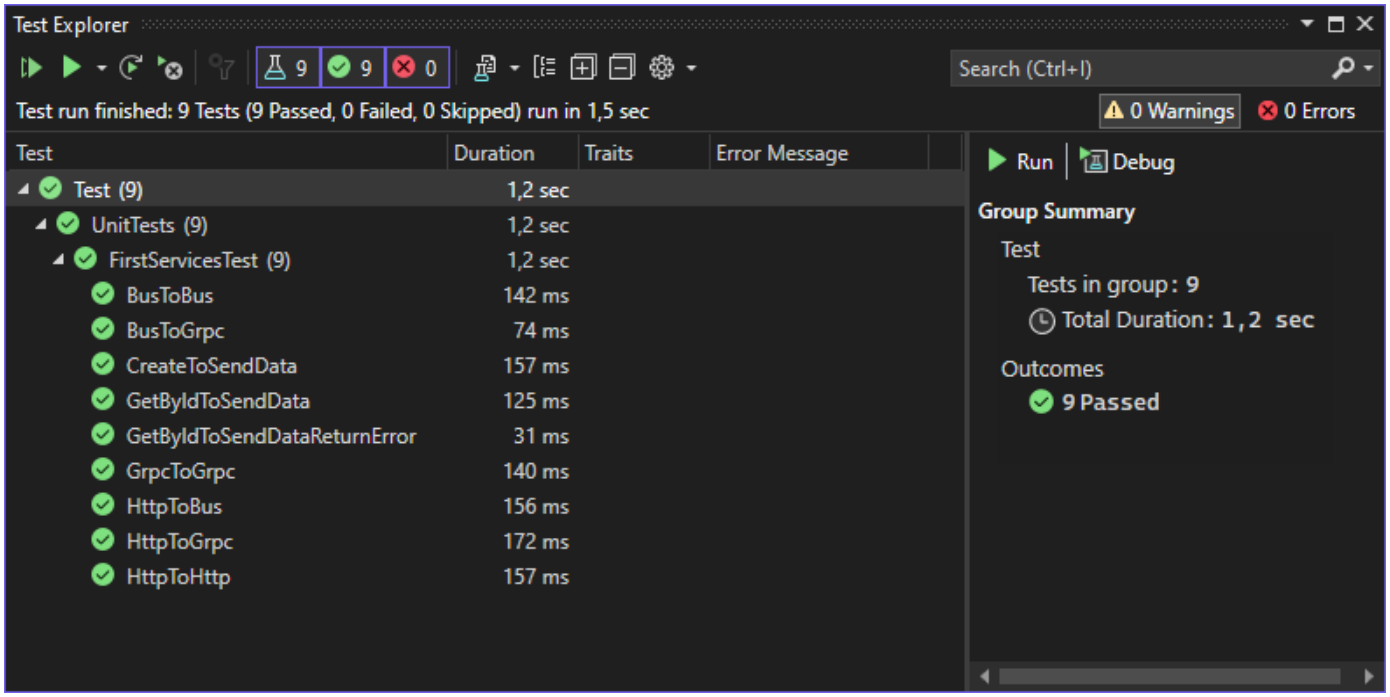

Рисунок 4.10 – Юніт тести

З даного рисунку можна зрозуміти, що всі тестові сценарії були виконані, а отже методи працюють коректно. Проте, такий вид тестування не може в повній мірі гарантувати вірність розгорнутої системи за допомогою Docker та Kubernetes, наприклад, мікросервіс можуть працювати коректно, проте доступ до них може бути неможливим через некоректну роботу Api Getaway. Для цього з допомогою Insomnia було здійснено запити до першого мікросервісу при локальному розгортанні,

напряму через адресу в Kubernetes та через Api Getaway. Приклад виконання запиту зображено на рисунку 4.11.

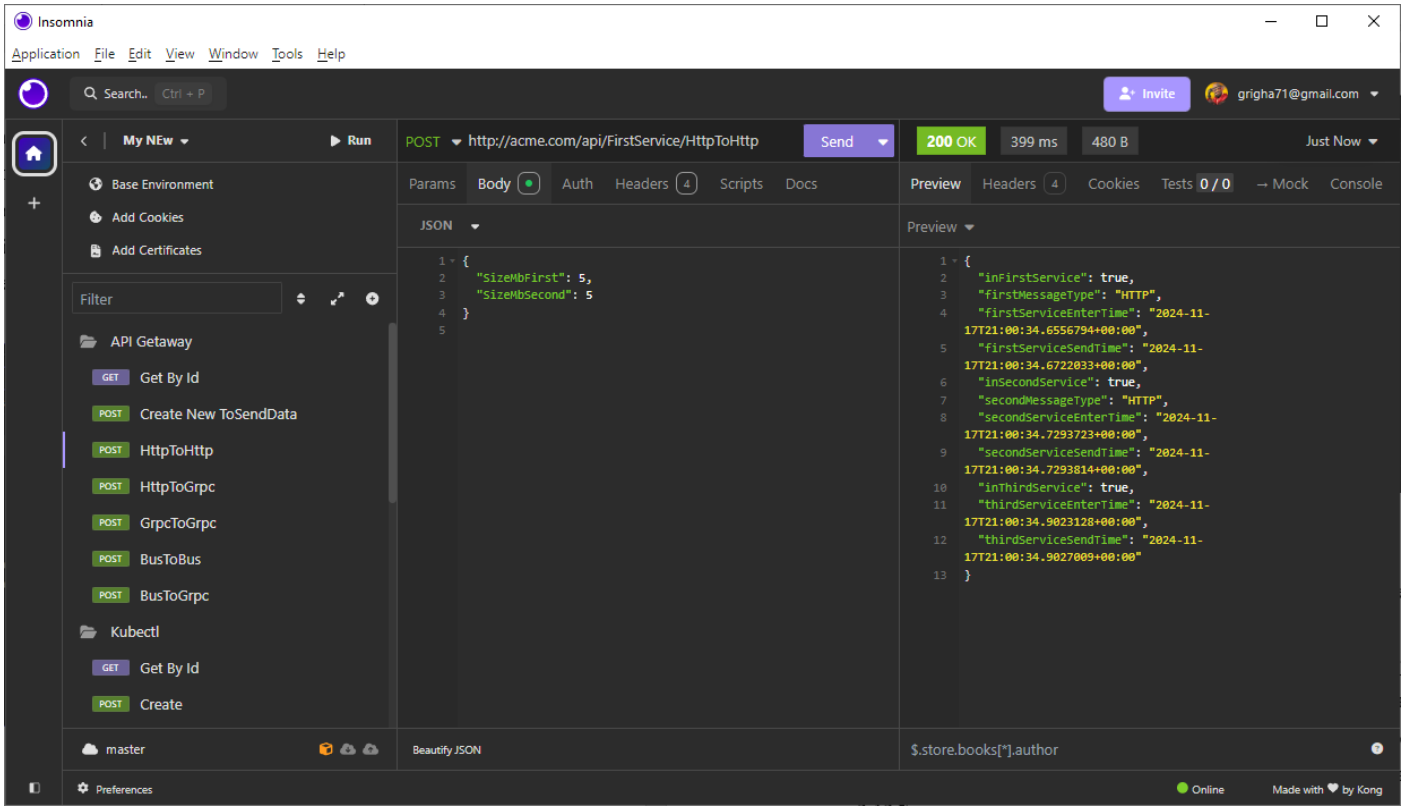

Рисунок 4.11 – Перевірка Api Getaway в Insomnia

Наступний крок складається з декількох типів тестувань, що об'єднанні використання k6.

Першим видом тестуванням із даного списку є оцінка того, яким чином різний спосіб розгортання впливає на ефективність роботи системи. Для виконання даного завдання на мові програмування JavaScript було створено сценарій тестування, що збирає ряд метрик та виконує певну кількість запитів. Фрагмент даного коду можна побачити на рисунку 4.12.

В результаті виконання даного фрагмента коду буде отримано ряд метрик, що дозволять оцінити ефективність роботи кожного протестованого методу, що у подальшому були збережені у базу даних та за потреби можуть бути застосовані для формування графіків чи інших матеріалів з метою кращого розуміння отриманих результатів.

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

let httpErrors = new Rate('http_errors');
let httpReqs = new Counter('http_reqs');

export let options = {
    vus: 1,
    iterations: 100,
    duration: '25m',
};

export default function () {
    let url = 'http://acme.com/api/FirstService/BusToGrpcById/7';

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

    let res = http.post(url, "", params);

    httpReqs.add(1);

