МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
НАЦІОНАЛЬНИЙ АВІАЦІЙНИЙ УНІВЕРСИТЕТ
ФАКУЛЬТЕТ КІБЕРБЕЗПЕКИ, КОМП'ЮТЕРНОЇ ТА ПРОГРАМНОЇ ІНЖЕНЕРІЇ
Кафедра комп'ютерних систем та мереж

ДОПУСТИТИ ДО ЗАХИСТУ
Завідувач кафедри
_____ Жуков І.А
"_____"_____ 2020 р.

ДИПЛОМНА РОБОТА
(ПОЯСНЮВАЛЬНА ЗАПИСКА)

ВИПУСКНИКА ОСВІТНЬОГО СТУПЕНЯ "МАГІСТР"
ЗА СПЕЦІАЛЬНІСТЮ 123 «КОМП'ЮТЕРНА ІНЖЕНЕРІЯ»

Тема: «Технологія оптимізації асинхронного процесінгу в розподілених інфокомунікаційних системах»

Виконавець:студент,КС-101Мз, Нестеровський Владислав Михайлович _____
Керівник: Гузій Микола Миколайович _____

Нормоконтролер: _____ Андрєєв Володимир Ілліч

Засвідчую, що у магістерській роботі немає
запозичень праць інших авторів
без відповідних посилань
Студент _____ В.М.Нестеровський

Київ 2020

НАЦІОНАЛЬНИЙ АВІАЦІЙНИЙ УНІВЕРСИТЕТ

Факультет кібербезпеки, комп'ютерної та програмної інженерії
Кафедра комп'ютерних систем та мереж
Освітнього ступеня магістр
Напрям 123 Комп'ютерна інженерія

ЗАТВЕРДЖУЮ
Завідувач випускової кафедри
Жуков І.А.
"___" _____ 2020 р

ЗАВДАННЯ
на виконання дипломної роботи студента
Нестеровського Владислава Михайловича

1. Тема роботи: «Технологія оптимізації асинхронного процесінгу в розподілених інфокомунікаційних системах» затверджена наказом ректора від « 25 » вересня 2020 р. № 1793/ст.
2. Термін виконання роботи: з 05.10.2020 р. до 31.12.2020 р.
3. Вихідні дані до роботи: Розробка програмного забезпечення та технологій для хмарних платформ. Використання сучасних підходів проектування та створення веб-сервісів та хмарних додатків. Вимоги до змісту та оформлення дипломних проектів студентів НАУ.
4. Зміст пояснювальної записки: Аналіз предметної області «Обробка даних в розподілених інфокомунікаційних системах». Аналіз способів оптимізації в розподілених інфокомунікаційних системах. Структура та архітектура застосунку для демонстрації оптимізації асинхронного процесінгу в розподілених інфокомунікаційних системах. Прототип застосунку для демонстрації оптимізації асинхронного процесінгу в розподілених інфокомунікаційних системах.
5. Перелік обов'язкових слайдів презентації:
1) Тема, виконавець, керівник.
2) Існуючи методики, аналіз недоліків, постановка завдання.
3) Вимоги до програмного засобу.
4) Структура засобу, діаграма класів
5) Інтерфейс програмного засобу
6) Демонстрація роботи прототипу засобу

6. Календарний план-графік

№ п/п	Етапи виконання дипломної роботи	Термін виконання етапів	Примітка
1	Узгодження технічного завдання з керівником роботи	25.09.20	
2	Підбір та вивчення науково-технічної літератури за темою дипломної роботи	26.09.20 – 15.10.20	
3	Опрацювання матеріалу щодо хмарних технологій та розподілених систем	16.10.20 – 31.10.20	
4	Аналіз впроваджених технологій	01.11.20 – 12.11.20	
5	Розробка практичної частини	13.11.20 – 01.12.20	
6	Оформлення пояснювальної записки	02.11.20 – 07.12.20	
7	Оформлення графічних матеріалів роботи та представлення роботи на кафедру	08.12.20 – 10.12.20	

7. Дата видачі завдання «25» вересня 2020 р.

Керівник дипломної роботи _____ Гузій М.М.
(підпис)

Завдання прийняв до виконання _____ Нестеровський В.М.
(підпис студента)

РЕФЕРАТ

Пояснювальна записка до дипломної роботи «Технологія оптимізації асинхронного процесінгу в розподілених інфокомунікаційних системах»: 75 сторінок, 35 рисунків, 41 використаних джерел.

ІНФОКОМУНІКАЦІЙНА СИСТЕМА, АСИНХРОННИЙ ПРОЦЕСІНГ, МОДЕЛЮВАННЯ, СИСТЕМНИЙ ДИЗАЙН, РОЗПОДІЛЕНА ЧЕРГА, ХМАРНІ ОБЧИСЛЕННЯ, МІКРОСЕРВІСИ.

Об'єкт дослідження – розподілені інфокомунікаційні системи.

Предмет дослідження – технології оптимізації розподілених інфокомунікаційних систем.

Мета дипломної роботи – дослідження асинхронного процесінгу в розподілених інфокомунікаційних системах та розробка технології оптимізації нефункціональних властивостей розподілених систем.

Методи дослідження – методи системного аналізу, аналізу нефункціональних вимог до архітектури, методи оптимізації, методи математичної статистики, методи математичного моделювання.

Технічні та програмні засоби – середовище проектування та програмування MS Visual Studio 2019, GitBash консольна утиліта, Azure DevOps для контролю версій та збірки та розгортування веб-застосунків.

Результати роботи рекомендується використовувати для покращення показників якості системи (QoS), таких як затримка відповіді, кількість одночасних з’єднань, та ін.

Прогнозні припущення щодо розвитку об'єктів розроблення – розроблювана технологія може бути використана у майбутньому шляхом застосування принципів описаних в цій роботі для оптимізації показників в розподілених інфокомунікаційних системах.

Актуальність теми полягає в переході на мікросервісну архітектуру та відкритті нових можливостей асинхронної обробки даних.

ABSTRACT

Explanatory note to the diploma project "Technology for optimization of asynchronous processing in distributed infocommunication systems": 75 pages, 35 figures, 41 sources used.

INFOCOMMUNICATION SYSTEM, ASYNCHRONOUS PROCESSING, MODELING, SYSTEM DESIGN, CLOUD COMPUTING, DISTRIBUTED QUEUE, MICROSERVICES.

The object of study – distributed infocommunication systems.

The subject of study – distributed infocommunication system technologies.

The aim of the thesis – is to improve the existing technologies of asynchronous processing in distributed infocommunication systems and implement a technology for the optimization.

Method development – system analysis, mathematical models, optimization models.

Technical and software – PC running Windows 8 or 10; environment for object-oriented programming MS Visual Studio 2020, GitBash utility, Azure DevOps.

Results of the project recommended to use during the implementation of the distributed infocommunication systems.

Projected assumptions about the development of facilities – the developed software can be expanded in the future by connecting services to monitor the processes of information exchange of distributed systems.

ЗМІСТ

ПЕРЕЛІК ПРИЙНЯТИХ СКОРОЧЕНЬ

ГІІ – глобальна інформаційна інфраструктура

CDN – *Content Delivery Network* – мережа доставки (розповсюдження) контенту

ISP – *Internet Service Provider* – провайдер послуг Інтернету, Інтернет-провайдер.

NaaS – *Network as a service* – мережа як послуга

NFV– *Network Functions Virtualization* – віртуалізація мережевих функцій

OFDM – *Orthogonal Frequency-Division Multiplexing* – метод мультиплексування

OTT – *Over-The-Top Service* – доставка відео та аудіо без причетності ISP

PaaS - *Platform as a Service* – платформа забезпечення, як сервіс

PoP – *Point of Presence* – точка (операторської) присутності

TE – *Traffic Engineering* – управління трафіком

QoE – *Quality of Expirience* – якість сприйняття сервісу

QoS – *Quality of Service* – якість сервісу

SDN – *Software-defined Networking* – програмно-конфігурована мережа

ASCII – *American Standard Code for Information Interchange*

ASP – *Active Server Pages*

ACID – абревіатура 4 властивостей, що гарантують надійну роботу транзакцій бази даних.

CGI – *Common Gateway Interface*

RSS – Родина XML-форматів, що використовується для публікації та постачання інформації, що часто змінюється

REST – *Representational State Transfer* (підхід до архітектури мережевих протоколів, які забезпечують доступ до інформаційних ресурсів)

CLI – *Command-line interface* (текстовий інтерфейс користувача)

XML – *Extensible Markup Language* (стандарт побудови мов розмітки ієрархічно структурованих даних для обміну між різними додатками)

ВСТУП

Сучасні інфокомунікаційні системи все частіше змушені витримувати великі навантаження через постійно зростаючу кількість нових користувачів мережі Internet. Для того щоб інфокомуніцайні системи мали здатність витримувати вищі навантаження ніж може обробити один сервер була створена й запроваджена концепція розподілених інфокомунікаційних систем – систем в яких інфраструктурне та корисне навантаження оброблюють багато фізичних серверів. Проте такі системи мають як переваги так і нові виклики, які необхідно задовольняти.

Метою даної роботи є дослідження шляхів оптимізації обробки даних під нерівномірним навантаженням в розподілених інфокомунікаційних системах та розробка технології оптимізації нефункціональних властивостей розподілених систем, таких як затримка відповіді, кількість одночасно оброблених запитів, показники RPS.