    let waitingTime = res.timings.waiting;
    let connectingTime = res.timings.connecting;
    let receivingTime = res.timings.receiving;
    let sendingTime = res.timings.sending;
```

Рисунок 4.12 – Фрагмент коду JavaScript для тестування ефективності роботи методів

З даного рисунка видно, що в коді визначається адреса, за яким буде виконано запит, та початкові дані у вигляді розмірів об'єктів, що будуть передаватись в

процесі виконання запиту. Задана адреса вказує на API Getaway, що буде звертатись до потрібного мікросервісу та передаватиме всі дані. За потреби дана адреса може бути змінена на іншу, що було виконано при проведені тестування системи при локальному способі розгортання.

Даний сценарій тестування передбачав, що кожен із методів передачі даних буде запущений мінімум декілька разів для оцінки ефективності його роботи при обміні різною інформацією.

Під час кожного тесту передбачається, що об’єм даних, яким обмінюються мікросервіси буде поступово змінюватись у сторону збільшення і відповідно дорівнювати ста кілобайтам, одному, п’яти, десяти, двадцяти та тридцяти мегабайтам.

По завершенню проведення тестування всі отримані результати будуть збережені до бази даних. На їхній основі у подальшому було сформовано ряд таблиць, що містять лише основну інформацію з результатами, що дозволить оцінити ефективність роботи кожного методу.

Варто відмітити, що під час тестування автоматично збирається велика кількість даних, наприклад, час відправки та отримання запиту, розмір даних, що були відправлені у запиті та отримані як відповідь, кількість успішних та провалених тестів, час встановлення з’єднання та інші характеристики, що не є доцільними у використанні при проведенні оцінки отриманих даних. Приклад результатів тестування з великою кількістю додаткових параметрів, що були виведені у консолі можна побачити на рисунку 4.13.

З даних причин для оцінки ефективності роботи різних методів таблиці повинні включати наступні параметри:

– назва всіх наявних методів для яких було проведено відповідний тип тестування;

– розмір даних, що використовувався під час тестування перевірки передачі інформації між мікросервісами;

– середній час очікування відповіді;

– мінімальний час відповіді;

– максимальний час відповіді;

– час в межах якого виконуються дев’яносто п’ять відсотків всіх відправлених запитів;

– час виконання сотні запитів.

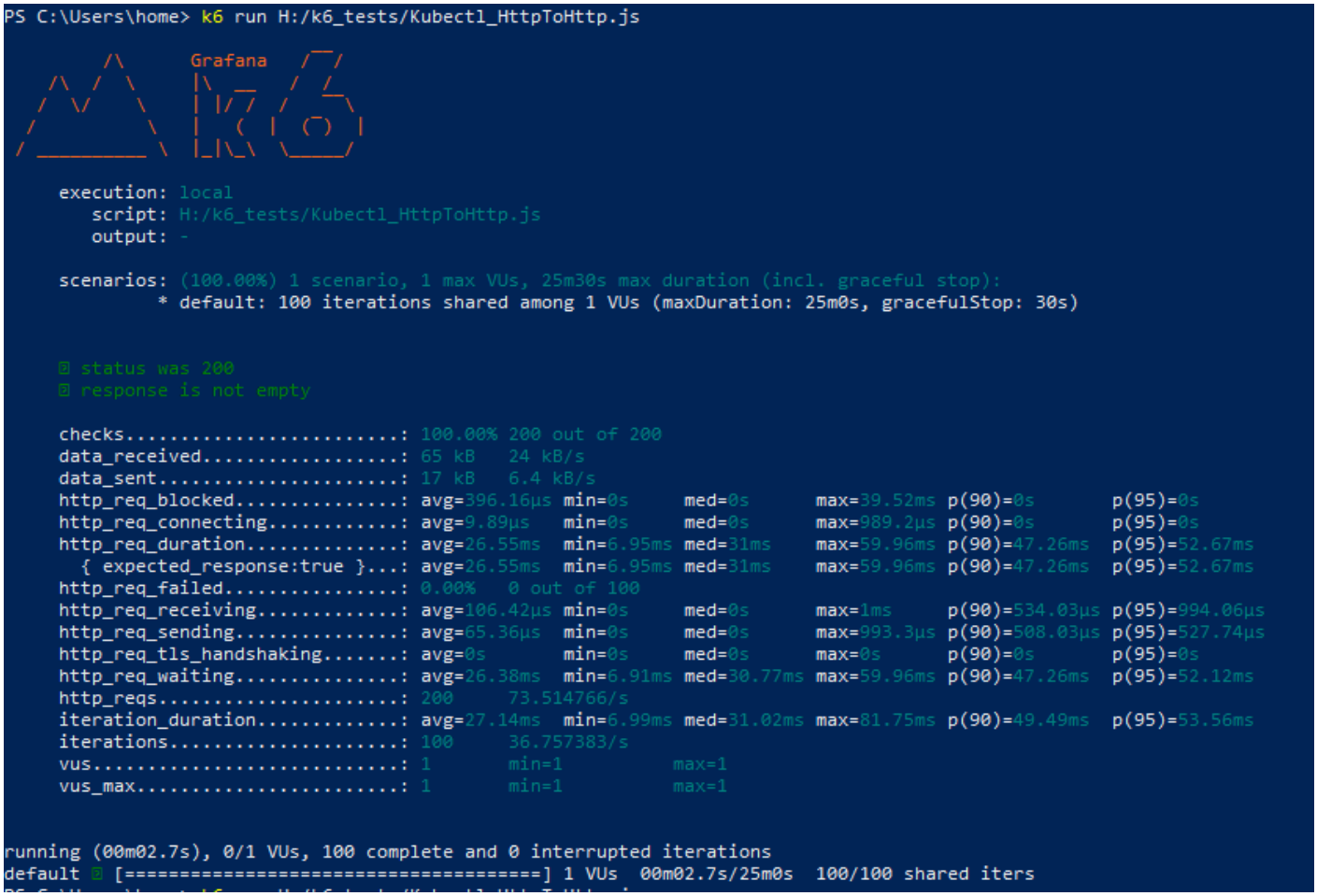

Рисунок 4.13 – Приклад результатів тестування, що виводяться у консоль по завершенню відповідного процесу

Наступним видом тестування, що було передбачено провести в рамках виконання даної роботи, є оцінка швидкості виконання запитів за допомогою комбінації різних методів передачі даних та з врахуванням розміру передаваємого об'єкту між сервісами.

Для виконання даного завдання використовувався ідентичний код написаний на мові JavaScript, проте дані, що передаються відповідному методу змінювались

відповідно до потреб, наприклад, під час проведення перевірки впливу великих та малих об’ємів даних на роботу різних комбінацій методів.

Даний вид тестування має повністю ідентичні умови проведення як і попередній, проте його особливістю є те, що дані між мікросервісами відбувається обмін даними різного об’єму.

Тобто, між першим та другим сервісом здійснюється обмін малим об’ємом даних, а між другим та третім – великий. Такий підхід до проведення тестування допоможе визначити найбільш ефективні комбінації методів з врахуванням розміру інформації та може допомогти визначити потенційні слабкі місця у їхній роботі та реалізації, що можуть впливати на різні аспекти програмного застосунку та на різні комбіновані методи.

4.4 Аналіз результатів

Перші тести, що були проведені в рамках даної роботи передбачали проведення дослідження потенційного впливу способу розгортання системи на роботу мікросервісної архітектури. Для цього розроблені мікросервіси були розгорнуті двічі – локально на комп’ютері та у контейнерах за допомогою використання Docker та Kubernetes.

Для їхнього тестування було використано раніше розроблений тестовий сценарій у вигляді JavaScript коду та здійснено його запуск за допомогою k6 з метою проведення тестування та збору всіх необхідних для оцінки метрик.

Для початку було вирішено провести тестування швидкості виконання запитів для розгорнутої системи за допомогою локального методу. По завершенню тестувань всі отримані показники у вигляді результатів були збережені до бази даних після чого всі необхідні для оцінки параметри були занесені до таблиці. Ознайомитись із отриманими результатами можна у таблиці 4.1.

Таблиця 4.1 – Результати тестування локального розгортання

	Розмір даних	Середній час відповіді	Мінімальний час відповіді	Максимальний час відповіді	95-й перцентиль	Час виконання запиту
HttpToHttp	0,1 Мб	21,22мс	6,59мс	49,07мс	38,19мс	2,2с
	1 Мб	30,83мс	14,42мс	71,2мс	66,53мс	3,1с
	5 Мб	76,69мс	46,33мс	139,68мс	114,95мс	7,8с
	10 Мб	151,64мс	89,6мс	1,16с	220,37мс	15,2с
	20 Мб	266,16мс	166,67мс	489,04мс	380,01мс	26,7с
	30 Мб	373,4мс	234,27мс	1,19с	474,69мс	37,4с
HttpToGrpc	0,1 Мб	17,02мс	6,74мс	43,05мс	37,82мс	1,8с
	1 Мб	28,53мс	14,89мс	76,49мс	63,45мс	2,9с
	5 Мб	73,84мс	52,66мс	125,88мс	100,58мс	7,5с
	10 Мб	144,09мс	111,71мс	211,41мс	188,46мс	14,5с
	20 Мб	275,15мс	211,61мс	413,83мс	352,27мс	27,6с
	30 Мб	400,32мс	307,26мс	603,65мс	479,39хв	40,1с
HttpToBus	0,1 Мб	16,82мс	5,22мс	66,44мс	42,27мс	1,8с
	1 Мб	29,64мс	12мс	100,65мс	58,58мс	3,0с
	5 Мб	63,29мс	39,04мс	157,79мс	96,33мс	6,4с
	10 Мб	126,01мс	84,53мс	444,01мс	180,09мс	12,7с
	20 Мб	241,86мс	163,23мс	484,79мс	346,45мс	24,3с
	30 Мб	375,34мс	235,68мс	484,15мс	551,32мс	37,6с
BusToGrpc	0,1 Мб	7,09мс	1,14мс	32,11мс	27,09мс	0,8с
	1 Мб	14,63мс	4,99мс	86,95мс	50,59мс	1,5с
	5 Мб	30,62мс	17,5мс	79,99мс	53,16мс	3,1с
	10 Мб	53,63мс	32,17мс	115,38мс	91,42мс	5,5с
	20 Мб	115,35мс	62,1мс	270,3мс	200,25мс	11,6с
	30 Мб	143,24мс	89,76мс	301,21мс	228,7мс	14,4с
BusToBus	0,1 Мб	6,04мс	1,99мс	31,68мс	24,55мс	0,7с
	1 Мб	12,11мс	5мс	54мс	39,67мс	1,3с
	5 Мб	26,85мс	17,16мс	65,16мс	45,76мс	2,8с
	10 Мб	51,08мс	31,99мс	109,78мс	84,35мс	5,2с
	20 Мб	104,08мс	63,3мс	212,36мс	165,72мс	10,5с
	30 Мб	152,67мс	91,56мс	427,48мс	245,33мс	15,4с
GrpcToGrpc	0,1 Мб	10,02мс	5,93мс	35,63мс	26,27мс	1,1с
	1 Мб	23,82мс	17мс	73,8мс	44,73мс	2,5с
	5 Мб	79,89мс	60,06мс	135,53мс	105,36мс	8,1с
	10 Мб	156,78мс	123,22мс	344,66мс	217,4мс	15,8с
	20 Мб	296,4мс	248,29мс	398,24мс	337,95мс	39,7с
	30 Мб	426,67мс	357,7мс	586,64мс	505,9мс	42,8с

Відразу після завершення локального тестування було проведено розгортання системи за допомогою Docker та Kubernetes та проведено ідентичне тестування. Ознайомитись із його результатами можна у таблиці 4.2.

Таблиця 4.2 – Результати тестування розгортання з допомогою Docker та Kubernetes

	Розмір даних	Середній час відповіді	Мінімальний час відповіді	Максимальний час відповіді	95-й перцентиль	Час виконання сотні запитів
HttpToHttp	0,1 Мб	27,15мс	7,17мс	103,6мс	72,69мс	2,8с
	1 Мб	35,25мс	10,88мс	94,74мс	59,33мс	3,6с
	5 Мб	75,3мс	45,86мс	131,13мс	104,2мс	7,6с
	10 Мб	133,7мс	91,24мс	199,3мс	175,96мс	13,4с
	20 Мб	269,99мс	175,4мс	476,26мс	326,78мс	27,1с
	30 Мб	403,18мс	287,59мс	739,15мс	582,73хв	40,2с
HttpToGrpc	0,1 Мб	20,44мс	7мс	57,35мс	43,3мс	2,1с
	1 Мб	31,39мс	13,99мс	72,92мс	54,3мс	3,2с
	5 Мб	77,89мс	56,15мс	186,35мс	107,34мс	7,9с
	10 Мб	153,6мс	108,12мс	265,44мс	201,07мс	15,5с
	20 Мб	284,78мс	210,44мс	447,23мс	393,29мс	28,6с
	30 Мб	417,79мс	290,32мс	639,96мс	528,83хв	41,8с
HttpToBus	0,1 Мб	18,43мс	5,25мс	141,78мс	42,37мс	1,9с
	1 Мб	28,79мс	12,99мс	73,73мс	55,39мс	3,0с
	5 Мб	65,98мс	38,35мс	154,96мс	112,54мс	6,7с
	10 Мб	119,65хв	79,57мс	204,51мс	178,9мс	12,0с
	20 Мб	243,72мс	144,72мс	412,56мс	371,08мс	24,5с
	30 Мб	378,17мс	257,72мс	672,53мс	533,19хв	37,9с
BusToGrpc	0,1 Мб	7,38мс	2,99мс	48,11мс	27,55мс	0,8с
	1 Мб	12,79мс	1,66мс	55,73мс	40,47мс	1,4с
	5 Мб	29,6мс	18,13мс	100,79мс	53,34мс	3,0с
	10 Мб	55,85мс	32,39мс	131,28мс	110,62мс	5,7с
	20 Мб	107,97мс	59,21мс	259,66мс	165,27мс	10,9с
	30 Мб	148,65мс	87,96мс	339,73мс	248,9мс	15,0с
BusToBus	0,1 Мб	7,01мс	3,21мс	34,33мс	25,64мс	0,8с
	1 Мб	13,07мс	4,77мс	49,79мс	40,47мс	1,4с
	5 Мб	30,7мс	17,52мс	76,11мс	66,08мс	3,2с
	10 Мб	51,56мс	28,48мс	115,68мс	88,75мс	5,2с
	20 Мб	102,61мс	102,61мс	211,7мс	135,91мс	10,4с
	30 Мб	143,89мс	90,47мс	457,06мс	209,57мс	14,5с
GrpcToGrpc	0,1 Мб	10,98мс	4,25мс	44,49мс	29,69мс	1,2с
	1 Мб	25,01мс	16,54мс	55,01мс	46,28мс	2,6с
	5 Мб	80,67мс	60,94мс	145,34мс	109,4мс	8,2с
	10 Мб	151,89мс	121,69мс	201,3мс	185,99мс	15,3с
	20 Мб	304,3мс	236,25мс	912,25мс	353,2мс	30,5с
	30 Мб	418,09мс	351,96мс	543,19мс	494,94хв	41,9с

Після завершення тестування першим кроком виконання даного завдання було порівняння ефективності роботи ідентичних методів при різному способі розгортання застосунків. Для цього було створено таблицю для порівняння результатів 4.3.

Таблиця 4.3 – Порівняння отриманих результатів

	Розмір даних	Середній час відповіді (локальне тестування) (мс)	Середній час відповіді (розгортання з допомогою Docker та Kuberners) (мс)	Часова різни між першим та другим значенням (мс)	Відсоткова різниця (%)
HttpToHttp	0,1 Мб	21,22	27,15	5,93	27,95
	1 Мб	30,83	35,25	4,42	14,34
	5 Мб	76,69	75,3	-1,39	-1,81
	10 Мб	151,64	133,7	-17,94	-11,83
	20 Мб	266,16	269,99	3,83	1,44
	30 Мб	373,4	403,18	29,78	7,98
HttpToGrpc	0,1 Мб	17,02	20,44	3,42	20,09
	1 Мб	28,53	31,39	2,86	10,02
	5 Мб	73,84	77,89	4,05	5,48
	10 Мб	144,09	153,6	9,51	6,60
	20 Мб	275,15	284,78	9,63	3,50
	30 Мб	400,32	417,79	17,47	4,36
HttpToBus	0,1 Мб	16,82	18,43	1,61	9,57
	1 Мб	29,64	28,79	-0,85	-2,87
	5 Мб	63,29	65,98	2,69	4,25
	10 Мб	126,01	119,65	-6,36	-5,05
	20 Мб	241,86	243,72	1,86	0,77
	30 Мб	375,34	378,17	2,83	0,75
BusToGrpc	0,1 Мб	7,09	7,38	0,29	4,09
	1 Мб	14,63	12,79	-1,84	-12,58
	5 Мб	30,62	29,6	-1,02	-3,33
	10 Мб	53,63	55,85	2,22	4,14
	20 Мб	115,35	107,97	-7,38	-6,40
	30 Мб	143,24	148,65	5,41	3,78
BusToBus	0,1 Мб	6,04	7,01	0,97	16,06
	1 Мб	12,11	13,07	0,96	7,93
	5 Мб	26,85	30,7	3,85	14,34
	10 Мб	51,08	51,56	0,48	0,94
	20 Мб	104,08	102,61	-1,47	-1,41
	30 Мб	152,67	143,89	-8,78	-5,75
GrpcToGrpc	0,1 Мб	10,02	10,98	0,96	9,58
	1 Мб	23,82	25,01	1,19	5,00
	5 Мб	79,89	80,67	0,78	0,98
	10 Мб	156,78	151,89	-4,89	-3,12
	20 Мб	296,4	304,3	7,9	2,67
	30 Мб	426,67	418,09	-8,58	-2,01

З даної таблиці можна зрозуміти, що в певних випадках виникає достатньо велика розбіжність у результатах, проте середня відсоткова різниця ефективності методів приблизно дорівнює 3.23%, що можна оцінити як відхилення у розрахунках.

Для проведення більш детального порівняння було створено ряд діаграм, що порівнюють ефективність роботи кожного методу при різних варіантах розгортання та враховуючи розмір даних, що передавався між мікросервісами, із середнім часом відповіді на запит. Ознайомитись із створеними діаграмами можна звернувшись до рисунку 4.14.

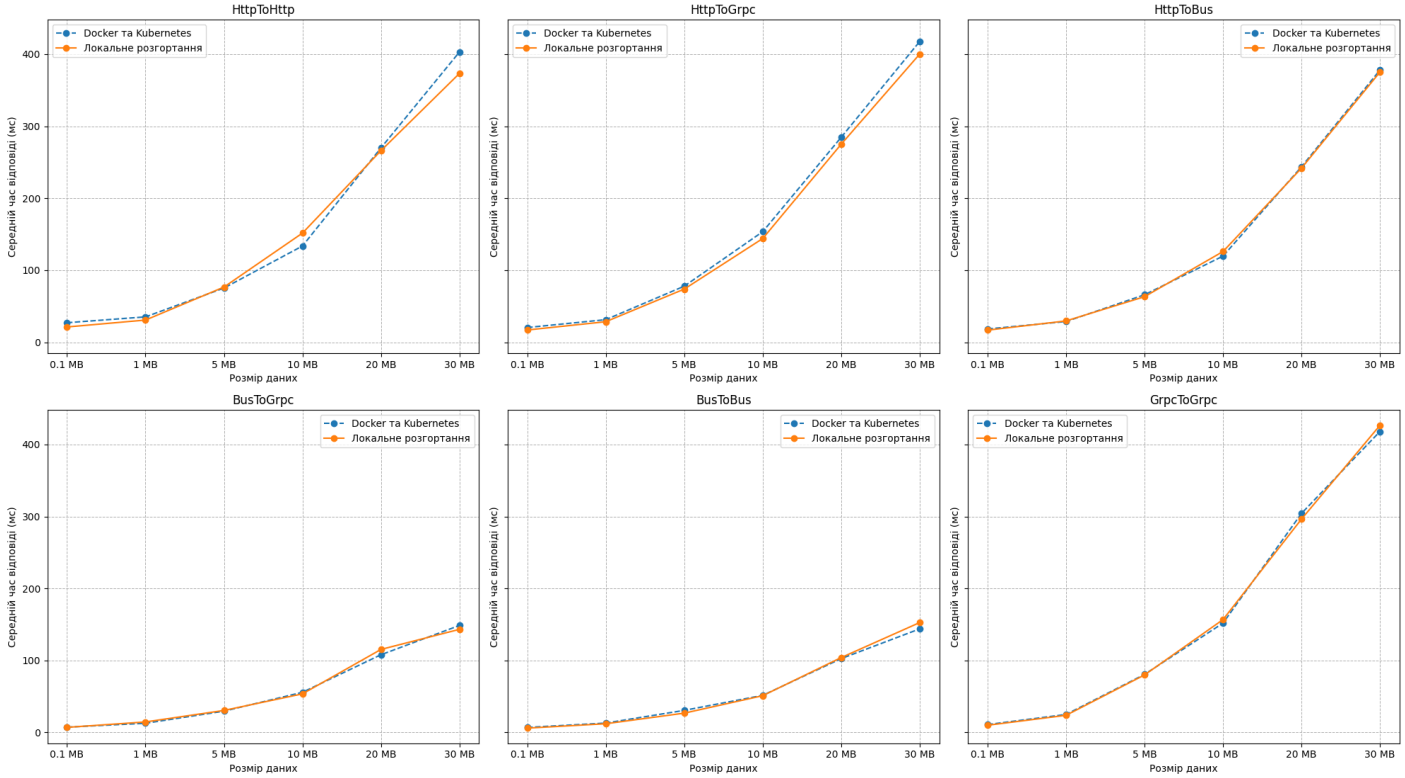

Рисунок 4.14 – Порівняння середнього часу виконання запитів відповідно до способу розгортання системи

Проаналізувавши отриманий результат можна зрозуміти, що різниця швидкість виконання запитів при використанні різних способів розгортання є мінімальною та в більшості випадках коливаються на невелике значення в більшу та меншу сторону, що може вказувати на наявність мінімального відхилень чи початок

виконання певних фонових процесів, що змогли вплинути на результат. Із шести перевірених комбінацій способів комунікації в чотирьох випадках локальний метод розгортання системи виявився швидшим при порівнянні результатів тестування з об’ємом даних у 50 мегабайт. У випадках, коли для тестування використовувались менші об’єми інформації різниця середнього часу відповіді на запит є мінімальною.

На основі отриманих результатів було зроблено висновок, що при коректному налаштуванні способу розгортання та достатній кількості ресурсів у вигляді оперативної пам’яті, потужності процесора і дискового простору програмне забезпечення, що розроблене на базі мікросервісної архітектури з використанням різних шляхів комунікації може працювати приблизно з ідентичною ефективністю при використанні різного методу розгортання.

Наступним завданням, що необхідно було виконати в рамках даного тестування є детальний аналіз результатів з метою оцінки ефективності роботи різних комбінацій методів комунікацій.

У зв’язку з тим, що результати обох тестувань є подібними результати оцінки роботи різних методів було вирішено розглянути на прикладі розгортання з використанням Docker та Kubernetes. Такий вибір пов'язаний з тим, що даний підхід використовується частіше при розгортанні мікросервісної архітектури, хоча його використання може поставити перед розробниками додаткові виклики, що можуть впливати на швидкість комунікації мікросервісів. Таким чином всі подальші тестування були проведені з розгорнутою системою на основі контейнерів.

З метою проведення більш детального аналізу отриманих результатів було вирішено створити декілька додаткових графіків, що у зручному та зрозумілому вигляді зможуть продемонструвати ефективність роботи методів.

Першим із них став графік, що зображає як змінюється швидкість виконання запиту для кожного методу на основі даних про середній час виконання запиту та з врахуванням розміру даних, що використовувався під час тестування. Ознайомитись із отриманим графіком можна звернувшись до рисунку 4.15.

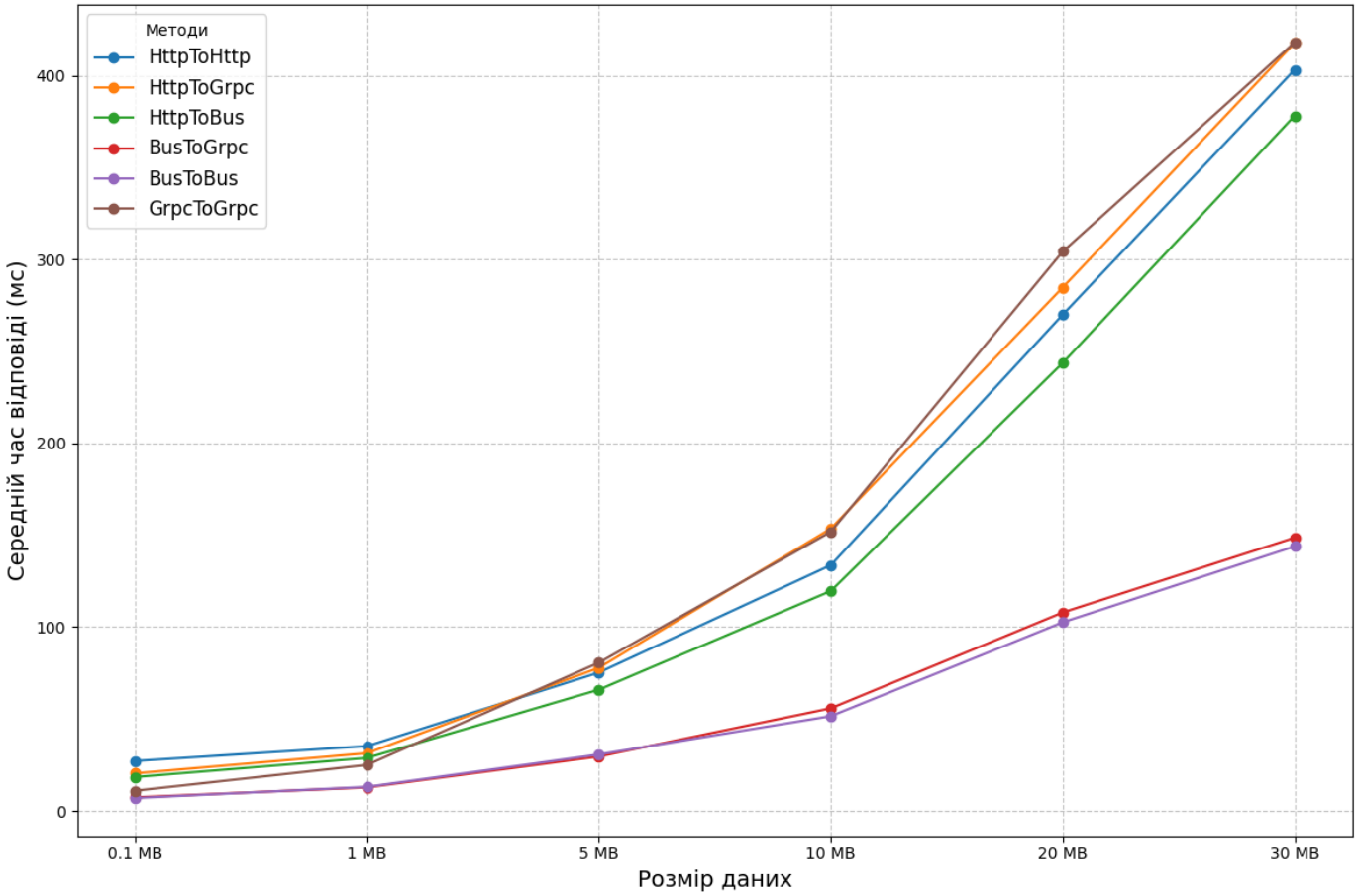

Рисунок 4.15 – Швидкість виконання запитів для методів комунікацій залежно від об’єму обмінюваних даних при розгортанні з допомогою Docker та Kubernetes

З даного рисунку видно, що більшість методів є досить стабільними, а середній час відповіді на запит плавно збільшується відповідно до збільшення об’єму даних з якими вони працюють.

Серед всіх наведених методів найбільше виділяються два способи комунікації, а саме подвійне використання брокера повідомлень та його комбінація з GRPC. У порівнянні з іншими підходами вони мають найменший час очікування відповіді, при збільшенні розміру даних час очікування росте не так стрімко, а найменша різниця між їхніми часовими показниками відрізняється майже у 2,5 разів.

Такі характеристики вказують на те, що дані комбінації методів комунікації можуть ефективно використовуватись не залежно від об’єму даних з якими вони працюють, що робить їх універсальними для багатьох випадків.

Проте, необхідно звернути увагу на те, що брокер повідомлень є обмеженим по кількості доступних ресурсів, а при надмірному збільшені об’ємів інформації він може бути перевантажений та вийти з ладу чи зменшити ефективність виконання поставлених задач.

Якщо ж розглядати інші методи, то можна зробити висновок, що при роботі з даними до 10 мегабайт вони можуть досить ефективно виконувати запити, проте із подальшим збільшенням ефективність може сильно зменшуватись.

На наступному кроці виконання задачі доцільним є проведення детального аналізу того, які методи є більш ефективними у порівнянні з іншими відповідно до кожного об’єму даних для яких проводилось тестування. Для цього було вирішено розробити стовпчикову діаграму, що відображає ефективність роботи кожного метода у відсотковому значені, де за 100% береться значення методу із найбільшими затримками. Ознайомитись із отриманим графіком можна на рисунку 4.16.

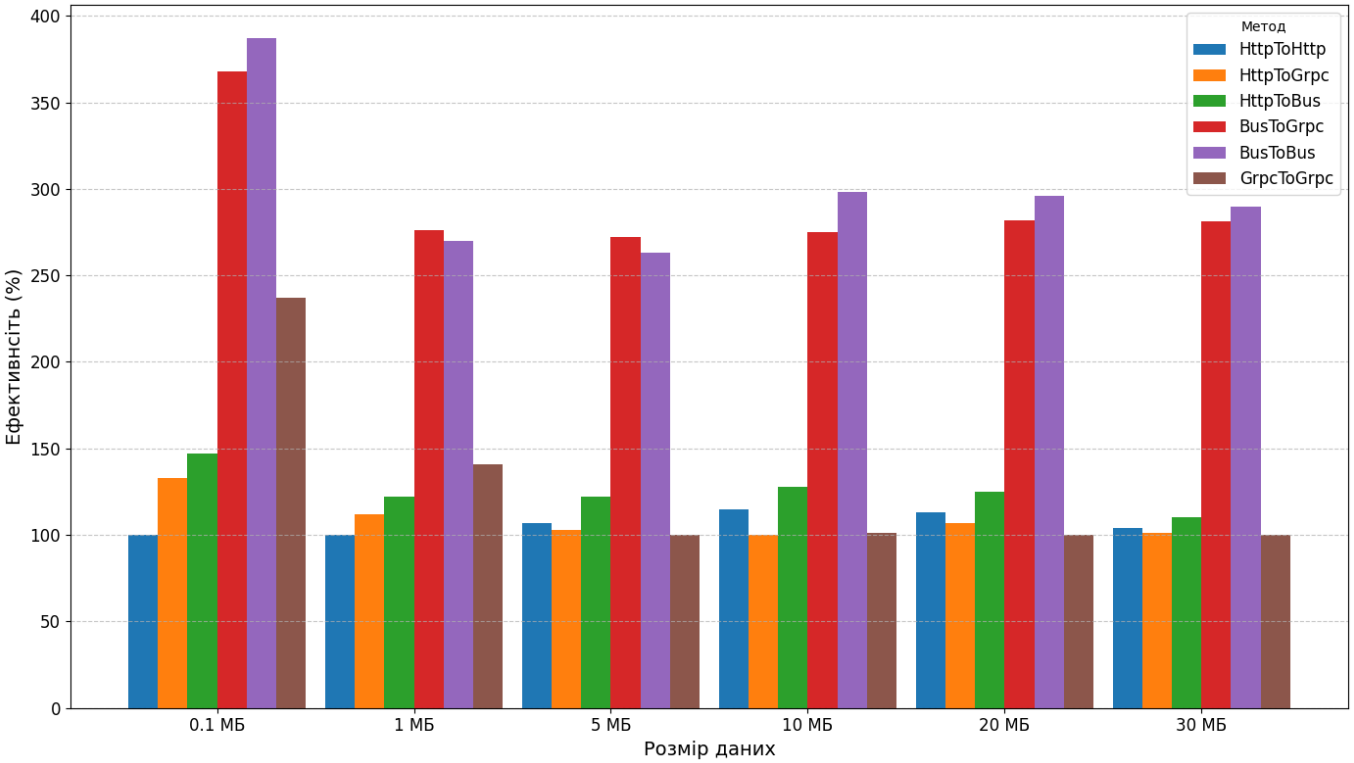

Рисунок 4.16 – Порівняння ефективності роботи методів комунікацій при розгортанні з допомогою Docker та Kubernetes

Отримані результати тестування демонструють, що серед всіх перевірених методів найбільш ефективними виявились комбінації брокера з GRPC та подвійне використання брокера. Не залежно від об’єму даних, що використовувався під час тестування, вони продемонстрували найменший час очікування відповіді. Якщо порівнювати їх із найменш швидкою комбінацією методів, то їхня швидкість роботи коливалась від 270% до 387%.

Першочерговим чинником такого результату є те, що RabbitMQ є асинхронним, а отже він продовжує виконання поставлених завдань без необхідності очікувати негайну відповідь на запит. Також він використовує достатньо швидкий протокол передачі даних, що спеціально розроблений для обміну повідомленнями та постійно відкриває TCP-з’єднання між відправником та отримувачем, що є більш ефективним ніж постійне встановлення та закриття з’єднання, що відбувається при обміні повідомленнями через HTTP.

При роботі з даними малих розмірів досить непогані результати продемонструвала комунікація на основі GRPC. Для найменшого об’єму даних різниця у часі середньої відповіді на запит становила не більше ніж 4 мілісекунди, що є досить малою розбіжністю якщо порівнювати з іншими методами, проте при обміні інформації від п’яти мегабайт його швидкість роботи падає одних із найменших показників.

Такий результат можна пояснити тим, що подвійне використання GRPC призводить до необхідності проведення додаткової серіалізації та десеріалізації даних займають більше часу при роботі з великими даними. Також варто пам’ятати, що при передачі великою кількості інформації потоки можуть блокуватись, оскільки TCP/IP працює на основі сегментів, що вимагають підтвердження доставки.

Всі інші методи виявились менш ефективними. До них відноситься подвійне використання протоколу HTTP, його поєднання з брокером повідомлень та GRPC. Першим фактором, що призвів до такого результату є більші накладні витрати при роботі з HTTP. Формат запиту та відповіді передбачає наявність метаінформацію у вигляді заголовків, що збільшує об’єм даних. Також перша версія даного протоколу

передбачає створення нового TCP-з’єднання для кожного запиту, що призводить до додаткових затримок.

Крім того HTTP не є оптимізованим для роботи з великими об’ємами даних, а тому в більшості випадках дані спочатку зберігаються у пам’ять, без використання інструментів стиснення за замовчуванням, а потім передаються, а враховуючи його синхронну природу очікування відповіді може зайняти більше часу.

Якщо ж розглядати використання HTTP із брокером, то таке поєднання буде швидшим приблизно у 2 рази при використання брокера як першого методу комунікації, що пов’язано з особливостями їхньої роботи, що є актуальним для всіх подібних комбінацій. У випадку GRPC така комунікація призводить до використання двох різних версій HTTP, що призводить до появи затримок.

Отже, на основі здійсненого аналізу можна зробити висновок, що при обміні ідентичними об’ємами даних найбільш швидкими виявились комбінації, що двічі використовують брокер повідомлень RabbitMQ та його комбінація з GRPC, що ефективно працюють як з великими, так і з малими даними.

При роботі з малими розмірами інформації досить ефективним виявилось використання GRPC, проте при збільшені об’єму інформації середня швидкість виконання запитів сильно знижується.

Всі методи, що використовують протокол HTTP та його комбінацію з іншими виявились менш ефективними, проте при зміні порядку виконання запитів при роботі з брокером ефективність обміну даних може значно збільшуватись, що дозволяє використовувати його у певних випадках.

Наступним проведеним видом тестування була оцінка ефективності роботи різних методів з використанням різних об’ємів даних в рамках виконання одного запиту. Для виконання даного завдання було проведено два тестування на основі яких була зібрана вся необхідна для оцінки інформація. Вони передбачали, що перший мікросервіс передаватиме до другого один об’єм даних після чого він збільшуватиметься чи зменшуватиметься, що залежить від параметрів об’єкта, та передається далі до третього мікросервісу. Ознайомитись із результатами першого тестування можна у таблиці 4.4.

Таблиця 4.4 – Перше тестування оцінки швидкості роботи різних методів комунікації при передачі різного об’єму даних

	Розмір даних (Мб)	Середній час відповіді	Мінімальний час відповіді	Максимальний час відповіді	95-й перцентиль	Час виконання сотні запитів
HttpToHttp	0.1 -> 1	30,81мс	12,18мс	124,71мс	89,46мс	3,02с
	1 -> 5	54,33мс	31,66мс	142,92мс	95,38мс	5,5с
	5 -> 10	92,29мс	60,84мс	153,57мс	126,13мс	9,3с
	10 -> 20	184,59мс	123,64мс	514,33мс	242,83мс	18,5с
	0.1 -> 20	126,92мс	86,96мс	180,83мс	164,15мс	12,8с
HttpToGrpc	0.1 -> 1	26,72мс	13,99мс	86,66мс	64,34мс	2,07с
	1 -> 5	59,69мс	37,4мс	139,66мс	82,64мс	6,1с
	5 -> 10	113,34мс	80,53мс	164,29мс	140,51мс	11,4с
	10 -> 20	202,97мс	156,48мс	258,95мс	235,74мс	20,4с
	0.1 -> 20	160,56мс	126,02мс	234,62мс	206,88мс	16,1с
HttpToBus	0.1 -> 1	19.98ms	9.91ms	80.38ms	61.75ms	2.03s
	1 -> 5	42.57ms	27.63ms	85.04ms	67.87ms	4.9s
	5 -> 10	81.18ms	61.19ms	135.28ms	121.42ms	8.6s
	10 -> 20	155.64ms	123.53ms	247.2ms	219.7ms	17.3s
	0.1 -> 20	109.99ms	73.23ms	236.44ms	176.16ms	12.0s
BusToGrpc	0.1 -> 1	12,15мс	5мс	46,39мс	37,42мс	1,3с
	1 -> 5	27,61мс	16,68мс	68,29мс	48,32мс	2,8с
	5 -> 10	51,6мс	35,3мс	110,36мс	86,24мс	5,2с