Розподілені інфокомунікаційні системи фізично знаходяться на групах фізичних серверів - такі системи називаються кластерами, та об’єднуються по спільним рисам – типам навантаження, бізнес логіці та іншим. Однак для того щоб утримувати й керувати таким кластером треба спеціалісти й відповідна інфраструктура. Сучасним спобом вирішувати цю проблему є концепція хмари – форма оренди фізичної інфраструктури та спеціалістів в якій користувач платить тільки за використання інфраструктури й часу користувачів, економлячи при цьому на пошуку та наймі спеціалістів, та покупці відповідної інфраструктури.

Комбінація розподілених систем та хмарного середовища дозволяють легко та ефективно вирішувати проблеми типового онлайн веб застосунку таких як месенджери, онлайн магазини, та інші типи застосунків для яких характерна нерівномірність навантаження, пікові навантаження, та інші складні задачі балансування та оптимізації навантаження.

Аналіз та дослідження було проведено за допомогою методів системного аналізу та дизайну, таких як підходи до компоновки архітектури з використанням розподілених черг. Також були розглянуті причини та методи оптимізації нерівномірного навантаження на веб ресурси, способи запобігання відмови веб сервісу та балансування навантаження використовуючи хмарні технології.

Також були розглянуті існуючі технології які задають свої обмеження та сценарії використання, способи їх коректної експлуатації в рамках розроблюваної технології, та принципи якими можна керуватися під час використання даного засобу. В рамках цієї роботи була створена типовий веб сервіс який був створений використовуючи запропоновану архітектуру, та який реалізує логіку роботи месенджера (p2p користувацькі повідомлення). На його прикладі розглянуто сильні та слабкі сторони даної архітектури, заміряні ключові показники які було оптимізовано використанням технології, розробленої в рамках даної праці.

За основу яку необхідно оптимізувати було взято мікросервісну архітектуру, яка має великий список переваг, та є основною для найбільш навантажених систем. За допомогою мікросервісів код розбивається на незалежні сервіси, які працюють як окремі процеси. Вихідні дані однієї служби використовуються як вхідні дані для іншої в організації незалежних комунікаційних служб. Мікросервіси особливо корисні для підприємств, які не мають заздалегідь заданого уявлення про масив пристроїв, які підтримуватимуть його додатки. Будучи агностичними щодо пристроїв та платформ, мікросервіси дозволяють компаніям розробляти додатки, які забезпечують послідовну взаємодію з користувачами на різноманітних платформах, що охоплюють Інтернет, мобільні пристрої, Інтернет речей, носні пристрої та середовища для фітнесу. Масштабованість є ключовим аспектом мікросервісів. Оскільки кожна послуга є окремим компонентом, можна масштабувати одну функцію або службу без необхідності масштабувати всю програму. Критично важливі для бізнесу служби можуть бути розгорнуті на декількох серверах для підвищення доступності та продуктивності, не впливаючи на продуктивність інших служб.

Однією з основних оптимізацій розглянутих в даній роботі є пакетна обробка, яка дозволяє значно збільшити загальну швидкість обробки даних. У системі ініційоване пакетне завдання, яке являє собою блок коду, який групує всі повідомлення, виконує дії над кожним повідомленням, потім зберігає результати та потенційно надсилає оброблений результат до інших систем або черг. Ця функціональність особливо корисна при роботі з потоковим введенням або при розробці інтеграції даних "майже в реальному часі" між застосунками SaaS. Такі пакетні завдання дозволяють описати надійний процес, який автоматично розподіляє вихідні дані та зберігає їх у постійних чергах, що дозволяє обробляти великі масиви даних, забезпечуючи надійність. У випадку, якщо додаток передислоковано або система виходить з ладу, виконання завдання може відновитись у момент зупинки.

РОЗДІЛ 1
АНАЛІЗ ПРОЦЕСІНГУ В РОЗПОДІЛЕНИХ ІНФОКОМУНІКАЦІЙНИХ СИСТЕМАХ

1.1. Аналіз інфокомунікаційних та хмарних технологій

Хмарні обчислення - це надання різних послуг через Інтернет. Ці ресурси включають інструменти та додатки, такі як зберігання даних, сервери, бази даних, мережі та програмне забезпечення. Замість того, щоб зберігати файли на власному жорсткому диску або локальному запам'ятовуючому пристрої, хмарне сховище дозволяє зберігати їх у віддаленій базі даних. Поки електронний пристрій має доступ до Інтернету, він має доступ до даних та програмного забезпечення для його запуску. Хмарні обчислення є популярним варіантом для людей та бізнесу з ряду причин, включаючи економію витрат, підвищення продуктивності, швидкості та ефективності, продуктивності та безпеки.

Хмарні обчислення називаються такими, оскільки інформація, до якої здійснюється доступ, знаходиться віддалено в хмарі або віртуальному просторі. Компанії, що надають хмарні послуги, дозволяють користувачам зберігати файли та програми на віддалених серверах, а потім отримувати доступ до всіх даних через Інтернет. Це означає, що користувач не повинен знаходитись у певному місці, щоб отримати до нього доступ, що дозволяє користувачеві працювати віддалено. Хмарні обчислення забирають всю важку роботу, пов’язану з обробкою та обробкою даних, від пристрою, який в кишені до робочої станції.

Це також переносить всю цю роботу у величезні комп'ютерні кластери далеко у віртуальному просторі.

Інтернет стає хмарою, і дані робота та програми доступні з будь-якого пристрою, за допомогою якого можна підключитися до Інтернету в будь-якій точці світу.

Хмарні обчислення можуть бути як державними, так і приватними. Державні хмарні служби надають свої послуги через Інтернет за окрему плату. Натомість приватні хмарні сервіси надають послуги лише певній кількості людей. Ці послуги являють собою систему мереж, що постачають послуги, що розміщуються. Існує також гібридний варіант, який поєднує в собі елементи як державних, так і приватних послуг.

Існують різні типи хмар, кожна з яких відрізняється від іншої. Публічні хмари надають свої послуги на серверах та зберігання в Інтернеті. Вони експлуатуються сторонніми компаніями, які обробляють та контролюють все обладнання, програмне забезпечення та загальну інфраструктуру. Клієнти отримують доступ до послуг через облікові записи, доступ до яких має майже кожен. Приватні хмари зарезервовані для конкретної клієнтури, як правило, для одного бізнесу чи організації. Центр обслуговування даних фірми може розміщувати послугу хмарних обчислень. Багато приватних хмарних обчислень надаються у приватній мережі. Гібридні хмари - це, як випливає з назви, поєднання як державних, так і приватних послуг. Цей тип моделі надає користувачеві більшу гнучкість та допомагає оптимізувати інфраструктуру та безпеку користувача.

Хмарні обчислення - це не одна технологія, така як мікрочіп чи мобільний телефон. Швидше, це система, що складається переважно з трьох служб: програмне забезпечення як послуга (*SaaS*), інфраструктура як послуга (*IaaS*) та платформа як послуга (*PaaS*). Програмне забезпечення як послуга (SaaS) передбачає ліцензування програмного забезпечення для клієнтів. Ліцензії, як правило, надаються за моделлю оплати на час або на вимогу. Цей тип системи можна знайти в *Microsoft Office* 365.1 Інфраструктура як послуга (*IaaS*) передбачає метод доставки всього, від операційних систем до серверів та зберігання через підключення на основі *IP,*

як частину послуги за запитом. Клієнти можуть уникнути необхідності купувати програмне забезпечення або сервери, а замість цього закуповувати ці

ресурси в службі на замовлення, що передається підрядником. Популярні приклади системи IaaS включають *IBM Cloud* та Microsoft *Azure*.2 3 Платформа як послуга (*PaaS*) вважається найскладнішим із трьох шарів хмарних обчислень. *PaaS* має подібність із *SaaS*, головна відмінність полягає в тому, що замість доставки програмного забезпечення в Інтернеті, насправді це платформа для створення програмного забезпечення, яке доставляється через Інтернет. Ця модель включає такі платформи, як *Salesforce.com* та *Heroku*.

Хмарне програмне забезпечення пропонує компаніям з усіх секторів ряд переваг, включаючи можливість використання програмного забезпечення з будь-якого пристрою або через власний додаток, або через браузер. В результаті користувачі можуть переносити свої файли та налаштування на інші пристрої абсолютно безперебійно. Хмарні обчислення - це набагато більше, ніж просто доступ до файлів на декількох пристроях. Завдяки послугам хмарних обчислень користувачі можуть перевіряти свою електронну пошту на будь-якому комп'ютері і навіть зберігати файли за допомогою таких служб, як *Dropbox* та *Google Drive*. Послуги хмарних обчислень також дозволяють користувачам створювати резервні копії своєї музики, файлів та фотографій , переконавшись, що ці файли є негайно доступними у випадку аварії жорсткого диска. Він також пропонує великому бізнесу величезний потенціал економії коштів. Перш ніж хмара стала життєздатною альтернативою, компанії повинні були придбати, побудувати та підтримувати дорогі технології та інфраструктуру управління інформацією. Компанії можуть обміняти дорогі серверні центри та IT-відділи на швидкі підключення до Інтернету, де співробітники взаємодіють із хмарою в Інтернеті для виконання своїх завдань. Хмарна структура дозволяє людям економити місце для зберігання на своїх робочих столах або ноутбуках. Це також дозволяє користувачам швидше оновлювати програмне забезпечення, оскільки компанії, що виробляють програмне забезпечення, можуть пропонувати свої продукти через Інтернет.[1]

А не за допомогою більш традиційних, відчутних методів, що включають диски або флешки.