	10 -> 20	109,67мс	60,81мс	352,71мс	176,5мс	11,1с
	0.1 -> 20	103,85мс	61,5мс	218,36мс	168,34мс	10,5с
BusToBus	0.1 -> 1	7,19мс	2,98мс	37,83мс	26,33мс	0,8с
	1 -> 5	11,92мс	5мс	56,55мс	36,62мс	1,3с
	5 -> 10	26,92мс	16,91мс	70,75мс	43,56мс	2,08с
	10 -> 20	52,85мс	33,93мс	122,94мс	97,55мс	5,04с
	0.1 -> 20	47,5мс	29,14мс	103,87мс	108,07мс	4.4сс
GrpcToGrpc	0.1 -> 1	11,56мс	6,97мс	40,63мс	32,34мс	1,2с
	1 -> 5	24,08мс	15,48мс	47,67мс	42,75мс	2,5с
	5 -> 10	83,41мс	65,01мс	124,63мс	105,3мс	8,4с
	10 -> 20	153,13мс	119,84мс	199,83мс	175,36мс	15,4с
	0.1 -> 20	103,77мс	81,4мс	158,03мс	121,78мс	12,5с

З даної таблиці можна побачити, що для кожної комбінації методів було проведено по п’ять тестів. Значення перших чотирьох змінюються плавно, а останній передбачає різкий стрибок від малого об’єму до великого.

Далі було проведено друге тестування, що передбачало зміну розміру обмінюваних даних у сторонній бік. Всі результати тестування були занесені до таблиці 4.5.

Таблиця 4.5 – Друге тестування оцінки швидкості роботи різних методів комунікації при передачі різного об’єму даних

	Розмір даних (Мб)	Середній час відповіді	Мінімальний час відповіді	Максимальний час відповіді	95-й перцентиль	Час виконання сотні запитів
HttpToHttp	1 -> 0.1	29,24мс	11,89мс	581,7µс	45,12мс	3,0с
	5 -> 1	52,11мс	31,2мс	126,41мс	94,33мс	5,3с
	10 -> 5	91,67мс	62,99мс	162,76мс	132,16мс	9,2с
	20 ->10	187,36мс	137,02мс	280,78мс	249,98хв	18,8с
	20 -> 0.1	132,7мс	85,45мс	250,24мс	182,62мс	13,3с
HttpToGrpc	1 -> 0.1	22,58мс	10,99мс	49,53мс	41,15мс	2,3с
	5 -> 1	48,49мс	34,37мс	99,51мс	71,07мс	4,9с
	10 -> 5	104,14мс	76,43мс	153,57мс	139,15мс	10,5с
	20 ->10	197,62мс	140,1мс	311,98мс	253,97хв	19,8с
	20 -> 0.1	125,1мс	80,01мс	207,67мс	173,67мс	12,6с
HttpToBus	1 -> 0.1	25,01мс	8,99мс	113,88мс	47,5мс	2,4с
	5 -> 1	51,05мс	26,77мс	99,97мс	81,09мс	4,9с
	10 -> 5	104,5мс	62,2мс	166,81мс	137,26мс	9,3с
	20 ->10	191,2мс	105,79мс	301,92мс	257,79мс	17,9с
	20 -> 0.1	134,5мс	67,06мс	279,97мс	174,91мс	11,6с
BusToGrpc	1 -> 0.1	7,74мс	2,99мс	68,27мс	26,99мс	0,8с
	5 -> 1	12,85мс	6мс	53,26мс	34,14мс	1,4с
	10 -> 5	27,77мс	18мс	101,17мс	46,86мс	2,9с
	20 ->10	50,74мс	33,41мс	110,16мс	88,12мс	5,2с
	20 -> 0.1	43,06мс	23,61мс	103,54мс	104,78мс	4,2сс
BusToBus	1 -> 0.1	8,34мс	3,26мс	39,67мс	28,05мс	0,9с
	5 -> 1	12,49мс	6,32мс	58,18мс	38,34мс	1,4с
	10 -> 5	25,76мс	15,51мс	64,96мс	41,74мс	2,01с
	20 ->10	55,64мс	36,12мс	141,56с	95,83мс	5,14с
	20 -> 0.1	38,93мс	19,7мс	81,04мс	54,76мс	3,43с
GrpcToGrpc	1 -> 0.1	13,28мс	8,04мс	45,3мс	37,51мс	1,4с
	5 -> 1	21,72мс	12,09мс	43,62мс	38,4мс	2,3с
	10 -> 5	79,18мс	54,86мс	131,81мс	111,34мс	8,2с
	20 ->10	158,67мс	124,19мс	205,96мс	181,58мс	15,6с
	20 -> 0.1	109,91мс	91,72мс	164,28мс	135,72мс	13,0с

Ознайомившись із отриманими результатами було зроблено висновок, що результати тестування подвійних методів є подібними і варіюються в межах відхилень. Таким чином ефективність їхньої роботи залежить лише від розміру даних з якими вони взаємодіють, а їхній середній час виконання запитів є подібними до попередніх тестувань. З даних причин можна вважати, що комбінації даних методів комунікації не потребують детального аналізу, а основну увагу варто акцентувати на інші комбінації.

Для початку було вирішено проаналізувати вплив обмінюваних даних при одночасному застосування HTTP та GRPC. Для цього було створено графік порівняння середнього часу відповіді для результатів отриманих під час тестування. Ознайомитись з графіком можна на рисунку 4.17.

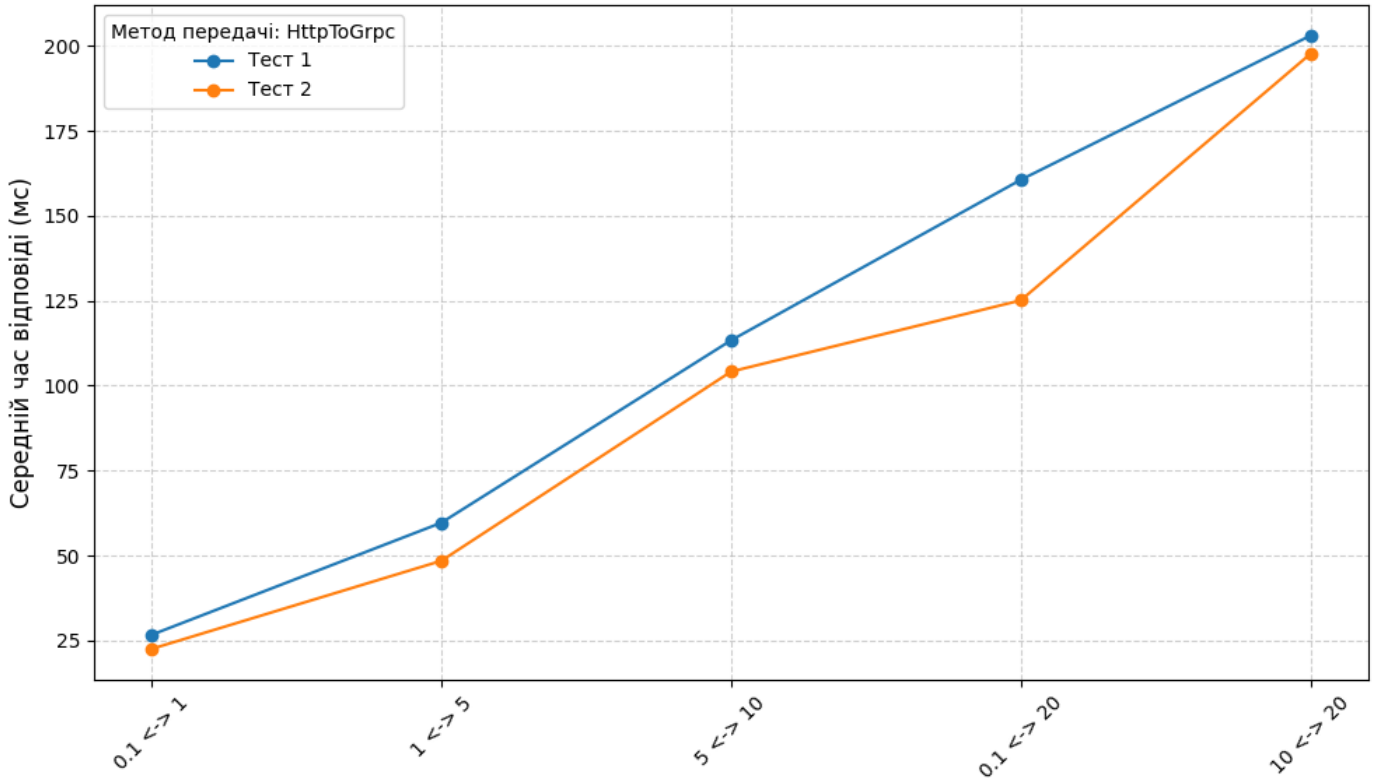

Рисунок 4.17 – Порівняння ефективності передачі даних через HTTP та GRPC при передачі різних об’ємів даних

На даному рисунку можна побачити, що середній час відповіді на запит в обох тестуваннях є близьким, хоча на ньому і присутнє невелике відхилення. Даний

результат є очікуваним, адже при тестуванні даної комбінації комунікації при обміні ідентичними розмірами даних було помічено, при роздільному використанні HTTP та GRPC їхні результати є подібними.

Наступний графік було вирішено реалізувати для порівняння ефективності роботи брокера повідомлень та GRPC. Ознайомитись із отриманим результатом можна на рисунку 4.18.

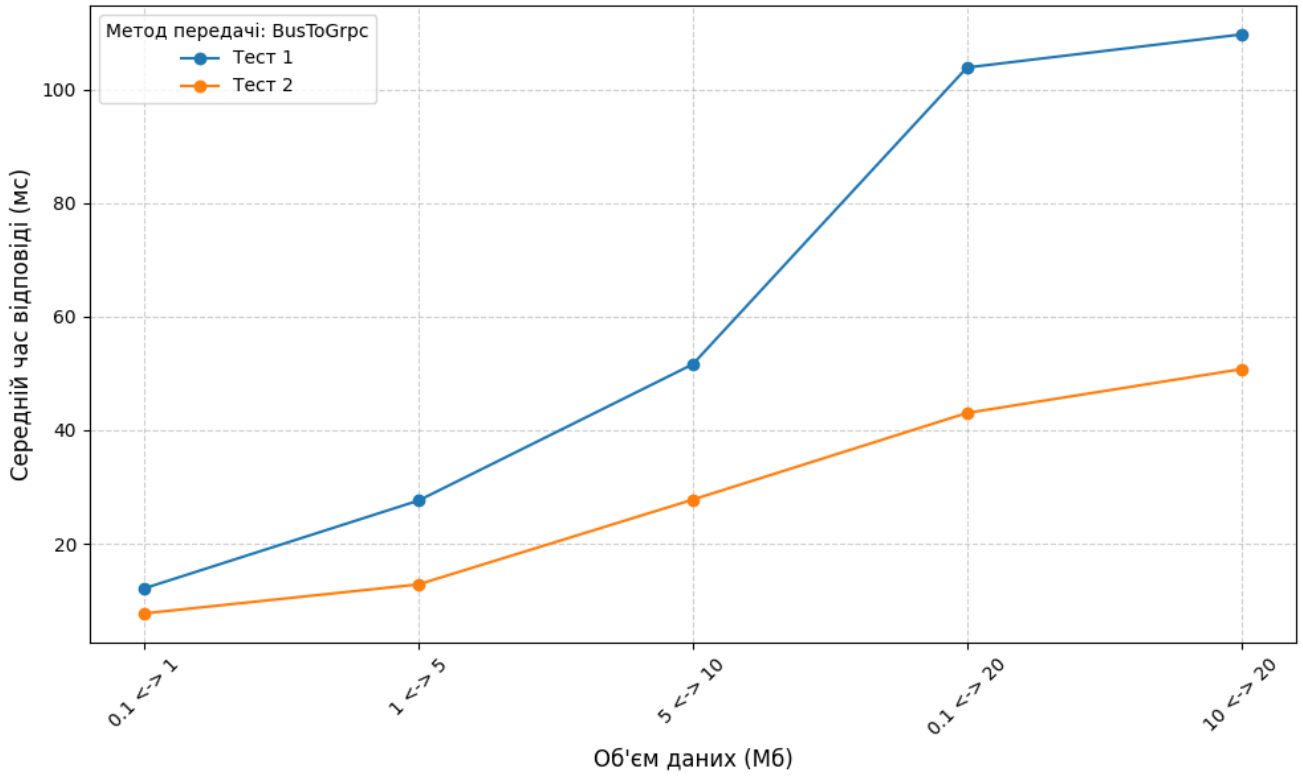

Рисунок 4.18 – Порівняння ефективності передачі даних через брокер та GRPC при передачі різних об’ємів даних

На відміну від попереднього рисунку на даному графіку чітко видно розрив у ефективності роботи методів комунікації залежно від різного об’єму даних, що вони передавали. У максимальній точці різниця дорівнює близько п’ятдесяти мілісекундам. Провівши більш детальний аналіз результатів було встановлено, що у випадках коли брокер повідомлення працює з більшим об’ємом даних, а GRPC з меншим, то швидкість виконання запитів збільшується. Це пов’язано з тим, що

GRPC є менш ефективним при роботі з великими розмірами інформації, проте більш ефективний у роботі з малими об’ємами, що вже було встановлено раніше.

Останньою комбінацією методів було вирішено зробити HTTP з брокером повідомлень та додатково їх обернений варіант. Ознайомитись із розробленим графіком можна на рисунку 4.19.

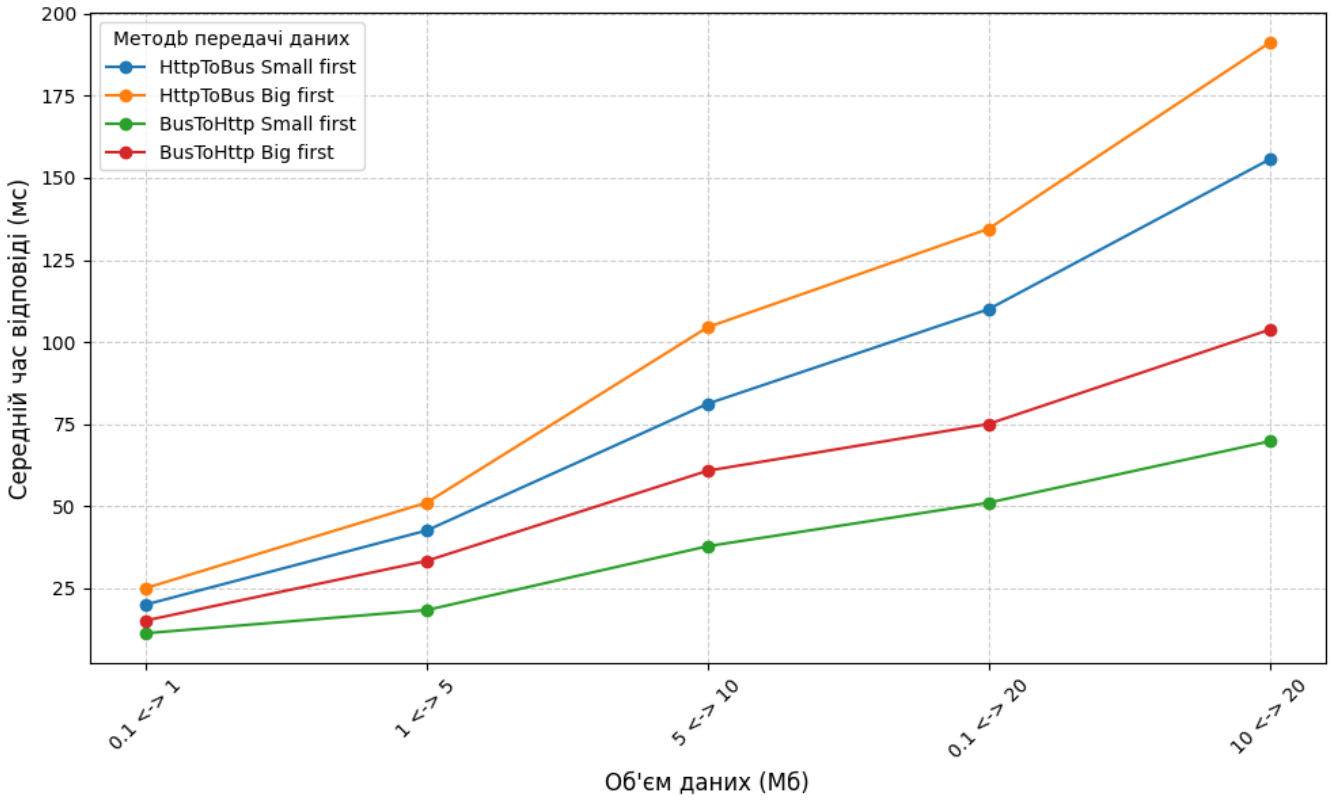

Рисунок 4.19 – Порівняння ефективності передачі даних через HTTP та брокер при передачі різних об’ємів даних

З даного рисунку можна зрозуміти, що, як і попередній метод, при роботі брокера повідомлень з великими об’ємами даних , а протоколу HTTP з малими середній час відповіді на запит покращується майже на двадцять мілісекунд. Проте, варто зауважити, що при використання брокера як першого методу обміну даними швидкість виконання запиту в середньому покращується у два рази, що особливо помітно при роботі з великими об’ємами даних..

Отже, на основі отриманих даних можна зробити висновки, що комунікація через HTTP в комбінації з GRPC є найменш ефективною. Це пов’язано з тим, що

дані методи мають подібні параметри ефективності роботи. Таким чином заміна одного методу на інший мінімальним чином вплине на її роботу. Також при спільному використанні даних методів комунікації максимальний час очікування запиту міг перевищувати показник у двісті мілісекунд, що є найгіршим показників серед проаналізованих методів.

Трохи кращі показники швидкості роботи були отримані при тестуванні комбінації RabbitMQ та HTTP. Брокер повідомлень дозволив більш ефективно оброблювати великі об’єми даних, тоді як HTTP відповідав за обробку меншої за розміром інформації.

Найбільш ефективною була комбінація брокера повідомлень та GRPC, що мала найменшу затримку часу відповіді у розмірі п’ятдесяти мілісекунд при використанні брокера для обробки великих об’ємів даних, що дало найкращі результати.

Отже, на основі отриманих даних можна зробити висновок що найбільш ефективними медами міжсервісної комунікації є підходи, що використовують асинхронну комунікацію на основі брокера повідомлень RabbitMQ, але лише у випадках коли він стоїть першим у черзі виконання запитів.

Використання подвійного GRPC демонструє кращу швидкість виконання запитів при роботі з малими об’ємами інформації та є кращим майже у 2,5 разів за найповільніший метод, проте зі збільшенням розміру обмінюваних даних його ефективність надзвичайно сильно зменшується, а його швидкодія стає однією з найгірших серед інших методів.

Якщо ж розглядати передачу різних об’ємів даних за допомогою різних методів передачі даних в рамках одного запиту, то найбільш помітні відмінності були зафіксовані у методах, що використовують брокер. При передачі більшого об’єму даних через RabbitMQ ефективність роботи методів виявилась швидшою ніж у обернених випадках. Найкраще це видно у комбінації брокера повідомлень та GRPC, де максимальна різниця у ефективності роботи перевищує 50 мілісекунд.

4.5 Висновок

На першому етапі розробки даного розділу було проведено максимально детальний аналіз розробленого програмного продукту, що передбачав ознайомлення з наступними компонентами:

– метод, що використовується для створення даних заданого об’єму, що використовуються під час проведення тестувань;

– огляд методів контролера першого мікросервіса;

– клієнт виконує відправку відповідного запиту за вказаною адресою та очікує відповідь;

– у випадку успішної відповіді результат десеріалізується та повертається у метод з якого було здійснено запит;

– ознайомлення із принципами роботи HTTP клієнта;

– ознайомлення із принципами роботи клієнта та слухача брокера повідомлень RabbitMQ;

– ознайомлення із принципами роботи клієнта та сервера для обміну повідомленнями через GRPC.

Після даних маніпуляцій було здійснено вибір інструментів тестування та створено ряд тестових сценаріїв основна мета яких була оцінити швидкість роботи різних комбінованих методів міжсервісної комунікації за різних умов роботи та при обміні різними об’ємами даних.

На основі отриманих результатів тестування було зроблено висновок, при коректно проведеному розгортанні розробленого програмного продукту та при достатній кількості ресурсів у вигляді оперативної пам’яті, потужності процесора та місця на жорсткому диску розроблений програмний продукт може працювати ідентично як при локальному, так і при розгортанні з допомогою контейнерів. На це вказує середня відсоткова перевага локального методу розгортання, що приблизно дорівнює 3.23%.

Найбільш ефективними методами комунікації виявились комбінації, що використовують брокер повідомлень, але лише у випадках коли він перший виконує запити, що можна пояснити особливостями роботи асинхронних методів комунікації та наявності декількох потоків обробки даних.

Тестування швидкості роботи різних методів міжсервісної комунікації при обробці різного об’єму даних демонструє, що комбінація брокеру повідомлень з іншими методами працює швидше у випадках надсилання великого об’єму інформації через брокер, що пов’язано з його більшою ефективністю роботи з великими масивами інформації.

ВИСНОВКИ

Відштовхуючись від інформації, що була отримана в рамках виконання даної кваліфікаційної роботи було проведено аналіз отриманих результатів окремо для кожного розділу.

Перший етап написання першого розділу у даній роботі передбачав максимально детальне ознайомлення з обраною предметною областю відповідно до обраної теми. Під час аналізу сучасної літератури було проведено аналіз трьох досліджень. У них автори роботи змогли визначити основні виклики пов’язані з розробкою мікросервісної архітектури, визначили декілька ознак, що вказують на ряд аспектів, що можуть демонструвати проблеми чи потенційній слабкі місця, а також спробували оцінити сучасний стан архітектури на основі різноманітних наукових робіт. Отримані результати демонструють, що більшість аспектів розробки мікросервісів потребують проведення подальших досліджень у всіх можливих напрямках.

На основі отриманої інформації було здійснено постановку основних задач, що необхідно виконати в рамках даного проекту та ряд загальних вимог.

У другому розділі було проведено детальний аналіз сучасних методів комунікації мікросервісів, що передбачав дослідження синхронних та асинхронних методів комунікації на прикладі HTTP, GRPC та брокері повідомлень, що передбачало ознайомлення з їхніми принципами роботи та можливими варіантами розгортання системи. Відштовхуючись від отриманих даних було визначено підходи по дослідженню способів комунікації.

Третій розділ даної роботи передбачав виконання проектування програмного забезпечення, що передбачало:

– визначення всіх вимог до програмного забезпечення з допомогою якого будуть проводитись всі види тестувань;

– розробка структури програмного забезпечення на основі сформованих вимог, що включало створення архітектури системи, структури кожного мікросервісу та демонстрацію можливості їхньої взаємодії;

– на основі сформованих вимог та структури програмного продукту було здійснено вибір технологій розробки, що передбачає використання мови програмування C#, фреймворку ASP.NET Core, інструментів Docker та Kubernetes для розгортання системи і k6 для проведення тестувань.

Завершальним етапом виконання роботи був четвертий розділ у якому було здійснено детальний опис програмної реалізації основних компонентів системи до яких відноситься реалізації методів комунікації. По завершенню розробки програмного продукту було проведено ряд тестувань.

Аналіз отриманих результатів свідчать про те, що ефективність роботи міжсерверних комунікацій може виконуватись з ідентичною ефективністю при здійснені коректних налаштувань середовища розгортання та при наявності достатньої кількості апаратних ресурсів. На це вказує середня перевага у швидкості роботи системи у розмірі 3.23% та високий степінь подібності отриманих результатів, що досягає найбільшого відсотку відхилень у розмірі 27,97% при роботі з даними об’єму 0,1 мегабайт, та найбільшої розбіжності у часі у розмірі 29,78 місілекунд при виконанні запитів за участі найбільшого протестованого розміру обмінюваної інформації.

Тестування швидкості роботи різних комбінацій варіантів міжсервесної комунікації демонструє, що незалежно від об’єму обмінюваної інформації найбільш стабільними та ефективними виявились поєднання брокера повідомлень з самим собою та іншими методами. Даний висновок було зроблено на основі порівняння швидкості роботи різних методів, де різниця між найменш ефективним та найшвидшим варіантом коливалась в районі від 270% до 387%.

Не зважаючи на це, комбінація брокера RabbitMQ та протоколу HTTP не продемонструвала настільки ж великого рівня продуктивності, проте при зміні порядку їхнього застосуванню результат покращується приблизно у 2 рази.

Про роботі з малими об’ємами інформації розміром до 1 мегабайту продемонструвало подвійне застосування GRPC, проте при подальшому збільшені даних він демонструє один із найгірших результатів, що пов’язане з необхідністю використання Protobuf, що призводить до необхідності у проведенні ряду додаткових завдань пов’язаних із даними, що призводить до збільшення часу виконання запитів.

Всі інші протестовані комбінації продемонстрували набагато менший середній час очікування відповіді, що вказує на досить низький рівень ефективності, тому їхнє використання може бути доцільним лише у випадках, коли вимоги щодо швидкості виконання запитів є мінімальними, а розробникам необхідно у максимально короткі терміни реалізувати обмін даними між мікросервісами.

Тестування роботи різних комбінацій міжсервісної комунікації при роботі з різними об’ємами інформації продемонструвало, що швидкість їхньої роботи є подібною до попередніх результатів, а використання одного способу передачі даних двічі є схожою не залежно від того, яка інформація оброблялась першою. З даних причин подальший аналіз акцентувався на дослідження інших методів.

Комбінація HTTP та GRPC продемонструвала подібність швидкості виконання запитів не залежно від розміру даних, що вони передають, на основі чого було зроблено висновок, що їхня ефективність завжди буде коливатись в рамках невеликого відхилення.

Одночасне використання HTTP та брокера продемонструвало, що брокер швидше працює з даними великого об’єму, а середній час виконання запиту є кращим ніж у HTTP та GRPC, що робить його використання більш доцільним.

Кращі результати продемонструвало використання брокера повідомлень та GRPC. У випадках, коли брокер відповідав за обробку більшого об’єму інформації середній час відповіді на запит був менший приблизно на п’ятдесят мілісекунд.

ПЕРЕЛІК ДЖЕРЕЛ

1. Söylemez M., Tekinerdogan B., Kolukısa Tarhan A. Challenges and Solution Directions of Microservice Architectures: A Systematic Literature Review // Appl. Sci. – 2022. – Vol. 12. – Art. No. 5507. URL: https://doi.org/10.3390/app121155077

2. Taibi D., Lenarduzzi V. On the Definition of Microservice Bad Smells. IEEE Software. vol 35. 2018. URL: https://doi.org/10.1109/MS.2018.2141031

3. Pahl C., Jamshidi P. Microservices: A Systematic Mapping Study. 2016. URL: https://doi.org/10.5220/0005785501370146

4. Villamizar M., Garcés O., Ochoa L., Castro H., Salamanca L., Verano Merino M., Casallas R., Gil S., Valencia C., Zambrano A., Lang M. Infrastructure Cost Comparison of Running Web Applications in the Cloud Using AWS Lambda and Monolithic and Microservice Architectures. 2016. URL: https://doi.org/10.1109/CCGrid.2016.37

5. Jamshidi P., Pahl C., Mendonça N. C., Lewis J., Tilkov S. Microservices: The Journey So Far and Challenges Ahead. IEEE Software, vol. 35, no. 3, pp. 24-35. 2018. URL: https://doi.org/10.1109/MS.2018.2141039

6. Lewis J.; Fowler, M. Microservices. 2014. URL: https://martinfowler.com/articles/microservices.html

7. Fowler M. Bounded context. 2014. URL: https://martinfowler.com/bliki/BoundedContext.html

8. Best Practice - An Introduction To Domain-Driven Design. 2009. URL: https://learn.microsoft.com/en-us/archive/msdn-magazine/2009/february/best-practice-an-introduction-to-domain-driven-design

9. Işıl K. A., Turgay Ç., Ahmet B. C., Bedir T.. Deployment and communication patterns in microservice architectures: A systematic literature review. Journal of Systems and Software, Volume 180, 2021, 111014. URL: https://doi.org/10.1016/j.jss.2021.111014

10. Microservice Architecture pattern. URL: https://microservices.io/patterns/microservices.html

11. Zmerzlyi I. Мікросервісна архітектура. URL: https://medium.com/@IvanZmerzlyi/microservices-architecture-461687045b3d

12. Microservice Architecture. URL: https://microservices.io/patterns/apigateway.html

13. Про мікросервісну архітектуру. 2024. URL: https://foxminded.ua/mikroservisna-arkhitektura/

14. Microservices Communication via HTTP and JSON. 2024. URL: https://www.geeksforgeeks.org/microservices-communication-via-http-and-json/

15. RabbitMQ 4.0 Documentation. URL: https://www.rabbitmq.com/docs

16. Mahedi H. N. Implement RabbitMQ in .NET Core 8. 2024. URL: https://medium.com/c-sharp-programming/implement-rabbitmq-in-net-core-8-4b518449364d

17. Documentation – GRPC. URL: https://grpc.io/docs/

18. Tronchin G. How to Implement gRPC Client and Service in .NET8. 2024. URL: https://medium.com/@gabrieletronchin/how-to-implement-grpc-client-and-server-in-net-8-2b722b50c3b0

19. Основна документація Docker. URL: https://docs.docker.com/

20. Susnjara S.; Smalley, I. What is Docker? 2024. URL: https://www.ibm.com/topics/docker

21. Campbell J. Kubernetes vs. Docker. URL: https://www.atlassian.com/microservices/microservices-architecture/kubernetes-vs-docker

22. Manricks G. Kubernetes vs Docker: The Backbone of Modern Backend Technologies. 2023. URL: https://www.warp.dev/terminus/kubernetes-vs-docker

23. Iliev, B. Introduction to Docker. GitHub. Retrieved from https://github.com/bobbyiliev/introduction-to-docker-ebook

24. The Definitive Guide to Kubernetes. 2020. URL: https://gabrieltanner.org/blog/the-definitive-guide-to-kubernetes/

25. Shadija D.; Rezai M.; Hill, R. Towards an Understanding of Microservices. In Proceedings of the ICAC 2017—2017 23rd IEEE International Conference on Automation and Computing: Addressing Global Challenges through Automation and Computing,

Huddersfield, UK, 7–8 September 2017. URL: https://www.researchgate.net/publication/320823533_Towards_an_understanding_of_microservices

26. Pahl C.; Lee, B. Containers and clusters for edge cloud architectures - a technology review. In 3rd International Conference on Future Internet of Things and Cloud (FiCloud-2015). IEEE, 2015. URL: https://doi.org/10.1109/FiCloud.2015.35

27. Söylemez M.; Tekinerdogan B.; Kolukısa T., A. Feature-Driven Characterization of Microservice Architectures: A Survey of the State of the Practice. Appl. Sci. 2022, 12, 4424. URL: https://doi.org/10.3390/app12094424

28. Pahl C., Jamshidi P., Zimmermann O. Microservices and Containers. 2020. URL: https://www.researchgate.net/publication/339746834_Microservices_and_Containers

29. Kerimivs A., Scalability and performance of microservices architectures. Global Journal of Computer Science and Technology. 2023. 77-84. URL: https://www.researchgate.net/publication/376065700_Scalability_and_Performance_of_Microservices_Architectures

30. Baresi L., Quattrocchi G., Tamburii D. A. Microservice Architecture Practices and Experience: a Focused Look on Docker Configuration Files. 2022. URL: https://www.researchgate.net/publication/366063465_Microservice_Architecture_Practices_and_Experience_a_Focused_Look_on_Docker_Configuration_Files

31. Velepucha V., Flores P. A Survey on Microservices Architecture: Principles, Patterns and Migration Challenges 2023. URL: https://www.researchgate.net/publication/373151876_A_survey_on_microservices_architecture_Principles_patterns_and_migration_challenges

ДОДАТОК А

(обов’язковий)

ПРОГРАМНИЙ КОД

А.1 – Код контролера першого мікросервісу

```
using FirstService.AsyncDataServices;
using FirstService.Dtos;
using FirstService.Models;
using FirstService.SyncDataServices.GRPC;
using FirstService.SyncDataServices.HTTP;
using Microsoft.AspNetCore.Mvc;
using Microsoft.EntityFrameworkCore;

namespace FirstService.Controllers
{
    [Route("api/[controller]")]
    [ApiController]
    public class FirstServiceController : ControllerBase
    {
        private readonly AppDbContext _context;
        private readonly ISecondServiceDataClient _secondServiceDataClient;
        private readonly ILogger<FirstServiceController> _logger;
        private readonly IGrpcFirstServiceDataClient _grpcFirstServiceDataClient;
        private readonly IMessageBusClient _messageBusClient;

        public FirstServiceController(AppDbContext context,
            ISecondServiceDataClient secondServiceDataClient,
            ILogger<FirstServiceController> logger,
            IGrpcFirstServiceDataClient grpcFirstServiceDataClient,
            IMessageBusClient messageBusClient)
        {
            _context = context;
            _secondServiceDataClient = secondServiceDataClient;
            _logger = logger;
            _grpcFirstServiceDataClient = grpcFirstServiceDataClient;
            _messageBusClient = messageBusClient;
        }

        [HttpPost("HttpToHttp")]
        public     async     Task<IActionResult>     HttpToHttp(CreateToSendDataDto
createToSendDataDto)
        {
            ResponseData responseData = new ResponseData();
            responseData.FirstServiceEnterTime = DateTime.Now;
            responseData.InFirstService = true;

            var toSendData = CreateNewToSendData(createToSendDataDto); ;

            if (toSendData == null)
            {
                return NotFound("Об'єкт не знайдено.");
            }

            responseData.FirstMessageType = "HTTP";
            responseData.SecondMessageType = "HTTP";

            toSendData.MessageTypeFirst = "HTTP";
            toSendData.MessageTypeSecond = "HTTP";

            responseData.FirstServiceSendTime = DateTime.Now;
            var                  response                  =                  await
_secondServiceDataClient.SendDataToSecondService(toSendData);

            if (response == null)
            {
                return StatusCode(500, "Помилка під час отримання відповіді");
```

```
            }
            else
            {
                responseData.InSecondService = response.InSecondService;
                responseData.InThirdService = response.InThirdService;
                responseData.SecondServiceEnterTime                                    =
response.SecondServiceEnterTime;
                responseData.SecondServiceSendTime                                     =
response.SecondServiceSendTime;
                responseData.ThirdServiceEnterTime                                     =
response.ThirdServiceEnterTime;
                responseData.ThirdServiceSendTime = response.ThirdServiceSendTime;

                return Ok(responseData);
            }
        }

        [HttpPost("HttpToGrpc")]
        public     async     Task<IActionResult>     HttpToGrpc(CreateToSendDataDto
createToSendDataDto)
        {
            ResponseData responseData = new ResponseData();
            responseData.FirstServiceEnterTime = DateTime.Now;
            responseData.InFirstService = true;

            var toSendData = CreateNewToSendData(createToSendDataDto); ;

            if (toSendData == null)
            {
                return NotFound("Об'єкт не знайдено.");
            }

            responseData.FirstMessageType = "HTTP";
            responseData.SecondMessageType = "GRPC";

            toSendData.MessageTypeFirst = "HTTP";
            toSendData.MessageTypeSecond = "GRPC";


            var                  response                  =                  await
_secondServiceDataClient.SendDataToSecondService(toSendData);
            if (response == null)
            {
                return StatusCode(500, "Помилка під час отримання відповіді");
            }
            else
            {
                responseData.InSecondService = response.InSecondService;
                responseData.InThirdService = response.InThirdService;
                responseData.SecondServiceEnterTime                                    =
response.SecondServiceEnterTime;
                responseData.SecondServiceSendTime                                     =
response.SecondServiceSendTime;
                responseData.ThirdServiceEnterTime                                     =
response.ThirdServiceEnterTime;
                responseData.ThirdServiceSendTime = response.ThirdServiceSendTime;

                return Ok(responseData);
            }
        }

        [HttpPost("HttpToBus")]
```

```
        public     async     Task<IActionResult>     HttpToBus(CreateToSendDataDto
createToSendDataDto)
        {
            ResponseData responseData = new ResponseData();
            responseData.FirstServiceEnterTime = DateTime.Now;
            responseData.InFirstService = true;

            var toSendData = CreateNewToSendData(createToSendDataDto); ;

            if (toSendData == null)
            {
                return NotFound("Об'єкт не знайдено.");
            }

            responseData.FirstMessageType = "HTTP";
            responseData.SecondMessageType = "BUS";

            toSendData.MessageTypeFirst = "HTTP";
            toSendData.MessageTypeSecond = "BUS";


            var                  response                  =                  await
_secondServiceDataClient.SendDataToSecondService(toSendData);
            if (response == null)
            {
                return StatusCode(500, "Помилка під час отримання відповіді");
            }
            else
            {
                responseData.InSecondService = response.InSecondService;
                responseData.InThirdService = response.InThirdService;
                responseData.SecondServiceEnterTime                                =
response.SecondServiceEnterTime;
                responseData.SecondServiceSendTime                                 =
response.SecondServiceSendTime;
                responseData.ThirdServiceEnterTime                                 =
response.ThirdServiceEnterTime;
                responseData.ThirdServiceSendTime = response.ThirdServiceSendTime;

                return Ok(responseData);
            }
        }

        [HttpPost("GrpcToGrpc")]
        public     async     Task<IActionResult>     GrpcToGrpc(CreateToSendDataDto
createToSendDataDto)
        {
            var toSendData = CreateNewToSendData(createToSendDataDto);
            var                   respons                   =                  await
_grpcFirstServiceDataClient.SendAllData(toSendData);