Наприклад, клієнти *Adobe* можуть отримати доступ до програм у своїй *Creative Cloud* через підписку на Інтернет. Це дозволяє користувачам легко завантажувати нові версії та виправлення своїх програм.

З усією швидкістю, ефективністю та нововведеннями, що поставляються з хмарними обчисленнями, природно існують ризики. Безпека завжди викликала велике занепокоєння у хмарі, особливо коли мова йде про конфіденційні медичні записи та фінансову інформацію. Хоча регламенти змушують служби хмарних обчислень посилювати свої заходи безпеки та дотримання вимог, це залишається постійним питанням. Шифрування захищає життєво важливу інформацію, але якщо цей ключ шифрування втрачено, дані зникають. Сервери, що обслуговуються компаніями, що займаються хмарними обчисленнями, також можуть стати жертвами стихійних лих, внутрішніх помилок та перебоїв з електропостачанням. Географічне охоплення хмарних обчислень скорочує в обох напрямках: відключення електроенергії в Каліфорнії може паралізувати користувачів у Нью-Йорку, а фірма в Техасі може втратити свої дані, якщо щось спричинить збій у постачальника послуг із штату Мен. Як і будь-яка інша технологія, існує крива навчання як для працівників, так і для менеджерів. Але, оскільки багато людей отримують доступ до інформації та маніпулюють нею через єдиний портал, ненавмисні помилки можуть переноситись на цілу систему.

1.2 Аналіз проблем в розподілених інфокомунікаційних системах в хмарному середовищі.

Масштабованість: масштабування є однією з головних проблем розподілених інфокомунікаційних систем. Проблема масштабування складається з такого виміру як комунікаційна здатність. Системи повинні бути спроектовані таким чином, щоб місткість могла бути збільшена зі збільшенням попиту на систему.

Неоднорідність: це важливе питання дизайну для розподілених інфокомунікаційних систем. Інфраструктура зв'язку складається з каналів різної ємності. Кінцеві системи володітиме різноманітними техніками презентації.

Представлення об'єктів та переклад: Вибір найкращої моделі програмування для розподілених об'єктів, таких як *CORBA, Java,* тощо є важливою проблемою.

Управління ресурсами: У розподілених системах, об’єкти, що складаються з ресурсів, розташовані в різних місцях. Маршрутизація є проблемою на мережевому рівні розподіленого системи та на рівні програми. Управління ресурсами в розподіленій системі буде взаємодіяти зі своєю неоднорідною природи.

Безпека та конфіденційність: Як застосувати політику безпеки до взаємозалежної системи є серйозним питанням у розподілених інфокомунікаційних системах. Оскільки розподілені системи мають справу з конфіденційними даними та інформація, тому система повинна мати надійний захист та вимірювання конфіденційності. Захист розподіленої системи активи, включаючи базові ресурси, сховище, зв’язок і інтерфейсу вводу-виводу, а також композитів вищого рівня ці ресурси, такі як процеси, файли, повідомлення, відображення *Windows* і більш складні об'єкти, є важливими питаннями в розподілених інфокомунікаційних системах.[2]

Прозорість: прозорість означає, до якої міри розподілена система повинна здаватися користувачеві єдиною системою? Розподілена система повинна бути розроблена для приховування складності системи більшою мірою.

Відкритість: відкритість означає, до якої міри повинна бути система розроблена із використанням стандартних протоколів для підтримки сумісності коли розробникам треба додати нові функції або замінити якусь підсистему в майбутньому. Щоб досягти цього, розподілена система повинна мати чітко визначені інтерфейси.

Якість обслуговування (*QoS*): Як визначити якість наданих послуг що надаються користувачам системи та який прийнятний рівень якості послуг,

що надається користувачам. Якість обслуговування сильно залежать від процесів, які будуть виділені процесори в системі, розподіл ресурсів, обладнання, адаптивність системи, мережі тощо. Хороша продуктивність, доступність та надійність необхідні репрезентації хорошої якості обслуговування.

Незважаючи на прогресивний стан системних досліджень, ми все ще не можемо остаточно визначити, коли ресурс вийшов з ладу. Стани нетермінальної відмови, такі як взаємоблокування та візантійська проблема (одночасний доступ до ресурсів), неймовірно важко виявити та вирішити. Крім того, у великих розподілених системах невеликі відмови можуть потенційно каскадувати через систему, перетворюючись на катастрофу.

Також одною з основних проблем є коректна робота веб сервісу під час пікових навантажень.

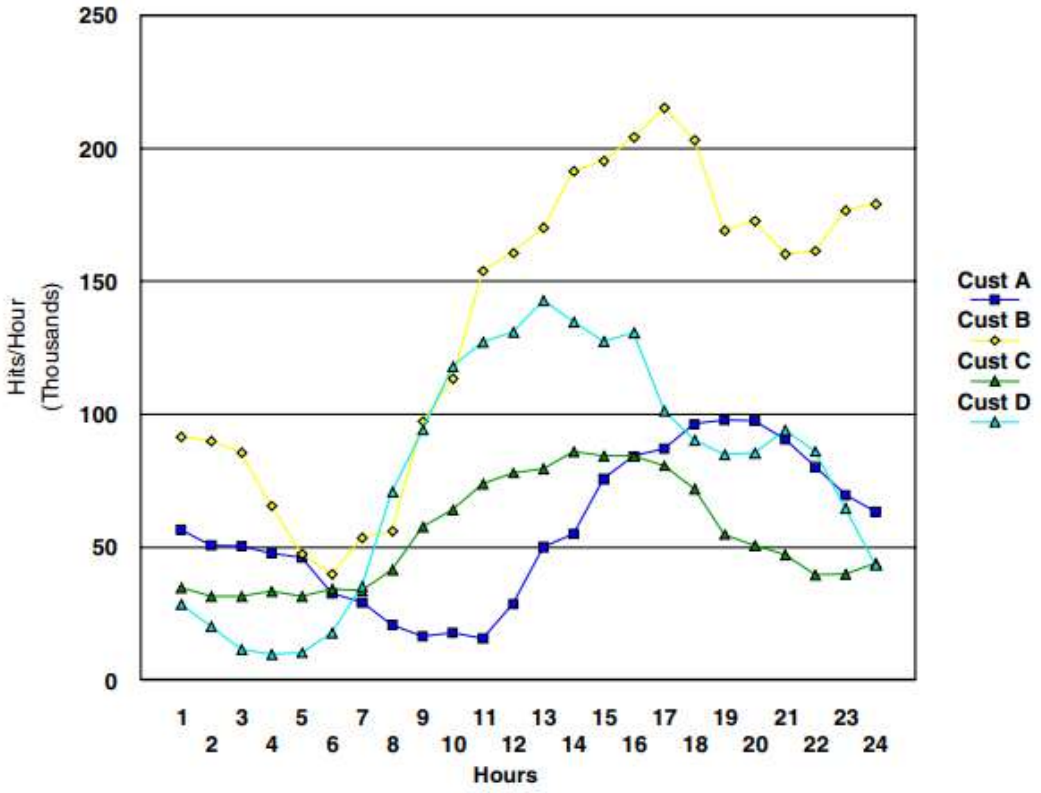

Рис. 1.1. Спрощена схема навантаження веб сервісу протягом доби [3]

Очевидно, що не всі несправності можна виявити та усунути, але в цій ситуації незрозуміло, що повинні робити системи, щоб справлятися з такого роду помилками.

З рис 1.1 очевидно що навантаження на веб ресурс протягом доби є нерівномірне, та є періоди коли навантаження сильно перевищує середній показник. Подібною є ситуація і протягом року:

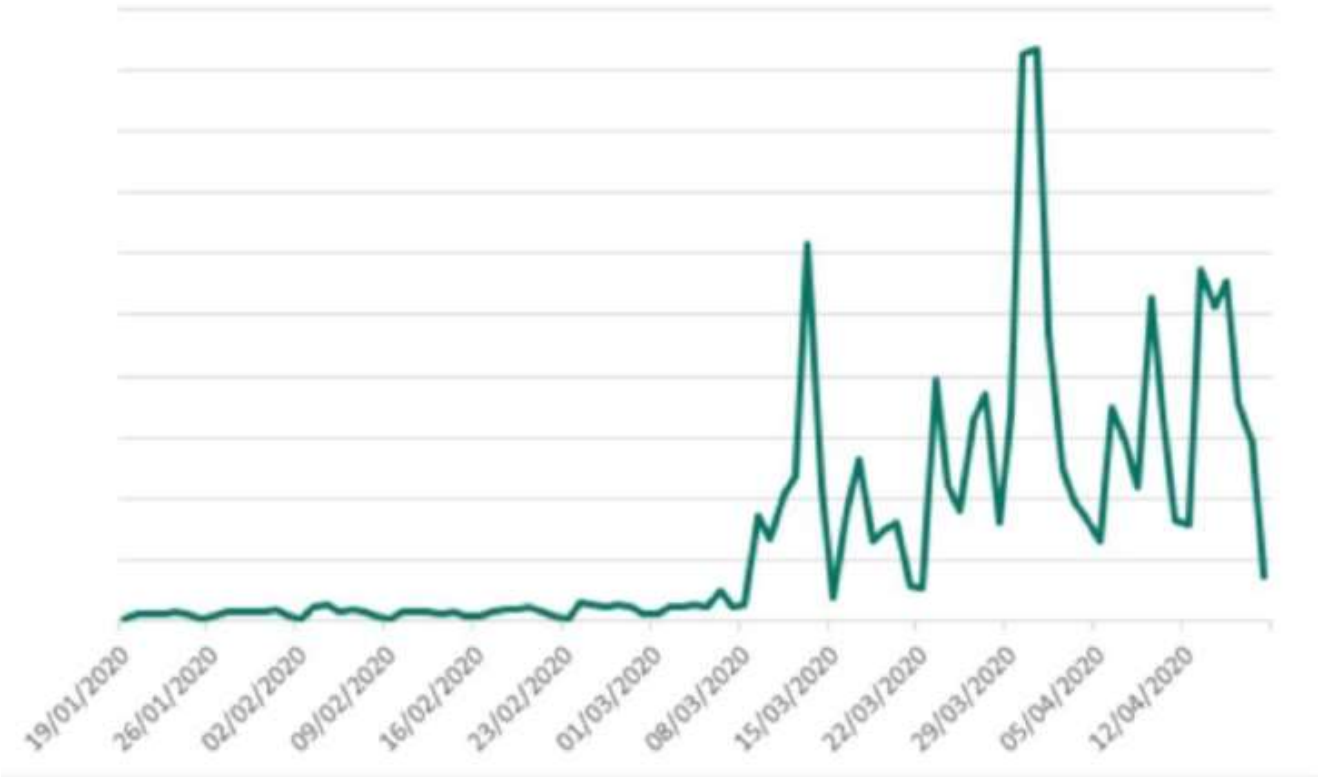

Рис. 1.2. Графік навантаження веб сервісу протягом пандемії *COVID*-19 [4]

Також однією з особливостей розподілених систем є їх асинхронна натура. Розподілена інфокомунікаційна система що складається з паралельних процесів є синхронною, якщо всі процеси виконуються з використанням одного і того ж годинника, тому процеси працюють із замкнутим кроком, і вона називається асинхронною, якщо кожен процес має свій незалежний годинник. Прикладами синхронних систем є певні великі централізовані багатопроцесорні комп'ютери та мікросхеми *VLSI*, що містять безліч окремих елементів паралельної обробки.

Прикладами асинхронних систем є розподілені комп'ютерні мережі та системи вводу-виводу для звичайних комп'ютерів. Фактичною проблемою в

синхронних системах є час очікування між клієнтом та сервером, де обидва повинні чекати завершення процесу. У системі, де в секунду надходять мільйони запитів, вони зазвичай приймають запит і обробляють його асинхронно із системами обміну повідомленнями, такими як *Kafka* або будь-яка черга повідомленнь.

1.3 Методи та моделі покращення показників *QoS* та нефункціональних характеристк веб сервісів

Веб-служби мають додаткові функції до своїх програм як частину реалізації аспекту у своїх послугах. Існує величезна кількість фреймворків у кожній категорії для впровадження у веб-службах у вигляді нефункціональних вимог. Є багато аспектів якості обслуговування, важливих для веб-служб. Вимоги щодо якості обслуговування, що стосуються веб-служб, стосуються якості як функціональних, так і нефункціональних характеристик веб-служб. Кожна категорія повинна мати набір кількісних параметрів або вимірювань. Атрибути якості обслуговування веб-служб мають найвищий пріоритет для різних постачальників послуг через непередбачувану та динамічну природу веб-служб. Вони включають час обслуговування, надійність, ціну виконання, доступність, масштабованість, доступність, продуктивність, ємність, цілісність, міцність / гнучкість, повнота, регулювання, репутація, вартість, точність, транзакційність, стабільність та *QoS*, пов'язані з безпекою. Проблеми веб-служб полягають у підвищенні вимог до якості обслуговування, щоб обґрунтувати їх обладнання для якості веб-послуг, і ця вимога повинна відповідати вибраним веб-службам. Знання вимог *QoS* в окремих перспективних веб-службах дає можливі результати згідно з визначеними та реалізованими прогнозами та найкращою практикою для покращення якості обслуговування веб-служб.

Рис. 1.3 QoS атрибути що відносяться до веб сервісів [5]

Продуктивність є однією з основних проблем веб-служб. Веб-сервіс - це гнучка концепції *QOS*. Веб-служби довели, що ця технологія є більш пристосованою та вищою за сучасні технології. Веб-служби досягають успіху завдяки своїй структурі, яка підтримує звичайну розподілену різнорідну архітектуру. Надалі обговорюється реальна продуктивність веб-сервісів та представлені різні набори даних для *QOS* веб-сервісу. Існує можливість для впроваджена технології сплячого режиму для покращення продуктивності веб-сервісу, впроваджуючи кращий підхід до відображення з відповідним сервером. Також розглядається питання веб-служб, пов’язаних із показником довіри до якості обслуговування запиту на послуги. Цей підхід передбачає, що кожен запит є справжнім, і жодного запиту який є аномалію не існує. У цій роботі пропонується відповідний розрахунок довіри для методу для нещодавно опублікованих веб-служб, але не для існуючих, і все ж визначити оптимальний спосіб налаштування цих веб-служб. Існує демонстрація функціональності виклику веб-сервісів із розподіленого та неоднорідного середовища, і ця робота зосереджена на таких питаннях, як ймовірність відмови служби, час відгуку тощо. Пропонується використати парадигму створення нової веб-служби на основі існуючих служб, яка називається гібридним підходом для підвищення продуктивності веб-служби. Сучасна архітектура орієнтована на технічні явища впровадження веб-служб і є стандартною методологією для розробки та

розгортання веб-служб на *.Net*. Існує пояснення основних проблем веб-служб розробки, пов’язану з публікацією, виявленням та відбором. Механізм реалізації цих подій залежить від типу веб-служби. Робота говорить про модель веб-сервісу на основі агента. Він підтримує інформацію щодо популярного, доступного на основі деяких відомих параметрів. Цю модель можна вдосконалити, проаналізувавши послуги в різні часові інтервали. Веб-сервіси довели, що ця технологія є більш гнучкою та вищою за сучасні. Веб-послуги мають величезний успіх завдяки своїй структурі, яка підтримує звичайне розподілене різнорідне середовище. Раніше було запропоновано автоматизований механізм прив'язки для зменшення накладних витрат на автентифікацію для запитів користувачів. Ця модель пакує всі подібні типи запитів і застосовує процес автентифікації в цілому, викликаючи відповідні веб-служби. Усі запити будуть оброблені, і в результаті не буде витрат на послуги. Аналіз на основі загальної моделі, який реалізований із використанням чотирьох різних комбінацій доступних технологій у веб-сервісах. Він відстежує показники продуктивності на основі часу, витраченого на відповідь, мережевого трафіку. Робота говорить про *QoS* у складних веб-сервісах. Це було досягнуто за рахунок оптимального планування послуги за допомогою моніторингу статусу послуги. Завжди існує розрив між розвитком та потребою послуги. Розробник більше зосереджується на питаннях, які очікуються під час розробки та впровадження, але не під час запиту користувача. Надалі описана метамодель, яка заповнює цю прогалину та робить її більш надійною.

Рис. 1.4. Компоненти моделі для передбачення *QoS* сервісу [6]

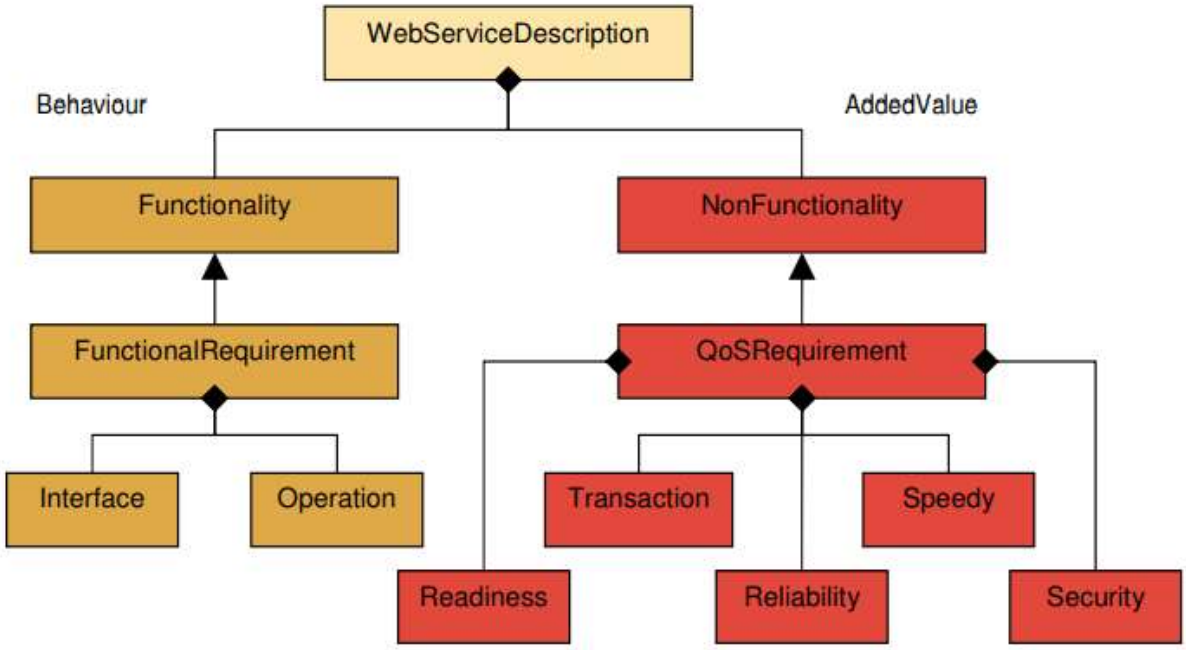

Рис. 1.5. Компоненти мета моделі для інтеграції *QoS* метрик до веб сервісу [7]

Веб-служби можуть зіткнутися з вузькими місцями щодо продуктивності через обмеження базових протоколів обміну повідомленнями та транспорту. Покладання на загальновизнані протоколи, такі як *HTTP* та *SOAP*, однак роблять їх постійним тягарем, на який потрібно лягти. Таким чином, важливо розуміти роботу цих обмежень.