            _logger.LogInformation($"         Отримані         данні         GRPC:
{respons.InFirstService}");
            return Ok(respons);
        }

        [HttpPost("BusToBus")]
        public     async     Task<IActionResult>     BusToBus(CreateToSendDataDto
createToSendDataDto)
        {
            var toSendData = CreateNewToSendData(createToSendDataDto);
```

```
            ResponseData rest = new ResponseData
            {
                InFirstService = true,
                FirstServiceEnterTime = DateTime.Now,
                FirstServiceSendTime = DateTime.Now,
            };

            toSendData.MessageTypeFirst = "BUS";
            toSendData.MessageTypeSecond = "BUS";

            var respons = await
_messageBusClient.SendDataToSecondService(toSendData);

            if (respons == null)
            {
                _logger.LogInformation($"Дані відсутні");
                return StatusCode(500, "Помилка. Відповідь відсутня");
            }

            respons.FirstMessageType = toSendData.MessageTypeFirst;
            respons.SecondMessageType = toSendData.MessageTypeSecond;
            respons.InFirstService = rest.InFirstService;
            respons.FirstServiceSendTime = rest.FirstServiceSendTime;
            respons.FirstServiceEnterTime = rest.FirstServiceEnterTime;
            respons.InSecondService = rest.InSecondService;
            respons.SecondServiceSendTime = rest.SecondServiceSendTime;
            respons.SecondServiceEnterTime = rest.SecondServiceEnterTime;

            _logger.LogInformation($" Отримані данні GRPC:
{respons.InFirstService}");
            return Ok(respons);
        }

        [HttpPost("BusToGrpc")]
        public async Task<IActionResult> BusToGrpc(CreateToSendDataDto
createToSendDataDto)
        {
            var toSendData = CreateNewToSendData(createToSendDataDto);

            ResponseData rest = new ResponseData
            {
                InFirstService = true,
                FirstServiceEnterTime = DateTime.Now,
                FirstServiceSendTime = DateTime.Now,
            };

            toSendData.MessageTypeFirst = "BUS";
            toSendData.MessageTypeSecond = "GRPC";

            var respons = await
_messageBusClient.SendDataToSecondService(toSendData);

            if (respons == null)
            {
                _logger.LogInformation($"Дані відсутні");
                return StatusCode(500, "Помилка. Відповідь відсутня");
            }

            respons.FirstMessageType = toSendData.MessageTypeFirst;
            respons.SecondMessageType = toSendData.MessageTypeSecond;
            respons.InFirstService = rest.InFirstService;
            respons.FirstServiceSendTime = rest.FirstServiceSendTime;
            respons.FirstServiceEnterTime = rest.FirstServiceEnterTime;
```

```
            respons.InSecondService = rest.InSecondService;
            respons.SecondServiceSendTime = rest.SecondServiceSendTime;
            respons.SecondServiceEnterTime = rest.SecondServiceEnterTime;

            _logger.LogInformation($"         Отримані         данні         GRPC:
{respons.InFirstService}");
            return Ok(respons);
        }

        [HttpGet("getAllToSendData")]
        public IActionResult GetToSendDataById()
        {
            var largeObject = _context.ToSendDatas.ToList();

            if (largeObject == null)
            {
                return NotFound("Об'єкт не знайдено.");
            }

            return Ok(largeObject);
        }

        [HttpGet("getToSendDataById/{id}")]
        public IActionResult GetToSendDataById(int id)
        {
            var  largeObject  =  _context.ToSendDatas.FirstOrDefault(lo  =>  lo.Id  ==
id);

            if (largeObject == null)
            {
                return NotFound("Об'єкт не знайдено.");
            }

            return Ok(largeObject);
        }

        public          ToSendData          CreateNewToSendData(CreateToSendDataDto
createToSendDataDto)
        {
            int sizeInBytes = (int)createToSendDataDto.SizeMbFirst * 1024 * 1024;

            if (sizeInBytes > int.MaxValue / sizeof(char))
            {
                sizeInBytes = 1000;
            }

            var data = new string('X', sizeInBytes);
            var toSendData = new ToSendData
            {
                Data = data,
                SizeMbFirst = createToSendDataDto.SizeMbFirst,
                SizeMbSecond = createToSendDataDto.SizeMbSecond
            };

            return toSendData;
        }

        [HttpPost("deleteToSendData/{id}")]
        public IActionResult DeleteToSendData(int id)
        {
            var  largeObject  =  _context.ToSendDatas.FirstOrDefault(lo  =>  lo.Id  ==
id);
```

```
            if (largeObject == null)
            {
                return NotFound("Об'єкт не знайдено.");
            }

            _context.ToSendDatas.Remove(largeObject);
            _context.SaveChanges();

            return Ok($"Об'єкт з Id #{id} було видалено");
        }

        [HttpPost("HttpToHttpById/{id}")]
        public async Task<IActionResult> HttpToHttpById(int id)
        {
            ResponseData responseData = new ResponseData();
            responseData.FirstServiceEnterTime = DateTime.Now;
            responseData.InFirstService = true;

            var toSendData = await _context.ToSendDatas.FirstOrDefaultAsync(lo =>
lo.Id == id);

            if (toSendData == null)
            {
                return NotFound("Об'єкт не знайдено.");
            }

            responseData.FirstMessageType = "HTTP";
            responseData.SecondMessageType = "HTTP";

            toSendData.MessageTypeFirst = "HTTP";
            toSendData.MessageTypeSecond = "HTTP";

            responseData.FirstServiceSendTime = DateTime.Now;
            var                  response                  =                  await
_secondServiceDataClient.SendDataToSecondService(toSendData);

            if (response == null)
            {
                return StatusCode(500, "Помилка під час отримання відповіді");
            }
            else
            {
                responseData.InSecondService = response.InSecondService;
                responseData.InThirdService = response.InThirdService;
                responseData.SecondServiceEnterTime                                =
response.SecondServiceEnterTime;
                responseData.SecondServiceSendTime                                 =
response.SecondServiceSendTime;
                responseData.ThirdServiceEnterTime                                 =
response.ThirdServiceEnterTime;
                responseData.ThirdServiceSendTime = response.ThirdServiceSendTime;

                return Ok(responseData);
            }
        }

        [HttpPost("HttpToGrpcById/{id}")]
        public async Task<IActionResult> HttpToGrpcById(int id)
        {
            ResponseData responseData = new ResponseData();
            responseData.FirstServiceEnterTime = DateTime.Now;
            responseData.InFirstService = true;
```

```
            var toSendData = await _context.ToSendDatas.FirstOrDefaultAsync(lo =>
lo.Id == id);

            if (toSendData == null)
            {
                return NotFound("Об'єкт не знайдено.");
            }

            responseData.FirstMessageType = "HTTP";
            responseData.SecondMessageType = "GRPC";

            toSendData.MessageTypeFirst = "HTTP";
            toSendData.MessageTypeSecond = "GRPC";


            var                  response                  =                  await
_secondServiceDataClient.SendDataToSecondService(toSendData);
            if (response == null)
            {
                return StatusCode(500, "Помилка під час отримання відповіді");
            }
            else
            {
                responseData.InSecondService = response.InSecondService;
                responseData.InThirdService = response.InThirdService;
                responseData.SecondServiceEnterTime                                =
response.SecondServiceEnterTime;
                responseData.SecondServiceSendTime                                 =
response.SecondServiceSendTime;
                responseData.ThirdServiceEnterTime                                 =
response.ThirdServiceEnterTime;
                responseData.ThirdServiceSendTime = response.ThirdServiceSendTime;

                return Ok(responseData);
            }
        }

        [HttpPost("HttpToBusById/{id}")]
        public async Task<IActionResult> HttpToBusById(int id)
        {
            ResponseData responseData = new ResponseData();
            responseData.FirstServiceEnterTime = DateTime.Now;
            responseData.InFirstService = true;

            var toSendData = await _context.ToSendDatas.FirstOrDefaultAsync(lo =>
lo.Id == id);

            if (toSendData == null)
            {
                return NotFound("Об'єкт не знайдено.");
            }

            responseData.FirstMessageType = "HTTP";
            responseData.SecondMessageType = "BUS";

            toSendData.MessageTypeFirst = "HTTP";
            toSendData.MessageTypeSecond = "BUS";


            var                  response                  =                  await
_secondServiceDataClient.SendDataToSecondService(toSendData);
            if (response == null)
            {
```

```
                return StatusCode(500, "Помилка під час отримання відповіді");
            }
            else
            {
                responseData.InSecondService = response.InSecondService;
                responseData.InThirdService = response.InThirdService;
                responseData.SecondServiceEnterTime                                    =
response.SecondServiceEnterTime;
                responseData.SecondServiceSendTime                                     =
response.SecondServiceSendTime;
                responseData.ThirdServiceEnterTime                                     =
response.ThirdServiceEnterTime;
                responseData.ThirdServiceSendTime = response.ThirdServiceSendTime;

                return Ok(responseData);
            }
        }

        [HttpPost("GrpcToGrpcById/{id}")]
        public async Task<IActionResult> GrpcToGrpcById(int id)
        {
            var toSendData = await _context.ToSendDatas.FirstOrDefaultAsync(lo =>
lo.Id == id);

            ResponseData rest = new ResponseData
            {
                InFirstService = true,
                FirstServiceEnterTime = DateTime.Now,
                FirstServiceSendTime = DateTime.Now,
            };

            var                    respons                    =                    await
_grpcFirstServiceDataClient.SendAllData(toSendData);

            toSendData.MessageTypeFirst = "GRPC";
            toSendData.MessageTypeSecond = "GRPC";

            respons.FirstMessageType = toSendData.MessageTypeFirst;
            respons.SecondMessageType = toSendData.MessageTypeSecond;
            respons.InFirstService = rest.InFirstService;
            respons.FirstServiceSendTime = rest.FirstServiceSendTime;
            respons.FirstServiceEnterTime = rest.FirstServiceEnterTime;
            respons.InSecondService = rest.InSecondService;
            respons.SecondServiceSendTime = rest.SecondServiceSendTime;
            respons.SecondServiceEnterTime = rest.SecondServiceEnterTime;

            _logger.LogInformation($"         Отримані         данні         GRPC:
{respons.InFirstService}");
            return Ok(respons);
        }

        [HttpPost("BusToBusById/{id}")]
        public async Task<IActionResult> BusToBusById(int id)
        {
            var toSendData = await _context.ToSendDatas.FirstOrDefaultAsync(lo =>
lo.Id == id);


            ResponseData rest = new ResponseData
            {
                InFirstService = true,
                FirstServiceEnterTime = DateTime.Now,
                FirstServiceSendTime = DateTime.Now,
```

```
            };

            toSendData.MessageTypeFirst = "BUS";
            toSendData.MessageTypeSecond = "BUS";

            var                  respons                  =                  await
_messageBusClient.SendDataToSecondService(toSendData);

            if (respons == null)
            {
                _logger.LogInformation($"Дані відсутні");
                return StatusCode(500, "Помилка. Відповідь відсутня");
            }

            respons.FirstMessageType = toSendData.MessageTypeFirst;
            respons.SecondMessageType = toSendData.MessageTypeSecond;
            respons.InFirstService = rest.InFirstService;
            respons.FirstServiceSendTime = rest.FirstServiceSendTime;
            respons.FirstServiceEnterTime = rest.FirstServiceEnterTime;
            respons.InSecondService = rest.InSecondService;
            respons.SecondServiceSendTime = rest.SecondServiceSendTime;
            respons.SecondServiceEnterTime = rest.SecondServiceEnterTime;

            _logger.LogInformation($"         Отримані         данні         GRPC:
{respons.InFirstService}");
            return Ok(respons);
        }

        [HttpPost("BusToGrpcById/{id}")]
        public async Task<IActionResult> BusToGrpcById(int id)
        {
            var toSendData = await _context.ToSendDatas.FirstOrDefaultAsync(lo =>
lo.Id == id);

            ResponseData rest = new ResponseData
            {
                InFirstService = true,
                FirstServiceEnterTime = DateTime.Now,
                FirstServiceSendTime = DateTime.Now,
            };

            toSendData.MessageTypeFirst = "BUS";
            toSendData.MessageTypeSecond = "GRPC";

            var                  respons                  =                  await
_messageBusClient.SendDataToSecondService(toSendData);

            if (respons == null)
            {
                _logger.LogInformation($"Дані відсутні");
                return StatusCode(500, "Помилка. Відповідь відсутня");
            }

            respons.FirstMessageType = toSendData.MessageTypeFirst;
            respons.SecondMessageType = toSendData.MessageTypeSecond;
            respons.InFirstService = rest.InFirstService;
            respons.FirstServiceSendTime = rest.FirstServiceSendTime;
            respons.FirstServiceEnterTime = rest.FirstServiceEnterTime;
            respons.InSecondService = rest.InSecondService;
            respons.SecondServiceSendTime = rest.SecondServiceSendTime;
            respons.SecondServiceEnterTime = rest.SecondServiceEnterTime;
```

```
            _logger.LogInformation($"         Отримані         данні         GRPC:
{respons.InFirstService}");
            return Ok(respons);
        }

    }
}
```

А.2 – HTTP клієнт першого мікросервісу

```
using FirstService.Models;
using System.Text.Json;

namespace FirstService.SyncDataServices.HTTP
{
    public class HttpSecondServiceDataClient : ISecondServiceDataClient
    {
        private readonly HttpClient _httpClient;
        private readonly IConfiguration _configuration;
        private readonly ILogger<HttpSecondServiceDataClient> _logger;

        public HttpSecondServiceDataClient(HttpClient httpClient, IConfiguration
configuration, ILogger<HttpSecondServiceDataClient> logger)
        {
            _httpClient = httpClient;
            _configuration = configuration;
            _logger = logger;
        }

        public async Task<ResponseData> SendDataToSecondService(ToSendData
toSendData)
        {
            string? secondServiceUrl = null;
            switch (toSendData.MessageTypeSecond)
            {
                case "HTTP":
                    secondServiceUrl = _configuration["HttpToHttp"];
                    break;
                case "GRPC":
                    secondServiceUrl = _configuration["HttpToGRPC"];
                    break;
                case "BUS":
                    secondServiceUrl = _configuration["HttpToBus"];
                    break;
            }

            if (string.IsNullOrEmpty(secondServiceUrl))
            {
                _logger.LogError("URL для SecondService не заданий у
конфігурації.");
                return null;
            }

            using (var memoryStream = new MemoryStream())
            {
                await JsonSerializer.SerializeAsync(memoryStream, toSendData);

                memoryStream.Seek(0, SeekOrigin.Begin);

                var streamContent = new StreamContent(memoryStream)
                {
                    Headers =
```

```
                    {
                        ContentType = new
System.Net.Http.Headers.MediaTypeHeaderValue("application/json")
                    }
                };

                try
                {
                    var response = await _httpClient.PostAsync(secondServiceUrl,
streamContent);

                    if (response.IsSuccessStatusCode)
                    {
                        _logger.LogInformation("-> Синхронний POST до
SecondService пройшов успішно!");

                        var responseContent = await
response.Content.ReadAsStringAsync();
                        var toSendDataResponse =
JsonSerializer.Deserialize<ResponseData>(
                            responseContent,
                            new JsonSerializerOptions {
PropertyNameCaseInsensitive = true });

                        if (toSendDataResponse == null)
                        {
                            _logger.LogError("Не вдалося десеріалізувати відповідь
у ResponseData.");
                            return null;
                        }

                        return toSendDataResponse;
                    }
                    else
                    {
                        _logger.LogError($"Не вдалося отримати правильну відповідь
від SecondService. Статус: {response.StatusCode}");
                        return null;
                    }
                }
                catch (Exception ex)
                {
                    _logger.LogError(ex, "Помилка під час відправки запиту до
SecondService.");
                    return null;
                }
            }
        }
    }
}
```

А.3 – GRPC клієнт першого мікросервісу

```
using AutoMapper;
using FirstService.Models;
using Grpc.Net.Client;

namespace FirstService.SyncDataServices.GRPC
{
    public class GrpcFirstServiceDataClient : IGrpcFirstServiceDataClient
    {
        private readonly IConfiguration _configuration;
```

```
        private readonly IMapper _mapper;
        private readonly GrpcChannel _channel;
        private readonly Services.Protos.GrpcFirstService.GrpcFirstServiceClient
_client;

        public GrpcFirstServiceDataClient(IConfiguration configuration, IMapper
mapper)
        {
            _configuration = configuration;
            _mapper = mapper;

            _channel = GrpcChannel.ForAddress(_configuration["GrpcPlatform"], new
GrpcChannelOptions
            {
                MaxReceiveMessageSize = 1000 * 1024 * 1024,
                MaxSendMessageSize = 1000 * 1024 * 1024
            });

            _client = new
Services.Protos.GrpcFirstService.GrpcFirstServiceClient(_channel);
        }

        public async Task<ResponseData> SendAllData(ToSendData toSendData)
        {
            var grpcRequest = _mapper.Map<Services.Protos.ToSendData>(toSendData);

            try
            {
                var reply = await
_client.SendDataToSecondServiceAsync(grpcRequest);
                return _mapper.Map<Models.ResponseData>(reply);
            }
            catch (Exception ex)
            {
                Console.WriteLine($"--> Could not call GRPC Server:
{ex.Message}");
                return null;
            }
        }
    }
}
```

А.4 – GRPC сервер другого мікросервісу

```
using AutoMapper;
using Grpc.Core;
using Services.Protos;
using SecondService.Controllers;
using Google.Protobuf;

namespace SecondService.SyncDataServices.GRPC
{
    public class GrpcSecondService : GrpcFirstService.GrpcFirstServiceBase
    {
        private readonly IMapper _mapper;
        private readonly ILogger<GrpcSecondService> _logger;
        private readonly ISecondServiceProcessor _processor;

        public GrpcSecondService(IMapper mapper, ILogger<GrpcSecondService>
logger, ISecondServiceProcessor processor)
        {
            _mapper = mapper;
```

```
            _logger = logger;
            _processor = processor;
        }

        public override async Task<ResponseData>
SendDataToSecondService(ToSendData request, ServerCallContext context)
        {
            ResponseData res = new ResponseData
            {
                InSecondService = true
            };

            var memoryStream = new MemoryStream(request.ToByteArray());
            var deserializedRequest =
Services.Protos.ToSendData.Parser.ParseFrom(memoryStream);

            Models.ToSendData toSendData =
_mapper.Map<Models.ToSendData>(deserializedRequest);

            var result = await _processor.ProcessDataAsync(toSendData);
            var mappedResult = _mapper.Map<ResponseData>(result);

            _logger.LogInformation($"ID {res.FirstMessageType},
{res.ThirdServiceEnterTime}, {request.SizeMb}");

            return res;
        }

        private ToSendData ProcessAndReturnToSendData(ToSendData toSendData)
        {
            if (toSendData == null)
            {
                _logger.LogError("Об'єкт ToSendData не може бути порожнім.");
                return null;
            }

            if (toSendData.Multiplier == 0)
            {
                _logger.LogError("Значення Multiplier не може бути нульовим.");
                return null;
            }

            int newSizeInBytes = (int)(toSendData.SizeMb * 1024 * 1024 *
toSendData.Multiplier);

            if (newSizeInBytes > int.MaxValue / sizeof(char))
            {
                _logger.LogError("Новий розмір об'єкта перевищує максимальний
допустимий розмір.");
                return null;
            }

            toSendData.Data = new string('X', newSizeInBytes);
            toSendData.SizeMb = toSendData.SizeMb * toSendData.Multiplier;

            return toSendData;
        }
    }
}
```

А.5 – RabbitMQ клієнт першого мікросервісу

```
using FirstService.Models;
using RabbitMQ.Client;
using RabbitMQ.Client.Events;
using System.Text;
using System.Text.Json;

namespace FirstService.AsyncDataServices
{
    public class MessageBusClient : IMessageBusClient
    {
        private readonly IConnection _connection;
        private readonly ILogger<MessageBusClient> _logger;
        private readonly IConfiguration _configuration;
        private readonly IModel _channel;
        private readonly string _replyQueueName;
        private readonly EventingBasicConsumer _consumer;
        private readonly TaskCompletionSource<string> _responseTcs;
        private readonly string _correlationId;
        private readonly IBasicProperties _props;

        public MessageBusClient(IConfiguration configuration,
ILogger<MessageBusClient> logger)
        {
            _configuration = configuration;
            _logger = logger;
            var factory = new ConnectionFactory()
            {
                HostName = _configuration["RabbitMQHost"],
                Port = int.Parse(_configuration["RabbitMQPort"])
            };

            try
            {
                _connection = factory.CreateConnection();
                _channel = _connection.CreateModel();
                _channel.ExchangeDeclare(exchange: "trigger", type:
ExchangeType.Fanout);

                _replyQueueName = _channel.QueueDeclare().QueueName;
                _consumer = new EventingBasicConsumer(_channel);
                _correlationId = Guid.NewGuid().ToString();
                _responseTcs = new TaskCompletionSource<string>();

                _consumer.Received += (model, ea) =>
                {
                    if (ea.BasicProperties.CorrelationId == _correlationId)
                    {
                        var response = Encoding.UTF8.GetString(ea.Body.ToArray());
                        _responseTcs.SetResult(response);
                    }
                };

                _channel.BasicConsume(
                    consumer: _consumer,
                    queue: _replyQueueName,
                    autoAck: true);

                _props = _channel.CreateBasicProperties();
                _props.CorrelationId = _correlationId;
                _props.ReplyTo = _replyQueueName;
```

```
                _connection.ConnectionShutdown += RabbitMQ_ConnectionShutdown;
                _logger.LogInformation("Connected to RabbitMQ");
            }
            catch (Exception ex)
            {
                _logger.LogError($"Failed to connect to RabbitMQ: {ex.Message}");
            }
        }

        public Task<ResponseData> SendDataToSecondService(ToSendData toSendData)
        {
            if (_connection == null || _channel == null)
            {
                _logger.LogError("--> RabbitMQ connection is not initialized.");
                throw new InvalidOperationException("RabbitMQ: Connection is not
initialized.");
            }

            var message = JsonSerializer.Serialize(toSendData);
            var body = Encoding.UTF8.GetBytes(message);

            if (_connection.IsOpen)
            {
                _channel.BasicPublish(
                    exchange: "trigger",
                    routingKey: "",
                    basicProperties: _props,
                    body: body);

                _logger.LogInformation("Message sent");

                return _responseTcs.Task.ContinueWith(t =>
                {
                    if (t.IsFaulted)
                    {
                        _logger.LogError("Error in receiving response");
                        throw new Exception("Error in receiving response");
                    }

                    return JsonSerializer.Deserialize<ResponseData>(t.Result);
                });
            }
            else
            {
                _logger.LogWarning("--> RabbitMQ connection is closed, not
sending.");
                throw new InvalidOperationException("RabbitMQ connection is
closed.");
            }
        }

        private void RabbitMQ_ConnectionShutdown(object sender, ShutdownEventArgs
e)
        {
            _logger.LogWarning("RabbitMQ connection shutdown");
        }

        public void Dispose()
        {
            _logger.LogInformation("Disposing MessageBusClient");
            if (_channel.IsOpen)
            {
                _channel.Close();
```

```
                _connection.Close();
            }
        }
    }
}
```

A.6 – RabbitMQ сервер першого мікросервісу

```
using RabbitMQ.Client;
using RabbitMQ.Client.Events;
using SecondService.Models;
using System.Text;
using System.Text.Json;

namespace SecondService.AsyncDataServices
{
    public class MessageBusSubscriber : BackgroundService
    {
        private readonly IConfiguration _configuration;
        private readonly ILogger<MessageBusSubscriber> _logger;
        private readonly IProcessData _processData;
        private IConnection _connection;
        private IModel _channel;

        public MessageBusSubscriber(IConfiguration configuration,
ILogger<MessageBusSubscriber> logger, IProcessData processData)
        {
            _configuration = configuration;
            _logger = logger;
            _processData = processData;
            InitializeRabbitMQ();
        }

        private void InitializeRabbitMQ()
        {
            var factory = new ConnectionFactory()
            {
                HostName = _configuration["RabbitMQHost"],
                Port = int.Parse(_configuration["RabbitMQPort"])
            };

            _connection = factory.CreateConnection();
            _channel = _connection.CreateModel();
            _channel.QueueDeclare(queue: "request-queue", exclusive: false);
            _connection.ConnectionShutdown += RabbitMQ_ConnectionShutdown;

            _logger.LogInformation("Connected to MessageBus");
        }

        protected override Task ExecuteAsync(CancellationToken stoppingToken)
        {
            stoppingToken.ThrowIfCancellationRequested();

            var consumer = new EventingBasicConsumer(_channel);
            consumer.Received += async (model, ea) =>
            {
                _logger.LogInformation("Received message from queue.");

                var body = ea.Body.ToArray();
                var message = Encoding.UTF8.GetString(body);
                var requestData = JsonSerializer.Deserialize<ToSendData>(message);
```

```
                var response = await _processData.ProcessDataAsync(requestData);

                SendMessage(response, ea.BasicProperties.ReplyTo,
ea.BasicProperties.CorrelationId);

                _channel.BasicAck(deliveryTag: ea.DeliveryTag, multiple: false);
            };

            _channel.BasicConsume(queue: "request-queue", autoAck: false,
consumer: consumer);

            return Task.CompletedTask;
        }

        private void SendMessage(ResponseData response, string replyTo, string
correlationId)
        {
            _logger.LogInformation($"Sending response with CorrelationId:
{correlationId}");
            var body = Encoding.UTF8.GetBytes(JsonSerializer.Serialize(response));

            var properties = _channel.CreateBasicProperties();
            properties.CorrelationId = correlationId;

            _channel.BasicPublish(
                exchange: "",
                routingKey: replyTo,
                basicProperties: properties,
                body: body
            );
        }

        private void RabbitMQ_ConnectionShutdown(object sender, ShutdownEventArgs
e)
        {
            _logger.LogWarning("RabbitMQ connection shutdown.");
        }