HTTP - це протокол доставки даних через мережу Інтернет. Це механізм передачі даних, який має тенденцію створювати дві основні проблеми: немає гарантії доставки пакетів до місця призначення та немає гарантії порядку надходження пакетів. Якщо пропускна здатність відсутня, пакети просто відкидаються. Пропускна здатність, очевидно, є вузьким місцем, оскільки користувачі та обсяги даних, що працюють по мережі, збільшуються. Традиційно багато програм передбачають нульову затримку та нескінченну пропускну здатність. Також традиційно програми використовують синхронні повідомлення. Синхронний обмін повідомленнями блокує потоки, коли запускається програма на власних серверах; компоненти взаємодіють із затримками, що вимірюються в мікросекундах. Однак за допомогою веб-служб вони спілкуються через Інтернет,

а це означає, що затримки вимірюються десятками, сотнями або навіть тисячами мілісекунд. Хоча можуть бути використані нещодавно розроблені протоколи, такі як надійний *HTTP* (*HTTPR*), блокуваний розширюваний протокол обміну (*BEEP*) та пряма інкапсуляція повідомлень в Інтернеті (*DIME),* широке впровадження цих нових протоколів для транспорту веб-служб, таких як *HTTPR* та *BEEP*, займе певний час.[8] Отже, розробники додатків, які використовують веб-сервіси, повинні розуміти такі проблеми роботи веб-сервісу, як затримка та доступність, розробляючи свої системи. Способи покращення продуктивності веб-сервісу наведені нижче.

Програми, які покладаються на віддалені веб-служби, можуть використовувати чергу повідомлень для підвищення надійності, але ціною часу відгуку. Програми та веб-служби на підприємстві можуть використовувати черги повідомлень, такі як Служба обміну повідомленнями *Java* (*JMS*) або виклики *IBM MQSeries* for *Web Service*. Обмін повідомленнями на підприємствах забезпечує надійний, гнучкий сервіс для асинхронного обміну критичними даними на всьому підприємстві. Черги повідомлень мають дві основні переваги:

- Це асинхронно: постачальник послуг обміну повідомленнями може доставляти повідомлення запитувачу в міру їх надходження, і запитувач не повинен запитувати повідомлення для їх отримання.
- Це надійно: служба обміну повідомленнями може забезпечити доставку повідомлення один раз і лише один раз.

Постачальники послуг можуть активно надавати високий рівень якості обслуговування запитувачам послуг, використовуючи різні звичні підходи, такі як кешування та балансування навантаження запитів на послуги. Кешування та балансування навантаження можуть здійснюватися як на рівні веб-сервера, так і на рівні сервера веб-додатків. Балансування навантаження визначає пріоритети для різних типів трафіку та гарантує, що кожен запит обробляється відповідно до ділової цінності, яку він представляє. Постачальник веб-послуг може

виконати моделювання ємності, щоб створити модель зверху вниз трафіку запитів, поточного використання пропускної спроможності та отриманого *QoS*.

Постачальник послуг також може класифікувати трафік веб-послуг за обсягом трафіку, трафіком для різних категорій послуг додатків та трафіком з різних джерел.

Це допоможе зрозуміти пропускну здатність, яка буде необхідна для забезпечення якісної якості обслуговування для обсягу попиту на послуги та для подальшого планування, як-от пропускна здатність та тип серверів веб-додатків та / або веб-серверів із балансуванням навантаження (наприклад, кількість серверів, необхідних для налаштування ферми кластерних серверів). Постачальники послуг можуть забезпечити диференційоване обслуговування, використовуючи модель пропускної здатності для визначення пропускної спроможності, необхідної для різних споживачів та типів послуг, а також забезпечуючи належний рівень якості обслуговування для різних додатків та споживачів. Наприклад, мультимедійна веб-служба може вимагати хорошої пропускної здатності, але банківська веб-служба може вимагати безпеки та транзакцій *QoS*.

Транзакційне *QoS* стосується рівня надійності та послідовності, на якому виконуються транзакції. Транзакційне *QoS* має вирішальне значення для підтримки цілісності веб-служби. Операції дуже важливі для бізнес-процесів, щоб гарантувати, що комплекс пов’язаних видів діяльності розглядається та завершується як одна одиниця роботи. Якщо під час виконання транзакції виникає виняток, транзакція повинна бути відновлена до стабільного стану. Ця властивість називається "атомністю" транзакції.[9] Окрім властивості атомності, транзакції в більш суворому розумінні повинні задовольняти властивостям послідовності, ізоляції та довговічності. Всі ці чотири властивості разом називаються *ACID* властивостями.

Висновки

Сучасні технології стрімко розвиваються, вирішуючи проблеми швидко зростаючої кількості користувачів мережі Інтернет. Проте це також породжує нові виклики в сфері розподілених та хмарних обчислень.

Основними перевагами розподілених систем є:

- Надійність

- Масштабованість

- Гнучкість

- Висока продуктивність

Основними недоліками розподілених систем є:

- Складне усунення несправностей

- Менша підтримка програмного забезпечення

- Високі витрати на мережеву інфраструктуру

- Проблеми безпеки

На відміну від централізованих систем, розподілені програмні системи додають новий рівень складності до і без того складної проблеми проектування програмного забезпечення. Незважаючи на це, з описаних причин поширюється все більше і більше сучасних розподілених систем та сценаріїв використання хмарних технологій.

РОЗДІЛ 2

АНАЛІЗ СПОСОБІВ ОПТИМІЗАЦІЇ АСИНХРОННОГО ПРОЦЕСІНГУ

2.1. Аналіз переваг та недоліків монолітної архітектури

Розглянемо типовий додаток для електронної комерції, який приймає замовлення від клієнтів, перевіряє інвентар та наявний кредит та доставляє їх. Додаток складається з декількох компонентів, включаючи *StoreFrontUI*, який реалізує користувальницький інтерфейс, а також деякі серверні сервіси для перевірки кредиту, ведення інвентаризації та замовлення на доставку. Додаток розгортається як єдиний монолітний додаток. Наприклад, веб-програма Java складається з одного файлу *WAR*, який працює на веб-контейнері, такому як *Tomcat*. Додаток *Rails* складається з єдиної ієрархії каталогів, розгорнутої, використовуючи, наприклад, Phusion Passenger на *Apache* / *Nginx* або *JRuby* на Tomcat. Можна запускати кілька екземплярів програми за балансиром навантаження, щоб масштабувати та покращувати доступність.

Це рішення має ряд переваг:

- Просте у розробці - мета сучасних засобів розробки та *IDE* полягає в підтримці розробки монолітних додатків.
- Просте для розгортання - вам просто потрібно розгорнути файл *WAR* (або ієрархію каталогів) у відповідному середовищі виконання.
- Просте у масштабі - можна масштабувати сервіс, запускаючи кілька копій застосунку за балансувальником навантаження.

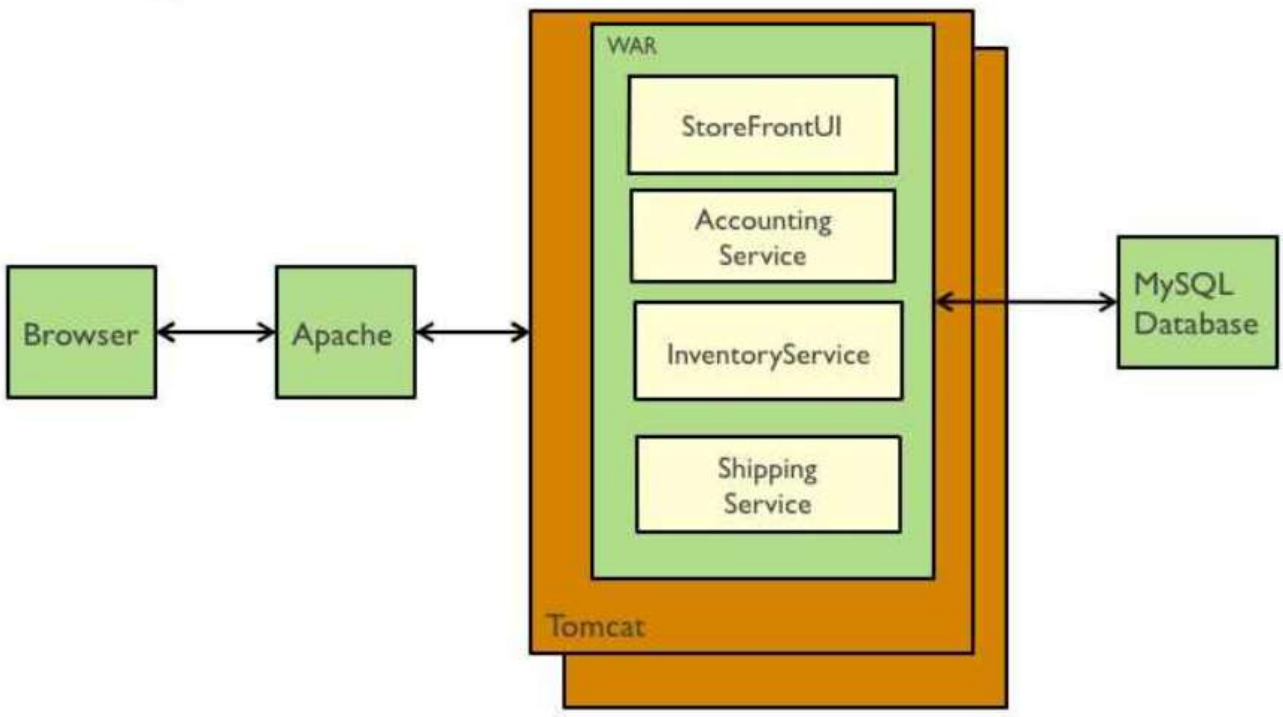

Рис. 2.1. Архітектура типового монолітного веб застосунку [8]

Однак, як тільки додаток стає великим, а команда збільшується в розмірах, цей підхід має ряд недоліків, які стають дедалі суттєвішими. Розглянемо недоліки та потенційні проблеми.