        public override void Dispose()
        {
            _logger.LogInformation("Disposing MessageBusSubscriber");
            if (_channel.IsOpen)
            {
                _channel.Close();
                _connection.Close();
            }
            base.Dispose();
        }
    }
}
```

ДОДАТОК Б

(обов’язковий)

КОПІЯ НАУКОВОЇ ПУБЛІКАЦІЇ

ISSN 2307-5732
DOI 10.31891/2307-5732

НАУКОВИЙ ЖУРНАЛ **3.2024**

ВІСНИК

Хмельницького національного університету

Технічні науки

Technical sciences

SCIENTIFIC JOURNAL
HERALD OF KHMELNYTSKYI NATIONAL UNIVERSITY

2024, Issue 3, Volume 335, Part 1

Хмельницький

DOI 10.31891/2307-5732-2024-335-3-55
УДК 004.273

ГРИГА СЕРГІЙ
Хмельницький національний університет
https://orcid.org/0009-0002-4409-4531
e-mail: grigha71@gmail.com
ПРАВОРСЬКА НАТАЛІЯ
Хмельницький національний університет
https://orcid.org/0000-0001-6001-3311
e-mail: margana2000007@gmail.com

МЕТОДИ РЕАЛІЗАЦІЇ МІКРОСЕРВІСНИХ АРХІТЕКТУР: ПЕРЕВАГИ ТА НЕДОЛІКИ, ВПРОВАДЖЕННЯ ТА ТЕСТУВАННЯ ПРИ РОЗРОБЦІ ПРОГРАМНИХ ЗАСТОСУНКІВ

Розробка мікросервісної архітектури є складним та комплексним процесом, що потребує глибокого розуміння багатьох аспектів її реалізації. У даній статті було розглянуто дану архітектуру, її переваги та недоліки, у яких випадках її доцільно використовувати, деякі важливі методи, технології, патерни та підходи до тестування. Особливий акцент робиться на важливість коректного розбиття функціоналу системи на мікросервіси, що можна ефективно провести за допомого підходу Еріка Еванса «domain-driven design» та використанню граничних контекстів.

Серед важливих технологій, які було розглянуто, відносяться контейнеризація та оркестрація контейнерів, що при використанні у зв'язці дозволяють помістити кожен мікросервіс у окремий контейнер які можна швидко та легкого запустити на різних операційних системах без необхідності попереднього налаштування їхніх параметрів, допомагає ефективно використовувати і розподіляти ресурси між ними, а також спрощує процес управління.

Ефективність, та ряд інших важливих параметрів програмного продукту, напряму залежать від обраних патернів проектування та якості їхньої реалізації, наприклад, патерн вимикач при коректній розробці та налаштуванні може забезпечувати стабільну роботу всієї системи навіть за умови некоректної роботи окремих мікросервісів, що тимчасово обмежуються і стають недоступними для виклику до виправлення всіх помилок, чи шлюз та патерн «Sega», що дозволяють забезпечити маршрутизацію запитів та підтримувати коректну роботу кожного мікросервісу за умов виникнення різного виду помилок.

Тестування мікросервісної архітектури є надзвичайно важливим та комплексним завданням, від якого залежить вся подальша робота системи, тому було визначено, що його доцільно проводити одночасно за допомогою декількох різних методів тестування, а саме юніт-тестів, інтегрованого та наскрізного тестування, що у сукупності дозволяють на різних рівнях та етапах розробки перевірити коректність роботи, як усієї системи, так і її окремих елементів, що дозволяє швидко виявити помилки та усунути їх до розгортання програмного забезпечення.

В результаті досліджень було визначено та проаналізовано переваги та недоліки мікросервісної архітектури, доцільність її використання та ряд сучасних технологій, патернів проектування та способів тестування, що в сукупності допоможуть уникнути ряду проблем при розробці та створити якісну мікросервісну архітектуру.

Ключові слова: мікросервіси, мікросервісна архітектура, технології розробки програмних застосунків, патерни проектування, конструювання програмного забезпечення, тестування.

HRYHA SERHII , PRAVORSKA NATALYA
Khmelnitsky national university, Ukraine

METHODS FOR IMPLEMENTING MICROSERVICE ARCHITECTURES: ADVANTAGES AND DISADVANTAGES, IMPLEMENTATION AND TESTING IN THE DEVELOPMENT OF SOFTWARE APPLICATIONS

Developing a microservice architecture is a complex and comprehensive process that requires a deep understanding of many aspects of its implementation. This article discusses this architecture, its advantages and disadvantages, when it is appropriate to use it, and some important methods, technologies, patterns, and approaches to testing. Particular emphasis is placed on the importance of correctly dividing the system functionality into microservices, which can be effectively done with the help of Eric Evans' domain-driven design approach and the use of boundary contexts.

Among the important technologies that were considered were containerization and container orchestration, which, when used in conjunction, allow each microservice to be placed in a separate container that can be quickly and easily run on different operating systems without the need to pre-configure their parameters, helps to efficiently use and allocate resources between them, and simplifies the management process.

Efficiency, and a number of other important parameters of a software product, directly depend on the selected design patterns and the quality of their implementation, for example, the switch pattern, if properly designed and configured, can ensure stable operation of the entire system even if individual microservices are not working correctly, temporarily limited and become unavailable for calling until all errors are corrected, or the gateway and the Sega pattern, which allow you to route requests and maintain the correct operation of each microservice in the event of a problem.

Testing of microservice architecture is an extremely important and complex task, on which all further operation of the system depends, so it was determined that it should be carried out simultaneously using several different testing methods, namely unit tests, integrated and end-to-end testing, which together allow at different levels and stages of development to check the correctness of the operation of both the entire system and its individual elements, which allows you to quickly identify errors and eliminate them before deploying the software.

As a result of the research, the advantages and disadvantages of microservice architecture, the feasibility of its use, and a number of modern technologies, design patterns and testing methods were identified and analyzed, which together will help to avoid a number of problems in the development and create a high-quality microservice architecture.

Keywords: microservices, microservice architecture, software application development technologies, design patterns, software design, testing.

Постановка проблеми

На сьогоднішній день розробка програмних продуктів відбувається в умовах стрімкого розвитку різноманітних технологій, а одним із важливих аспектів є підхід до розробки архітектури програмного забезпечення. У даному контексті використання мікросервісної архітектури набуває все більшої популярності серед розробників, що дозволяє зосередитись на розділені функціональних можливостей програми на невеликі за розміром та незалежні сервіси.

Однак, як і всі інші архітектури, разом із рядом переваг виникають різноманітні виклики та проблеми, що потребують швидкого вирішення. До таких проблем можна віднести підхід до розбиття функціоналу на мікросервіси, методи їх взаємодії, роботи, підвищена складність розробки системи, проблеми з розгортанням та тестуванням.

З даних причин надзвичайно важливим є чітке розуміння доцільності використання мікросервісної архітектури, її основних переваги та недоліки при використанні, сучасні методи і підходи до її розробки, технології, патерни проектування та найбільш ефективні методи тестування. Лише комплексне розуміння даних аспектів всіх цих аспектів дозволить мінімізувати негативний вплив недоліків даної архітектури, покращить якість всієї системи.

Аналіз останніх джерел

На сьогоднішній день існує велика кількість досліджень, що пов’язані з мікросервісною архітектурою. Більшість із них акцентує увагу на конкретних проблемах архітектури, методах, інструментах та технологіях, що використовуються при розробці мікросервісної архітектури, а також різноманітних аспектах, що можуть напряму впливати на якість розробки. Однак, вичерпні дослідження мікросервісної архітектури майже повністю відсутні.

У дослідженні [1] дослідники акцентують увагу на тому, що мікросервісна архітектура має ряд серйозних недоліків та викликів, які варто ретельно оцінити на етапі проектування та визначити потенційні можливості їхнього вирішення. В роботі проводиться систематичний огляд літератури та відбір ряду статей, що використовувались як основа для подальшого дослідження. На їхній базі дослідники змогли визначити дев’ять основних категорій викликів, сорок підкатегорій і потенційні напрями їх вирішення.

У дослідженнях [2] та [3] дослідники аналізують сучасні підходи ефективного використання контейнерів у мікросервісній архітектурі включно із застосуванням хмарних технологій, тоді як у дослідженні [4] на основі опитування експертів та порівняння їхніх результатів з існуючими дослідженнями, для визначення можливостей мікросервісної та безсерверної архітектури у порівнянні з монолітною архітектурою. Результати дослідження вказують на те, що дані архітектури можуть ефективно вирішувати ряд проблем, що включають масштабованість і продуктивності.

На основі проаналізованих джерел можна зробити висновок, що дослідження мікросервісної архітектури здійснюється у багатьох напрямках, але вони не є повністю вичерпними і потребують подальших детальних досліджень у даних напрямках.

Виклад основного матеріалу

Мікросервісна архітектура – це підхід до розробки програмного забезпечення, що передбачає створення застосунку у вигляді набору невеликих та незалежних компонентів, що мають назву мікросервіси, кожен з яких виконує конкретну функцію, що працює повністю автономно та комунікує з іншими сервісами за допомогою легких механізмів, наприклад, HTTP.

Як будь-яка інша архітектура, вона має як переваги, так і серйозні недоліки. До переваг використання мікросервісної архітектури відносять:

– можливість незалежного розгортання компонентів програмного продукту;

– можливість залучення великої кількості незалежних команд розробників, що можуть використовувати різні мови програмування чи інші технології;

– можливість повторного використання коду;

– можливість незалежного маштабування кожного сервісу;

– високий рівень ізоляції неполадок, що дозволяє уникати поломки всієї системи.

До недоліків мікросервісної архітектури відносять:

– менша швидкість виконання програмного коду у порівнянні з іншими архітектурами;

– збільшену складність розробки програмного продукту ;

– висока складність підтримки сервісів через їхню велику кількість ;

– висока вартість розробки ;

– необхідність достатньо високого рівня знань розробників для отримання якісного результату.

Окремо варто визначити доцільність використання мікросервісної архітектури в кожному конкретному випадку. Це особливо важливий етап розробки, адже дана архітектура, хоч і має серйозні переваги над іншими, недоліки є не менш критичними. В загальному випадку можна виділити три основних причини використання мікросервісної архітектури:

– програмний продукт є об’ємним за розміром і є плани по його майбутньому розвитку при наявності декількох функцій, що можуть потребувати незалежного масштабування;

– програмний продукт передбачає використання динамічних мов програмування та є великим за об’ємом;

– команда розробників складається із великої кількості спеціалістів, що використовують різні мови програмування та технології.

Одним із перших серйозних викликів розробки мікросервісної архітектури є розбиття функціональних можливостей системи на окремі мікросервіси. Дати однозначну відповідь на дане запитання неможливо, адже не існує жодної кількісної величини, адже вони не враховують контексту бізнес-функцій. Для вирішення поставленого завдання доцільно використовувати підхід Еріка Еванса «domain-driven design» (DDD), що фокусується на вивчені предметної області бізнесу чи окремих процесів.

В основі даного підходу знаходиться тісна співпраця замовника та розробника, що ставить собі за мету розуміння всіх процесів обома сторонами, в основі якого лежить розподіл застосунку на домени.

Домен являє собою область знань чи діяльності, що описує ряд завдань. Наприклад, у секторі фінансових послуг доменом може виступати управління платежами чи кредитні операції. Кожен домен складається із субдоменів, що є множиною конкретних завдань чи бізнес аспектів, наприклад, у домені управління платежами можуть буди субдомени обробки платежів, управління транзакціями, перевірка балансу. У свою чергу субдомени можна робити на менші елементи:

– основні субдомени – критично важливі для бізнесу та забезпечують ключові функції;

– допоміжні субдомени – це набір функцій, що підтримують основні субдомени та забезпечують їхню стабільну роботу, але вони не є критичними;

– загальні субдомени – це набір неунікальних, загальних функцій чи сервісів, що використовуються в різних системах.

На базі основних, допоміжних і загальних субдоменів здійснюється подальше створення мікросервісів. Наприклад, компанія займається доставкою посилок, тому її доменом буде доставка та менеджер доставок. Для забезпечення їхньої коректної роботи у моделі використовуються субдомени.

У випадку даного прикладу субдоменами виступає створення маршруту, підбір персоналу, відслідковування посилки, повернення посилки, оплата, технічна підтримка, користувачі та персональні акції. Варто враховувати, що розробку моделі доцільно здійснювати з врахуванням майбутнього функціоналу, наприклад персональні акції можуть бути реалізовані, як майбутні оновлення застосунку. Відповідну модель зображено на рисунку 1.

Але варто розуміти, що одне лише розбиття функціоналу на окремі домени різних рівнів не може в повному обсягу визначити чіткі межі їхніх завдань. Для вирішення даної проблеми у DDD існує поняття граничного контексту.

Рисунок 1. Модель розбиття системи доставки товарів на домени

Граничний контекст (Bounded Context) – це логічна мережа, що визначає область відповідальності конкретної моделі в мережах системи. Основним призначенням граничного контексту є визначення: де починається і закінчується конкретна модель, із визначенням чітких меж її термінології, правил та поведінки. У «Domain-Driven Design» використання даного контексту ставить собі за мету розподіл складних доменів на окремі ізольовані та керовані частини з метою уникнення конфліктів між моделями. Розбиття на контексти здійснюється на основі аналізу функціональних відмінностей кожного домену.

На відміну від моделі на основі доменів, яка представляє собою відображення предметів реального світу, але у різних частинах системи, не є доцільним використання однакових представлень ідентичних речей. Наприклад, домени «Менеджер доставки» та «Доставка» містять дані про посилку, але «Менеджер доставки» потребує лише даних про кінцеве місце призначення та місце відправки, тоді як домен «Доставка» додатково

потребує даних про клієнта для його ідентифікації. Такий підхід до проектування дозволяє зменшити складність всієї системи та дозволить робити зміни в одній моделі без вплину на іншу. Приклад розбиття на граничні контексти можна побачити на рисунку 2.

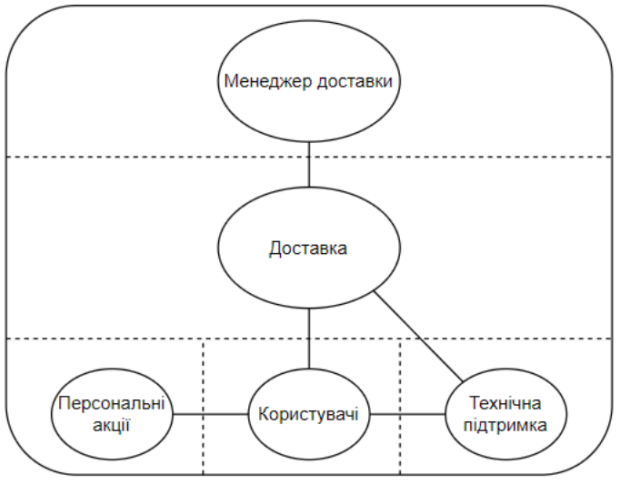

Рисунок 2. Граничні контексти

Технології для створення мікросервісів

У мікросервісній архітектурі важливо забезпечити максимальну ізольованість кожного мікросервіса від інших, високий рівень портативності та можливість швидкого розгортання. Для забезпечення цих характеристик доцільно використовувати контейнери.

Контейнер – це легкі та ізольовані середовища, що використовуються для запуску декілька ізольованих просторів на одному ядрі, що включають у себе програмний код, середовище для його виконання, інструменти, бібліотеки та налаштування. З їх допомогою будь-який мікросервіс може бути запущеним у різних середовищах розгортання і при цьому працювати коректно.

Сучасне програмне забезпечення для створення контейнерів, зазвичай, складається з трьох етапів:

– етап 1 – створення файлів з налаштуваннями та файлу з програмним кодом та необхідними додатковими компонентами;

– етап 2 – доставка отриманих файлів на сервер, де буде розгорнуто програму;

– етап 3 – запуск отриманих файлів для створення контейнера.

Головною проблемою при використання контейнерів є забезпечення ефективного управління контейнерами. Для вирішення даної проблеми доцільним є застосування процесу оркестрації контейнерів.

Оркестрація контейнерів – це процес автоматизації розгортання, управління та забезпечення взаємодії між контейнерами, що дозволяє розробникам керувати кластером контейнерів. Оркестрація дозволяє забезпечити ефективний розподіл навантаження між контейнерами, високий рівень доступності та відмовостійкості, а також забезпечити комунікацію між контейнерами.

Оркестрація контейнерів виконується за допомогою відповідних платформ та складається з наступних етапів:

– визначення конфігурації сервісів, що будуть запущені в контейнерах;

– налаштування мережі для ефективної взаємодії між контейнерами;

– запуск процедури оркестрації;

– моніторинг стану системи.

Архітектурні патерни проектування

Для створення ефективної архітектури важливо вірно використовувати різноманітні патерни проектування. Для мікросервісної архітектури виділяють ряд важливих патернів, що впливають на різні аспекти системи. Першим важливим патерном є вимикач.

Вимикач (Circuit Breaker) – це патерн проектування, що використовується в мікросервісній архітектурі з метою покращення стабільності роботи та стійкості системи. Основною метою його використання є запобігання виклику сервісів, що не можуть відправити відповідь.

Принцип роботи даного патерну передбачає наявність трьох станів для кожного мікросервіса, що можуть накладати обмеження:

– першим є закритий стан, що передбачає, що всі запити до мікросервіса є успішними, а сервіс працює коректно;

– другим є відкритий стан. Він призначався мікросервісу у випадках появи різних збоїв, наприклад, занадто великий час очікування відповіді, що допомагає запобігти додатковому навантаженню на систему.

– третій стан називається напіввідкритим. Мікросервіс переходить у даний стан коли він знаходиться у відкритому стані певний період часу. Він дозволяє виконання обмеженої кількості запитів для перевірки коректності роботи сервісу. Якщо збої тривають, мікросерсів знову переходить у відкритий стан, а в іншому випадку у закритий стан.

Таким чином окремі мікросервіси, у яких виникають збої будуть обмежуватись, але всі інші функціональні можливості системи працюють коректно.