Велика монолітна база коду залякує розробників, особливо тих, хто новачок у команді. Додаток може бути важко зрозуміти та змінити. Як результат, розвиток зазвичай сповільнюється. Крім того, оскільки немає жорстких меж модулів, модульність з часом руйнується. Більше того, оскільки може бути важко зрозуміти, як правильно здійснити зміну, якість коду з часом знижується. Це спіраль вниз. Перевантажена *IDE* - чим більша база коду, тим повільніше *IDE* і менш продуктивні розробники. Перевантажений веб-контейнер - чим більше програма, тим більше часу потрібно для запуску. Це має величезний вплив на продуктивність розробника через втрату часу на очікування запуску контейнера. Це також впливає на розгортання. Постійне розгортання ускладнене - великий монолітний додаток також є перешкодою для частого розгортання. Для того, щоб оновити один компонент, вам слід передислокувати весь додаток. Це призведе до переривання фонових завдань (наприклад, завдань планувальника у програмі Java), незалежно

від того, чи вплинуть на них зміни, і, можливо, спричинить проблеми. Також є ймовірність того, що компоненти, які не були оновлені, не зможуть правильно запуститися. В результаті зростає ризик, пов’язаний із передислокацією, що відбиває часті оновлення. Це особливо проблема для розробників користувальницького інтерфейсу, оскільки їм, як правило, потрібно швидко повторити і часто передислокуватися. Масштабування програми може бути складним - монолітна архітектура полягає в тому, що вона може масштабуватися лише в одному вимірі. З одного боку, він може масштабуватися зі збільшенням обсягу транзакцій, запускаючи більше копій програми. Деякі хмари можуть навіть динамічно регулювати кількість екземплярів залежно від навантаження.

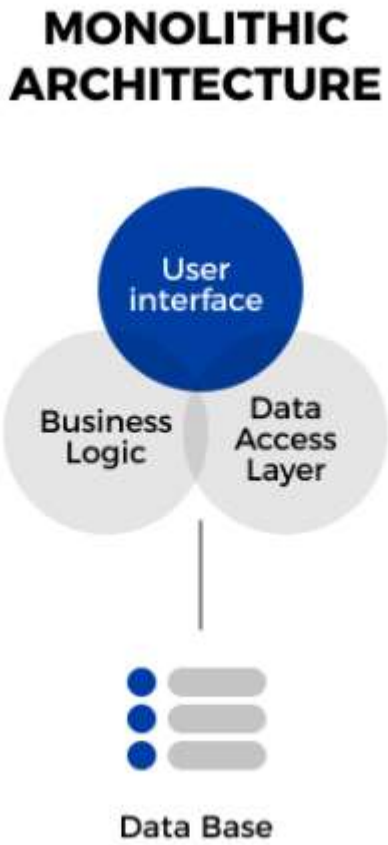

Рис. 2.2. Компоненти типового монолітного веб сервісу [9]

Але з іншого боку, ця архітектура не може масштабуватися зі збільшенням обсягу даних. Кожна копія екземпляра програми отримає доступ до всіх даних, що робить кешування менш ефективним і збільшує споживання пам'яті та трафік

вводу-виводу. Крім того, різні компоненти програми мають різні вимоги до ресурсів - один може вимагати великої кількості процесора, а інший - пам'яті. За допомогою монолітної архітектури ми не можемо масштабувати кожен компонент самостійно Перешкода для розвитку масштабування - Монолітна програма також є перешкодою для розвитку масштабування. Як тільки додаток досягне певного розміру, корисно розділити інженерну організацію на групи, які зосереджуються на конкретних функціональних областях. Наприклад, ми можемо захотіти мати команду інтерфейсу користувача, групу бухгалтерів, команду інвентаризації тощо. Проблема монолітного додатка полягає в тому, що це заважає командам працювати самостійно. Команди повинні координувати свої зусилля з розвитку та передислокації. Команді набагато складніше внести зміни та оновити виробництво. Потрібна довгострокова прихильність до стеку технологій - монолітна архітектура змушує одружитися зі стеком технологій (а в деяких випадках і з певною версією цієї технології), яка була вибрана на початку розробки. З монолітним застосуванням може бути важко поступово застосовувати новішу технологію. Наприклад, уявімо, що була вибрана *JVM*. Є кілька варіантів вибору мови, оскільки також Java, можна використовувати інші мови *JVM*, які добре взаємодіють з *Java*, такі як *Groovy* та *Scala*. Але компонентам, написаним не мовами *JVM,* буде не місце в такій монолітній архітектурі.

2.2 Аналіз переваг та недоліків мікросервісної архітектури

Існує архітектура, яка структурує додаток як набір вільно пов’язаних служб, що комунікують між собою. Кожен мікросервіс:

- Легкий для підтримки та тестування - забезпечує швидке та легке розгортання та еволюцію.

- Тісно пов'язаний з іншими сервісами - дозволяє команді працювати самостійно більшість часу на своїх послугах, не зазнаючи змін в інших послугах та не впливаючи на інші сервіси.

- Незалежне розгортання - дозволяє команді розгортати свою службу без необхідності координації з іншими командами.

- Здатність розроблятись невеликою командою - це важливо для високої продуктивності, уникаючи високого рівня комунікації великих команд Послуги взаємодіють, використовуючи або синхронні протоколи, такі як *HTTP* / *REST*, або асинхронні протоколи, такі як *AMQP*. Послуги можна розробляти та застосовувати незалежно одна від одної. Кожна служба має власну базу даних, щоб бути відокремленою від інших служб. Узгодженість даних між службами підтримується за допомогою паттерну Сага.

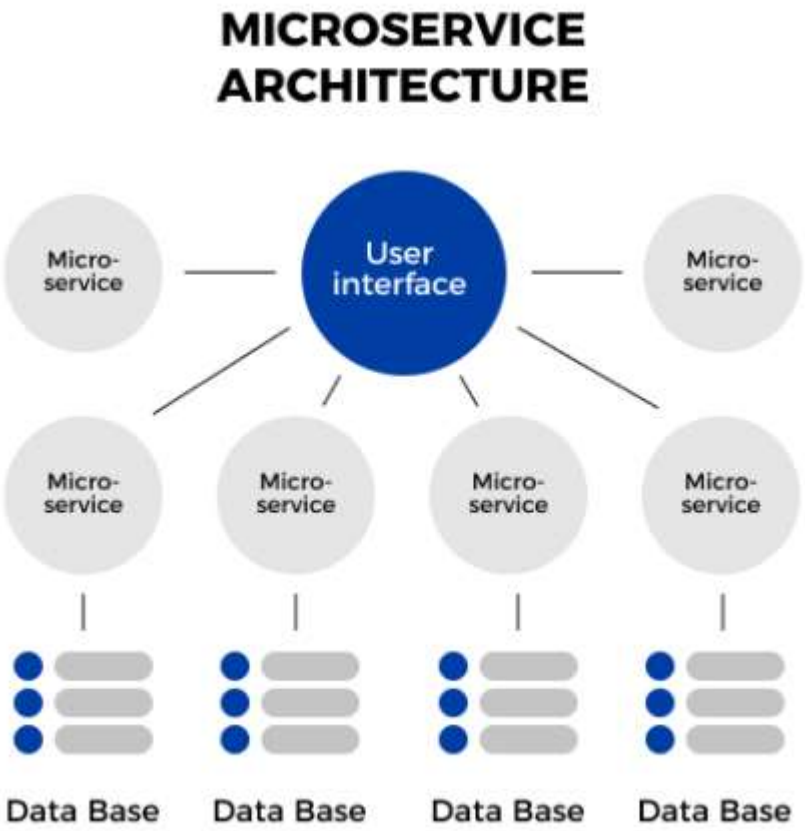

Рис. 2.3. Типова архітектура веб сервісу на основі мікросервісів [10]

Розглянемо типовий додаток для електронної комерції, який приймає замовлення від клієнтів, перевіряє інвентар та наявний кредит та доставляє їх. Додаток складається з декількох компонентів, включаючи *StoreFrontUI*, який реалізує користувальницький інтерфейс, а також деякі серверні сервіси для

перевірки кредиту, ведення інвентаризації та замовлення на доставку. Додаток складається з набору послуг.