Ще одним важливим патерном проектування є шлюз (API Gateway). Шлюз – це патерн проектування, який передбачає наявність сервісу, що виступає в ролі єдиного можливого входу всіх зовнішніх запитів до мікросервісів, що дозволяє забезпечити маршрутизацію запитів, обробку помилок, безпеку даних та об'єднання відповідей від кількох сервісів в одну. На рисунку 3 наведено схему використання шлюзу.

Окрім виконання своєї основної задачі шлюз також можна використовувати для виконання додаткових функцій, наприклад, аутентифікацію користувачів, логування та моніторинг, визначення списку дозволених та заблокованих IP-адрес.

Однією із ключових проблем у мікросервісній архітектурі є реалізація розподілених операцій, що охоплюють відразу декілька сервісів, що може бути складно особливо у випадках наявності окремих баз даних для кожного сервісу. Для вирішення цієї проблеми розробники використовують патерни взаємодії сервісів, що розподіляють операцію на декілька локальних транзакцій, прикладом якого є патерн «Sega».

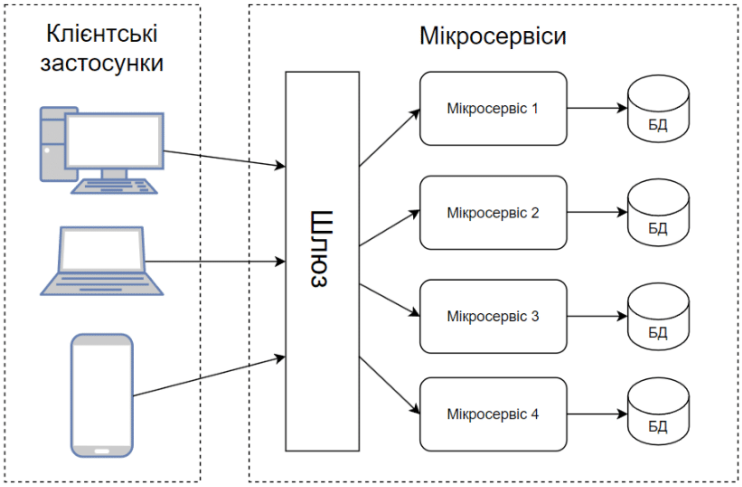

Рисунок 3. Схема використання шлюзу

Його сенс полягає в тому, що одна велика транзакція розбивається на декілька локальних транзакцій всередині кожного мікросервіса, що у подальшому виконуються поетапно. Важливим аспектом даного патерна є досягнення атомарності на кожному кроці виконання транзакції, що дозволить уникнути будь-яких змін у системі та подальшого виконання транзакції при появі помилки. Схема роботи патерна «Sega» наведено на рисунку 4.

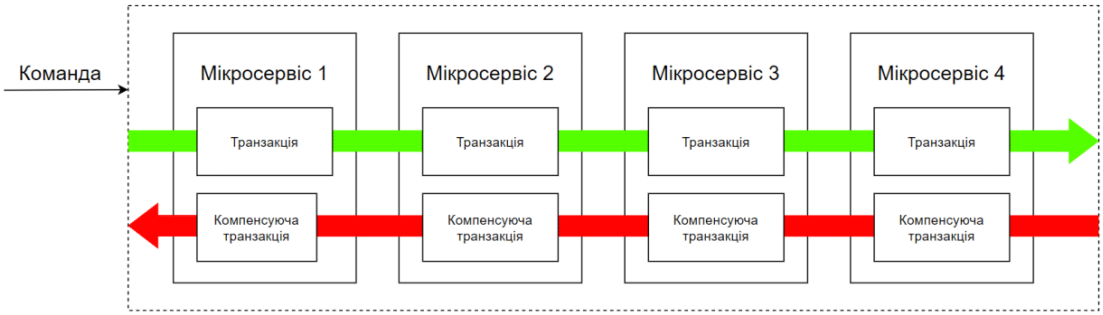

Рисунок 4. Схема використання шлюзу

Тестування мікросервісів

Оскільки мікросервісна архітектура є складною за рахунок її розподілу на незалежні мікросервіси, надзвичайно важливим аспектом розробки є тестування мікросервісів як окремо, так і у взаємодії з іншими. З цією метою розробники програмного забезпечення часто використовують процес безперервної інтеграції, що передбачає автоматизацію збірки та тестування системи кожного разу, коли до неї вносяться будь-які зміни.

Для швидкого тестування найнижчих рівнів та перевірки їхньої логіки застосовуються юніт-тести, що мають обмежену область застосування і прив'язуються до конкретних функцій та методів в межах окремого мікросервісу.

Наступним важливим видом тестів є інтеграційне тестування. У зв'язку з тим, що кожен мікросервіс виконує певну частину завдань важливим аспектом є забезпечення їхньої коректної взаємодії. Для цього створюються тестові сценарії, що передбачають взаємодію між різними мікросервісами, проводиться їх запуск та аналіз результатів.

Ще одним важливим та складним типом тестування є наскрізне тестування, з допомогою якого здійснюється перевірка всієї архітектури, що включає: інтерфейси застосунків, бізнес логіку та обмін даними. Головними недоліками даного тестування є те, що воно повільне, вся система повинна бути розгорнута повністю, а інтерпретація результатів невдалого тестування є складним і часто потребує проведення додаткових ручних тестів.

Висновки

На основі проведеного аналізу мікросервісної архітектури та її різних аспектів можна стверджувати, що практика останніх років демонструє позитивний розвиток архітектури, її методів реалізації, технологій та патернів проектування, хоча недоліки її використання досі мають суттєвий вплив на всю систему. У зв'язку з цим перед фінальним затвердженням архітектури варто додатково перевірити доцільність використання мікросервісної архітектури, переглянути бізнес-функції майбутньої системи та визначити можливості для потенційного розвитку.

На початкових етапах розробки мікросервісної архітектури надзвичайно важливим є визначення майбутніх мікросервісів, що будуть розроблені у майбутньому. Для цього варто використовувати підхід «domain-driven design» для розбиття домени, що описують певну діяльність чи ряд завдань, а використання даного підходу у зв'язці з граничними контекстами допоможе визначити відповідальність кожної моделі та спростити систему.

Сучасні технології, такі як, контейнеризація та оркестрація контейнерів демонструють, що мікросервіси можуть швидко розгортатись на різних системах та можливість відносно простого управління ними, а сучасні патерни проектування можуть вирішити ряд проблем, що наявні у мікросервісній архітектурі, але не позбутись їх повністю, покращити взаємодію між мікросервісами та підвищити надійність системи.

Тестування у мікросервісній архітектурі є одним із найважливіших етапів розробки програмного забезпечення, адже при наявності великою кількості мікросервісів неможливо визначити чи будуть вони коректно взаємодіяти між собою. З даних причин тестування є комплексним процесом, що включає в себе різні види тестування, наприклад, юніт-тести, інтеграційне та наскрізне шифрування.

Всі отримані дані будуть використані для кращого розуміння сучасних практик розробки мікросервісної архітектури, визначення переваг, недоліків та доцільності їхнього використання.

Література

1. Söylemez, Mehmet ; Tekinerdogan, Bedir ; Tarhan, Ayça Kolukısa. / Challenges and Solution Directions of Microservice Architectures : A Systematic Literature Review. In: Applied Sciences (Switzerland). 2022
2. Спасітєлєва, Світлана Олексіївна та Чичкань, І. та Шевченко, Світлана Миколаївна та Жданова, Юлія Дмитрівна (2023) Розробка безпечних контейнерних застосунків з мікросервісною архітектурою Кібербезпека: освіта, наука, техніка, 1 (21). с. 193-210.
3. Pahl, C., & Jamshidi, P. (2016). Microservices: a systematic mapping study. In Proceedings of the 6th International Conference on Cloud Computing and Services Science (CLOSER 2016) (pp. 137-146)
4. J. Heikkinen, Serverless and microservice architecture in modern software development, JAMK University of Applied Sciences, 2023.

References

1. Söylemez, Mehmet ; Tekinerdogan, Bedir ; Tarhan, Ayça Kolukısa. / Challenges and Solution Directions of Microservice Architectures : A Systematic Literature Review. In: Applied Sciences (Switzerland). 2022
2. Spasitielieva, Svitlana Oleksiivna ta Chychkan, I. ta Shevchenko, Svitlana Mykolaivna ta Zhdanova, Yuliia Dmytrivna (2023) Rozrobka bezpechnykh konteinernykh zastosunkiv z mikroservisnoiu arkhitekturoiu Kiberbezpeka: osvita, nauka, tekhnika, 1 (21). s. 193-210.
3. Pahl, C., & Jamshidi, P. (2016). Microservices: a systematic mapping study. In Proceedings of the 6th International Conference on Cloud Computing and Services Science (CLOSER 2016) (pp. 137-146)
4. J. Heikkinen, Serverless and microservice architecture in modern software development, JAMK University of Applied Sciences, 2023.

ДОДАТОК В

(обов’язковий)

ПРЕЗЕНТАЦІЙНІ МАТЕРІАЛИ

Рисунок В.1 – Слайд 1

Рисунок В.2 – Слайд 2

Рисунок В.3 – Слайд 3

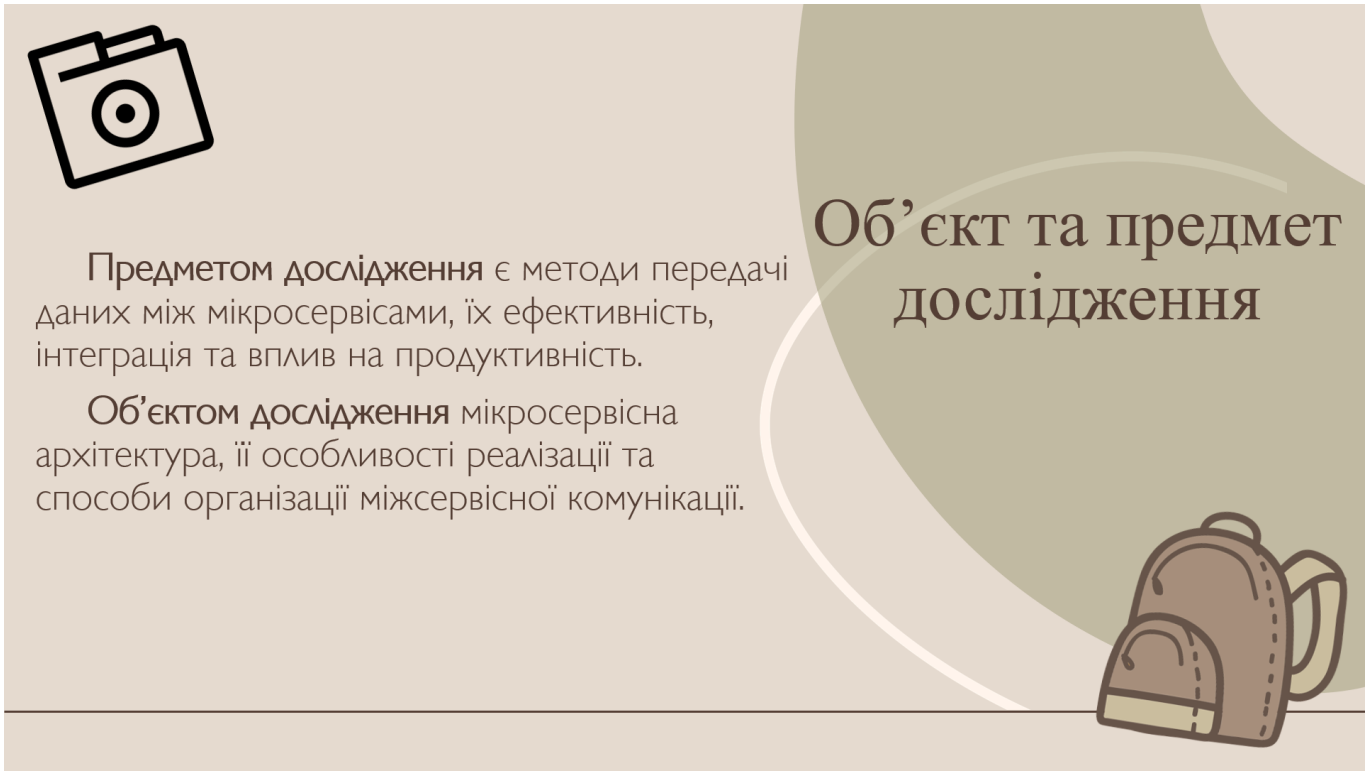

Рисунок В.4 – Слайд 4

Рисунок В.5 – Слайд 5

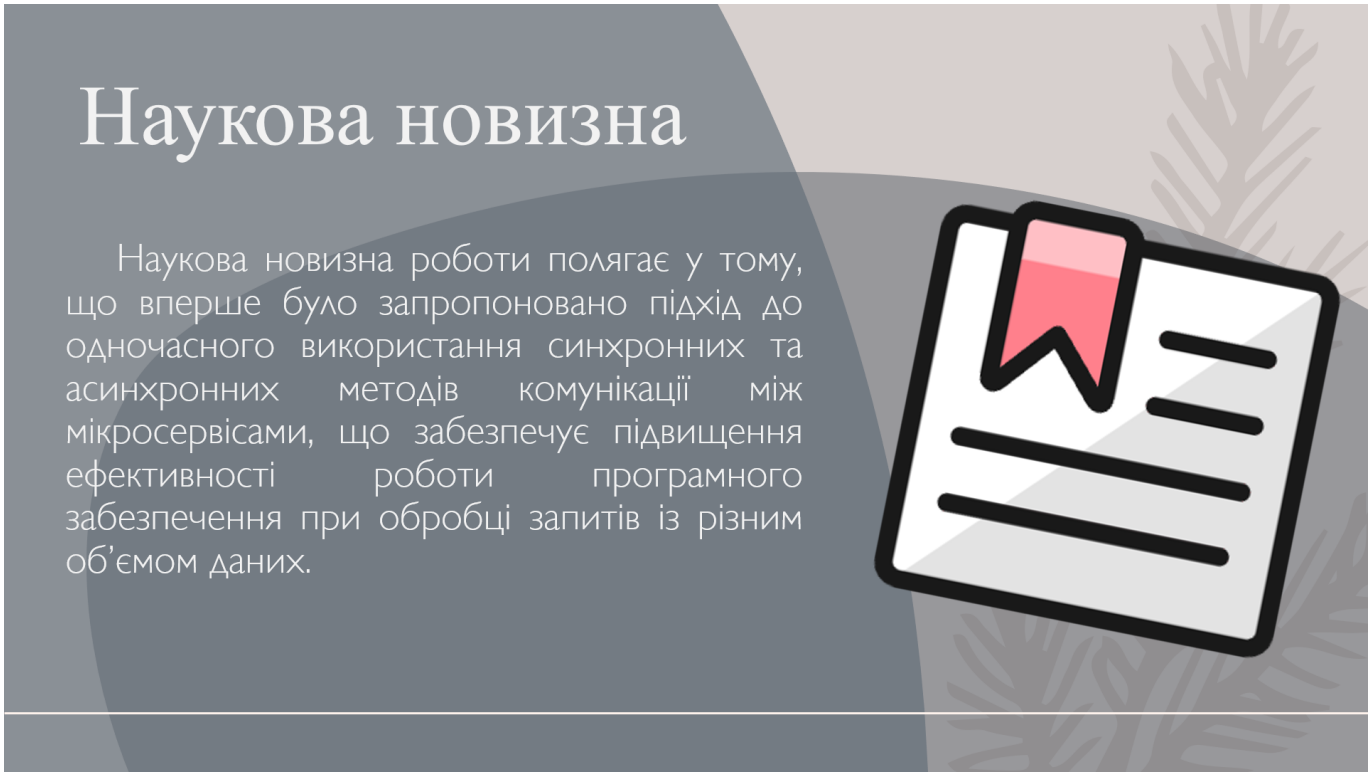

Рисунок В.6 – Слайд 6

Рисунок В.7 – Слайд 7

Рисунок В.8 – Слайд 8

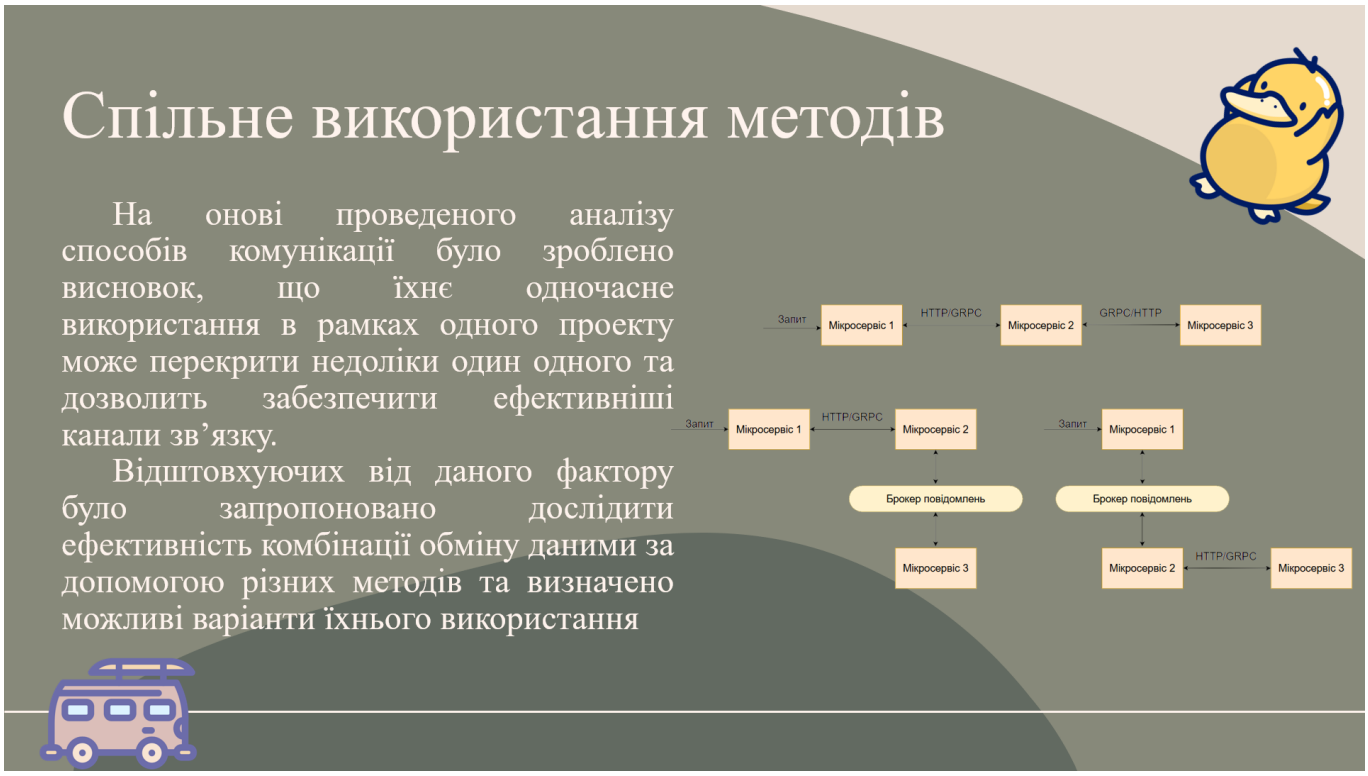

Рисунок В.9 – Слайд 9

Рисунок В.10 – Слайд 10

Аналіз результатів

Першим було проведено два тестування ефективності комунікацій мікросервісів для встановлення впливу способу розгортання на їхню роботу.

Середнє значення різниці становить 3.23%, що вказує на мінімальну різницю у швидкості роботи розгорнутих систем.

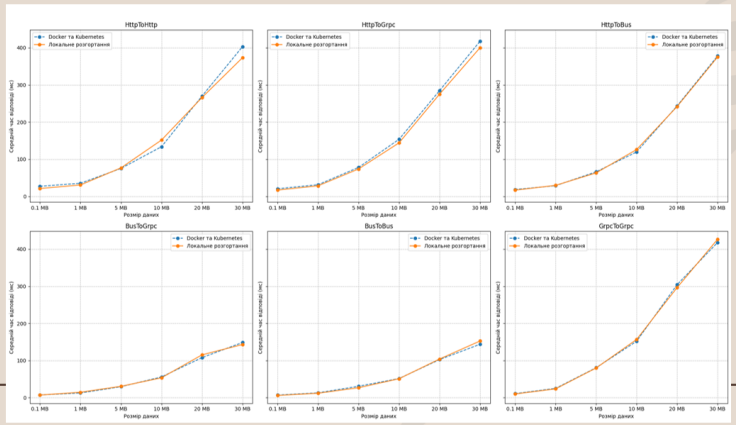

Рисунок В.11 – Слайд 11

Аналіз результатів

Наступне тестування передбачало оцінку швидкості роботи різних методів комунікації при обміні ідентичним розміром даних.

Найкращі результати були отримані при використанні подвійного брокера та у комбінації з GRPC.

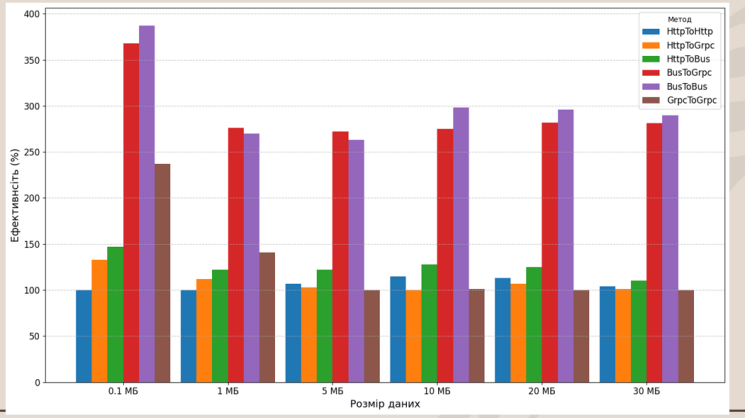

Рисунок В.12 – Слайд 12

Аналіз результатів

Останнє тестування передбачало оцінку швидкості роботи комбінацій методів комунікації при обміні різним об'ємом даних.

Результати показали подібний до попереднього тестування, проте вдалось встановити, що у таких комбінаціях брокер повідомлень краще підходить для роботи з великими об'ємами інформації.

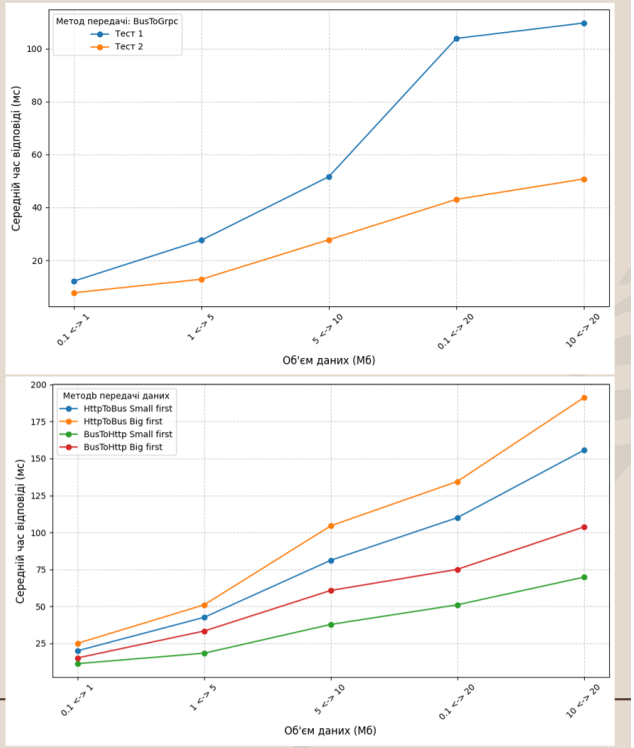

Рисунок В.13 – Слайд 13

Публікації

Для даної роботи було опубліковано одну статтю:

ПРАВОРСЬКА, Н.; ГРИГА, С. МЕТОДИ РЕАЛІЗАЦІЇ МІКРОСЕРВІСНИХ АРХІТЕКТУР: ПЕРЕВАГИ ТА НЕДОЛІКИ, ВПРОВАДЖЕННЯ ТА ТЕСТУВАННЯ ПРИ РОЗРОБЦІ ПРОГРАМНИХ ЗАСТОСУНКІВ. *Вісник Хмельницького національного університету. Серія: Технічні науки* **2024**, *335* (3(1), 404-408.

14

Рисунок В.14 – Слайд 14

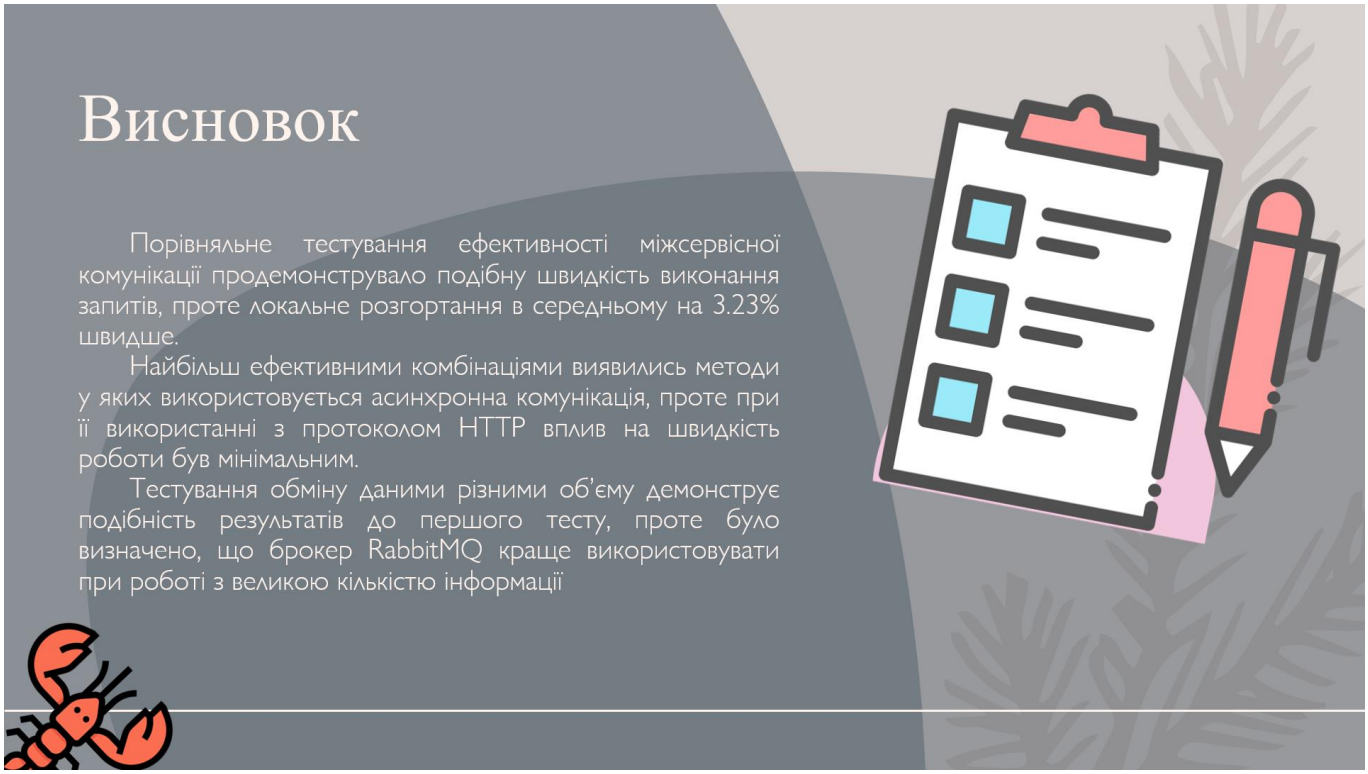

Рисунок В.15 – Слайд 15

Рисунок В.16 – Слайд 16

Завідувачу кафедри інженерії програмного
забезпечення проф. Леоніду БЕДРАТЮКУ
здобувача вищої освіти
Сергія ГРИГИ
факультет ІТ, 2 курс, група ІПЗм-23-1

ЗАЯВА

З правилами чинного Положення про систему забезпечення академічної доброчесності в Хмельницькому національному університеті, згідно з яким виявлення плагіату є підставою для відмови в допуску кваліфікаційної роботи до захисту і застосування заходів дисциплінарної та академічної відповідальності, ознайомлений. Про використання програмно-технічних засобів для перевірки кваліфікаційних робіт здобувачів вищої освіти на плагіат оповіщений та надаю свою згоду на обробку й збереження університетом моєї роботи в інституційному репозитарії університету.

Також надаю університету право на передачу моєї роботи для обробки та збереження в базах даних програмно-технічних засобів (StrikePlagiarism та Anti-Plagiarism) та використання роботи для виявлення плагіату в інших роботах, які перевіряються програмно-технічними засобами та користувачами, що мають доступ до цих програмно-технічних засобів, виключно в обмежених цілях для виявлення плагіату в текстах робіт.

Робота для перевірки університетом надається в друкованому та електронному варіанті. Електронна версія моєї роботи збігається (ідентична) з друкованою.

23.11.2024
дата

підпис

Mon Nov 25 23:02:59 EET 2024, Форкун Юрій Вікторович, Хмельницький національний університет, ХНУ

Anti-Plagiarism v-15.257

Максимальне співпадіння з одним документом 1.0%

Словники перевірки: en_US, ru_RU, ua_UA. **Помилок в документах: 12%**

ID: 149944 Назва: КВАЛІФІКАЦІЙНА РОБОТА Методи реалізації мікросервісних архітектур при розробці програмних застосунків: впровадження та тестування Додано в БД: 2024-11-25 Автора: Грига С. А. Керівники: канд. пед. наук, доцент Н. І. Праворська Консультанти: Опоненти:	Документ		Сумарний збіг по Базі Даних	
	Символи	Лексеми	Символи	Лексеми
	137865	988	2948 (2%)	38 (4%)

Джерело плагіату

ID	Опис	Наявність плагіату в документі	
		Символи	Лексеми

Протокол аналізу звіту подібності науковим керівником

Заявляю, що я ознайомився (-лась) з Повним звітом подібності, який був згенерований Системою виявлення і запобігання плагіату щодо роботи:

Автор: Грига С.А.

Співавтор:

Назва: МКР Методи реалізації мікросервісних архітектур при розробці програмних застосунків: впровадження та тестування

Науковий керівник: канд. пед. наук, доцент Н. І. Праворська

Підрозділ: Кафедра інженерії програмного забезпечення

Коефіцієнт подібності 1:1.6%

Коефіцієнт подібності 2:0.3%

Мікропробіли: 0

Заміна букв: 2

Інтервали: 0

Білі знаки: 0

Дата створення звіту: 2024-11-28 02:04:42.0

Після аналізу Звіту подібності констатую наступне:

☑ **Запозичення, виявлені в роботі є законними і не є плагіатом. Рівень подібності не перевищує допустимої межі. Таким чином робота незалежна і приймається.**

☐ **Запозичення не є плагіатом, але перевищено граничне значення рівня подібностей. Таким чином робота повертається на доопрацювання.**

☐ **Виявлено запозичення і плагіат або навмисні текстові спотворення (маніпуляції), як передбачувані спроби укриття плагіату, які роблять роботу невідповідною вимогам законодавства (Ст. 32. ЗУ Про вищу освіту, пункт 3.1, Ст. 42. ЗУ Про освіту) та вимог НАЗЯВО (Критерій 5), а також кодексу етики і процедурам. Таким чином робота не приймається.**

Обгрунтування:

28.11.2024
Дата

(Горицун К.В.)
експерт

ХМЕЛЬНИЦЬКИЙ НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ

РЕЦЕНЗІЯ НА КВАЛІФІКАЦІЙНУ РОБОТУ
освітнього ступеня «магістр»

Здобувач Грига Сергій Андрійович

Тема Методи реалізації мікросервісної архітектури при розробці програмних застосунків: впровадження та тестування

Спеціальність 121 «Інженерія програмного забезпечення»

Обсяг кваліфікаційної роботи:

Кількість сторінок кваліфікаційної роботи 94

1.Короткий зміст роботи та прийнятих рішень У кваліфікаційній роботі було проведено аналіз предметної області за обраною темою, проаналізовано ряд сучасних досліджень з визначенням проблемних місць і невирішених питань. На основі проведеного аналізу було запропоновано підхід до одночасного використання синхронних та асинхронних методів комунікації між мікросервісами для підвищення ефективності роботи програмного забезпечення. Встановлено, що за певних умов комбінація різних методів комунікації можуть бути ефективнішими із збереженням їхніх переваг використання.

2. Висновок про відповідність роботи поставленому завданню Кваліфікаційна робота освітнього ступеня «магістр» повністю відповідає поставленим вимогам та завданням.

3. Характеристика виконання кожного розділу роботи, ступінь використання останніх досягнень науки і техніки і передових методів роботи: У вступі було обґрунтовано актуальність теми роботи, формулюється мета та завдання дослідження, описується наукова новизна. У першому розділі було проведено аналіз обраної предметної області та розглянуто останні наукові дослідження на основі чого була здійснена постановка задачі. У другому розділі було досліджено сучасні методи міжсервісної комунікації за допомогою синхронних та асинхронних методів та способами розгортання системи на основі чого було визначено ряд підходів для проведення досліджень різних методів комунікації. У третьому розділі було проведено проектування програмного забезпечення, що включало визначення вимог до розроблюваного програмного забезпечення та його структури разом із вибором технологій та інструментів, що необхідні для виконання поставлених завдань. У четвертому розділі було розглянуто основні моменти, що стосувались розробки програмного забезпечення, розглянуто сценарії тестування та на основі проведеного аналізу результатів визначено ефективні комбінації методів комунікації.

4. Позитивні сторони роботи Кваліфікаційна робота містить низку інноваційних рішень, зокрема, було доведено, що обрані способи розгортання системи мінімально впливають на ефективність комунікації мікросервісів, а також визначено ряд комбінацій методів міжсервісної комунікації, що є більш ефективними.

5. Негативні сторони роботи У роботі розглядається три основних метода комунікації та декілька варіантів їхніх комбінацій. Можливо, було б доцільним розглянути ширший набір методів комунікації або альтернативні способи їхньої реалізації.

6. Оцінка графічного оформлення та пояснювальної записки роботи Графічне оформлення виконане відповідно до теми кваліфікаційної роботи з використанням рисунків та діаграм, що доповнюють текст. Пояснювальна записка оформлена з дотриманням всіх вимог та чинних стандартів.

7. Відгук про роботу в цілому В цілому дана кваліфікаційна робота заслуговує на позитивну оцінку. Матеріал роботи структурований, чіткий та послідовний, що дозволяє швидко розуміти викладену інформацію. Графічний матеріал доповнює роботу та наочно демонструє ефективність роботи обраних рішень та отриманих результатів.

8. Інші зауваження

9. Оцінка кваліфікаційної роботи Розглянувши всі позитивні та негативні сторони даної кваліфікаційної роботи можна зробити висновок, що вона заслуговує оцінки «відмінно».

РЕЦЕНЗЕНТ (прізвище, ім'я, по-батькові, посада, місце роботи) Мартинюк Валерій Володимирович, зав. каф. АКІТ та Р ХНУ

«27» 11 2024 р.

(підпис)

РІШЕННЯ ЕКСПЕРНОЇ КОМІСІЇ
КАФЕДРИ ІНЖЕНЕРІЇ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ
ПРО ДОПУСК КВАЛІФІКАЦІЙНОЇ РОБОТИ ДО ЗАХИСТУ

Підтверджуємо ознайомлення з результатами звіту/звітів подібності щодо роботи, продукованими програмно-технічним засобом(ами) перевірки текстів на плагіат.
Назва: «Методи реалізації мікросервісних архітектур при розробці програмних застосунків»
Автор: Грига Сергій Андрійович
Спеціальність: 121 – Інженерія програмного забезпечення
Освітня програма: Освітньо-професійна програма «Інженерія програмного забезпечення»
Науковий керівник: Правоська Наталія Іванівна канд. пед. наук, доцент
Після аналізу звіту подібності зроблено такий висновок:

№	Висновок	Позначка про відповідність
1	Запозичення, виявлені в роботі, є законними і не є плагіатом. Робота приймається до захисту.	**відповідає**
2	Виявлені запозичення не є плагіатом, розміщені у розділах, які не описують безпосередньо авторське дослідження, але кількість цитат перевищує обсяг, виправданий поставленою метою роботи. Робота приймається до захисту, але має бути відкоригована. Відкоригований варіант має бути поданий на кафедру за два дні до захисту, разом із заявою щодо самостійності виконання письмової роботи та ідентичності друкованої й електронної версії роботи.	
3	Виявлені запозичення не є плагіатом, але частково розміщені у розділах, які описують безпосередньо авторське дослідження, а кількість цитат перевищує обсяг, виправданий поставленою метою роботи. В зв'язку з цим мета роботи та поставлені завдання не були досягнені. Робота може бути допущена до захисту (наступного року) після того, як буде відкоригована та доопрацьована і успішно пройде повторну перевірку на академічний плагіат.	
4	Робота містить навмисні текстові спотворення, передбачувані спроби укриття запозичень або інші прояви академічного плагіату. Робота містить фабрикацію або фальсифікацію даних. Робота не допускається до захисту.	
5	Інше:	

Підтвердження:
Запозичення, виявлені у роботі, є законними і не є плагіатом, оскільки:
1) у тексті кваліфікаційної роботи системою перевірки на плагіат StrickePlagiarism виявлено схожість з деякими документами у частині загальновживаних обов'язкових словосполучень у стандартних бланках (титулка, бланк завдання), у структурі змісту, у назвах розділів/підрозділів, у назвах публікацій переліку джерел посилання тощо;
2) в якості запозичень системою StrickePlagiarism було зафіксовано деякі послідовності вихідного коду і посиланання на бібліотеки, які є стандартними мовними конструкціями програмування та не можуть розглядатися як об'єкт авторських прав і, відповідно, їх порушення;
3) запозичення, виявлені в тексті роботи, є фрагментарними або мають належним чином оформленні посилання;
4) виявлені модифікації тексту не впливають на відсоток схожості.
Максимальний обсяг запозичень, визначений системою Anti-Plagiarism, складає 1 %. За системою StrikePlagiarism коефіцієнт подібності 1 складає 1,6%, коефіцієнт подібності 2 складає 0.3%.

Дата 28.11.2024

Завідувач кафедри ІПЗ _____ Леонід БЕДРАТЮК
Гарант освітньої програми _____ Оксана ЯШИНА
Керівник кваліфікаційної роботи _____ Наталія ПРАВОРСЬКА