Це рішення має ряд переваг:

- Забезпечує постійну доставку та розгортання великих складних підсистем.
- Покращена підтримуваність – кожен сервіс порівняно невеликий, тому його легше зрозуміти та змінити.
- Краща тестованість - сервіси менші та швидші для тестування
- Краща можливість розгортання - послуги можна розгорнути незалежно Це дозволяє організувати зусилля з розвитку навколо декількох автономних команд. Кожна (так звана дві піци) команда є власником однієї або декількох послуг. Кожна команда може розробляти, тестувати, застосовувати та масштабувати свої послуги незалежно від усіх інших команд.
- Кожен мікросервіс порівняно невеликий: Розробнику легше зрозуміти, а *IDE* швидше робить розробників більш продуктивними Додаток запускається швидше, що робить розробників більш продуктивними та пришвидшує розгортання.
- Покращена ізоляція несправностей. Наприклад, якщо в одній службі спостерігається витік пам'яті, це вплине лише на цю службу. Інші служби продовжуватимуть обробляти запити. Для порівняння, одна неправильна складова монолітної архітектури може зруйнувати всю систему. Виключає будь-яке довгострокове прихильність до технологічного стеку. При розробці нової послуги можна вибрати новий стек технологій. Подібним чином, вносячи серйозні зміни до існуючої служби, її можливо переписати, використовуючи новий стек технологій.

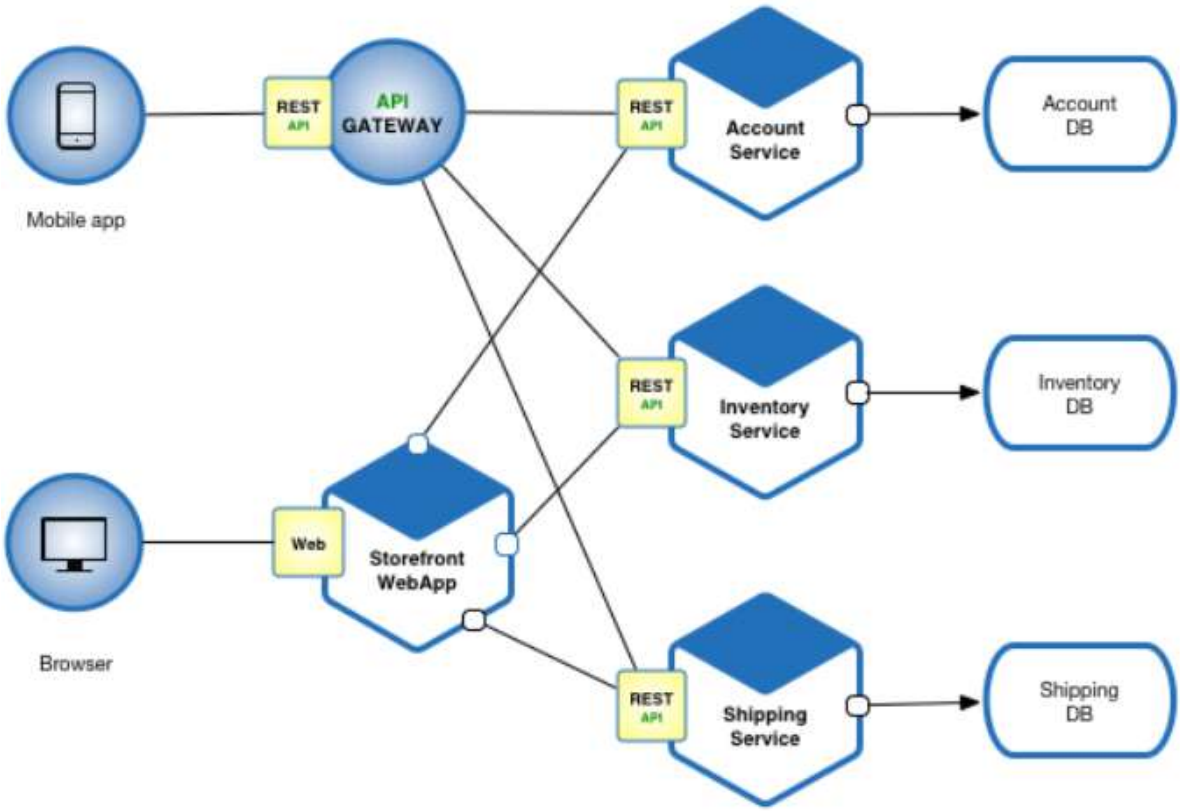

Рис. 2.3. Типові компоненти веб сервісу на основі мікросервісів [10]

Однією із проблем використання цього підходу є рішення, коли має сенс його використовувати. При розробці першої версії програми часто не виникає проблем, які вирішує цей підхід. Більше того, використання складної розподіленої архітектури уповільнить розвиток. Це може бути головною проблемою для стартапів, чия найбільша проблема часто полягає в тому, як швидко розвивати бізнес-модель та супровідний додаток. Однак пізніше, коли проблема полягає в тому, як масштабувати і вам потрібно використовувати функціональну декомпозицію, заплутані залежності можуть ускладнити розклад вашого монолітного додатку на набір служб.

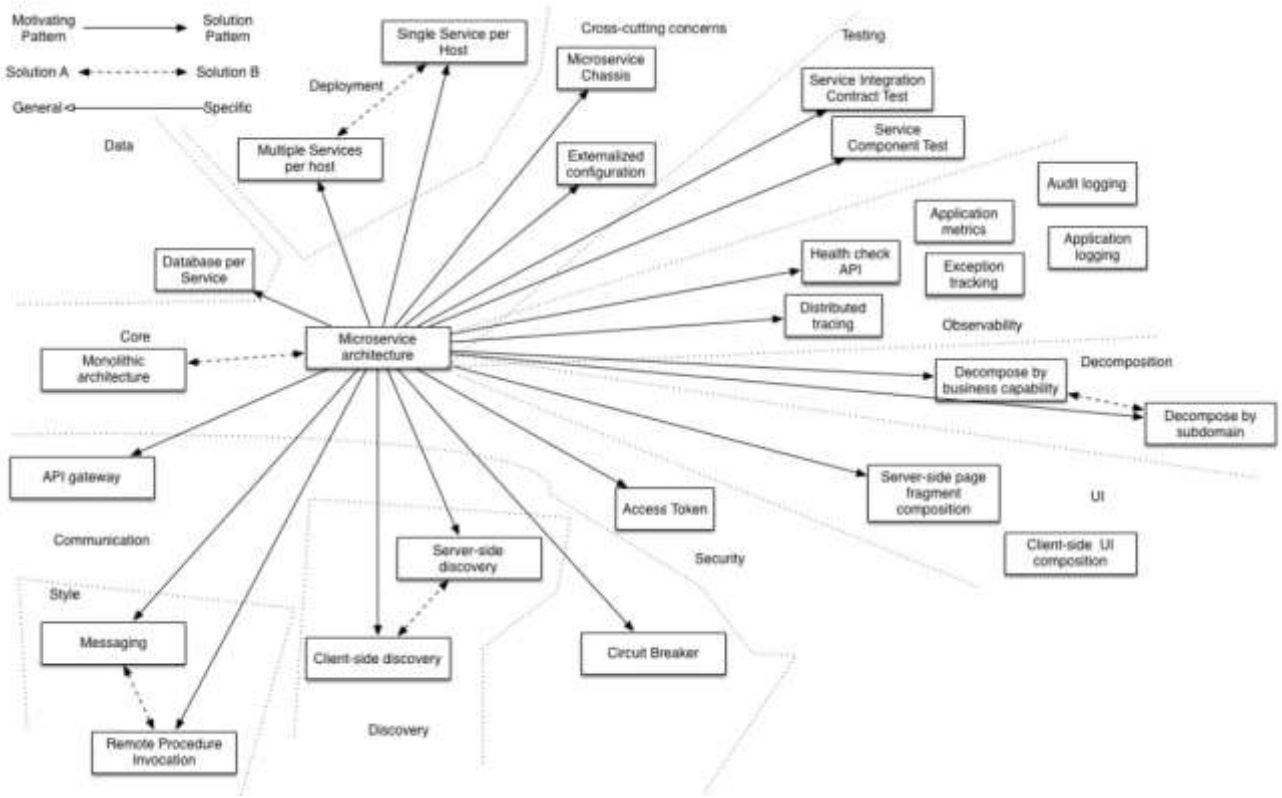

Рис. 2.4. Аспекти мікросервісної архітектури [11]

Для забезпечення вільного зв’язку кожна служба має свою базу даних. Підтримка узгодженості даних між службами є проблемою, оскільки транзакції з двома етапами розподілу / розподілу не є варіантом для багатьох додатків. Натомість програма повинна використовувати шаблон Сага. Сервіс публікує подію, коли її дані змінюються. Інші служби споживають цю подію та оновлюють свої дані. Існує кілька способів надійного оновлення даних та публікації подій, включаючи джерело подій та відстеження журналу транзакцій.

2.3 Розробка оптимізацій для мікросервісеної архітектури

Проаналізувавши існуючі архітектури з їх перевагами та недоліками очевидно що саме мікросервісна архітектура є найбльш підходящою для сучасних інфокомунікаційних систем та патернів навантажень як протягом дня, так і протягом року.

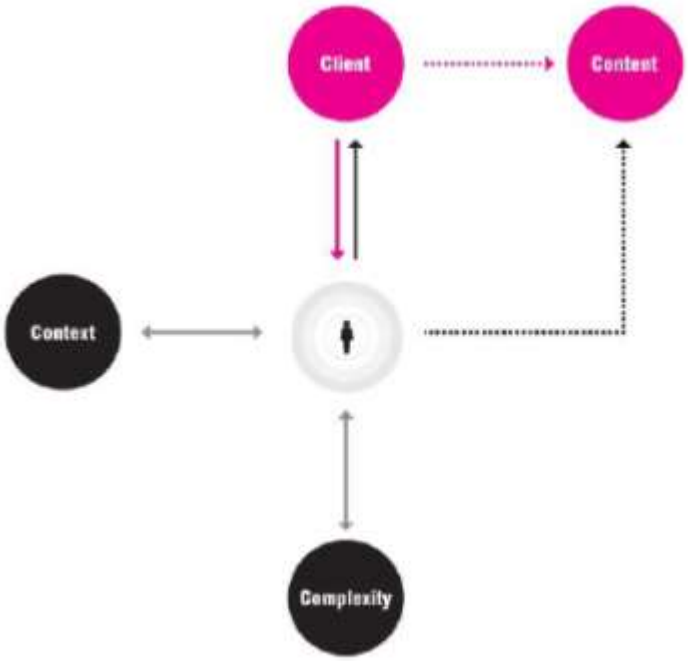

Рис. 2.4. Схема аналізу вимог до архітектури [11]

У сучасній хмарній архітектурі додатки розділені на менші, незалежні будівельні блоки, які простіше розробити, розгорнути та обслуговувати. Черги повідомлень забезпечують зв'язок та координацію для цих розподілених додатків. Черги повідомлень можуть суттєво спростити розробку відокремлених додатків, одночасно підвищуючи продуктивність, надійність та масштабованість. Черги повідомлень дозволяють різним частинам системи спілкуватися та обробляти операції асинхронно. Черга повідомлень забезпечує легкий буфер, який тимчасово зберігає повідомлення та кінцеві точки, що дозволяють програмним компонентам підключатися до черги для надсилання та отримання повідомлень. Повідомлення, як правило, невеликі, і це можуть бути речі, такі як запити, відповіді, повідомлення про помилки або просто інформація. Щоб надіслати повідомлення, компонент, який називається продюсером, додає повідомлення в чергу. Повідомлення зберігається в черзі, доки інший компонент, який називається споживачем, не отримує повідомлення і не робить з ним щось.

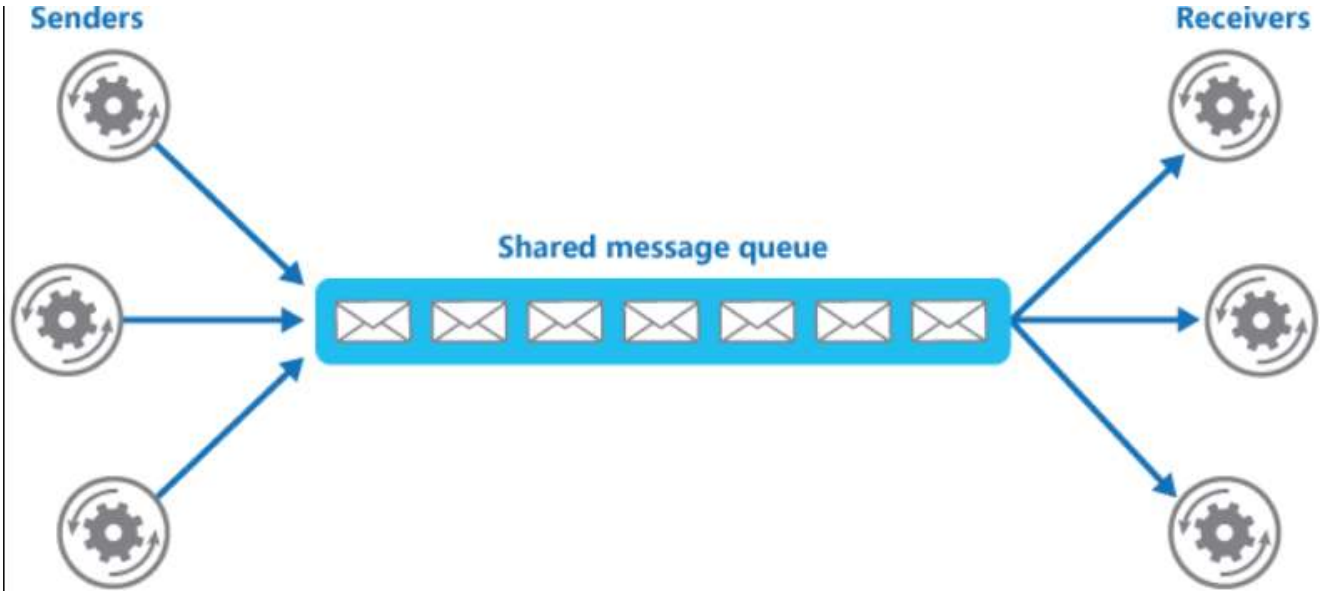

Рис. 2.5. Архітектура яка використовує розподілену чергу [12]

Черги повідомлень відіграють важливу роль в проектуванні архітектури веб додатку, оскільки вони допомагають добитися меншого зв'язування між сервісами. Два або більше додатків не зв'язані, якщо вони можуть спілкуватися між собою, не будучи на одній фізичній машині. Крім того, одна програма не знає про реалізацію іншої програми. Іншими словами, між ними немає залежностей.

Повідомлення - це дані, що передаються між відправником та додатком одержувача; це по суті байтовий масив з деякими заголовками зверху. Прикладом повідомлення може бути подія. Одна програма вказує іншій програмі розпочати обробку певного завдання через чергу. Основна архітектура черги повідомлень проста: існують клієнтські програми, що називаються паблішерами, які створюють повідомлення та доставляють їх до черги повідомлень. Інша програма, яка називається споживачем, підключається до черги і отримує повідомлення, які підлягають обробці. Повідомлення, розміщені в черзі, зберігаються, поки споживач їх не отримає.

Переваги розподіленої черги:

- Будь-які зміни, внесені в один застосунок, не впливають на інший застосунок, якщо контракт про обмін даними не порушений
- Можна розбити одну монолітну програму на менші, що зменшує загальну складність.

- Стає простіше підтримувати та налагоджувати програми. Ми можемо мати крос-платформні додатки. Менші програми можна самостійно розробити на будь-яких мовах програмування та масштабувати.

Більшість продуктів обміну повідомленнями підтримують підхід до обміну повідомленнями "точка-точка" або "публікація / підписка".

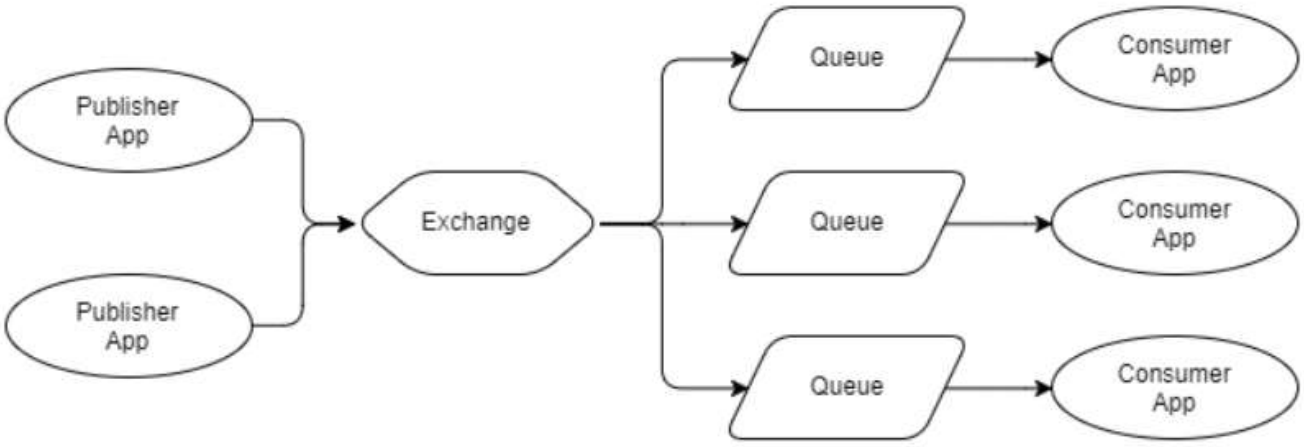

Рис. 2.6. Архітектура розподіленої черги [13]

Черга паблішерів та консьюмерів обміну повідомленнями: - за цією чергою обміну повідомленнями створіть тему, і кілька споживачів можуть підписатися на цю тему, і підписник отримує повідомлення, розміщене за темою.

Обмін повідомленнями традиційно має дві моделі: чергування та публікація-підписка. У черзі група споживачів може читати з сервера, і кожен запис надходить до одного з них; при публікації-підписці запис транслюється всім споживачам. Кожна з цих двох моделей має сильні та слабкі сторони. Сила черги полягає в тому, що вона дозволяє розподілити обробку даних на декілька екземплярів споживача, що дозволяє масштабувати обробку. На жаль, черги не є кількома підписниками - коли один процес зчитує дані, він зникає. *Publish-subscribe* дозволяє передавати дані в різні процеси, але не має можливості масштабування, оскільки кожне повідомлення надходить кожному підписнику.

Іншим способом оптимізації є також пакетна обробка, коли система агрегує дані та обробляє їх відразу групами.

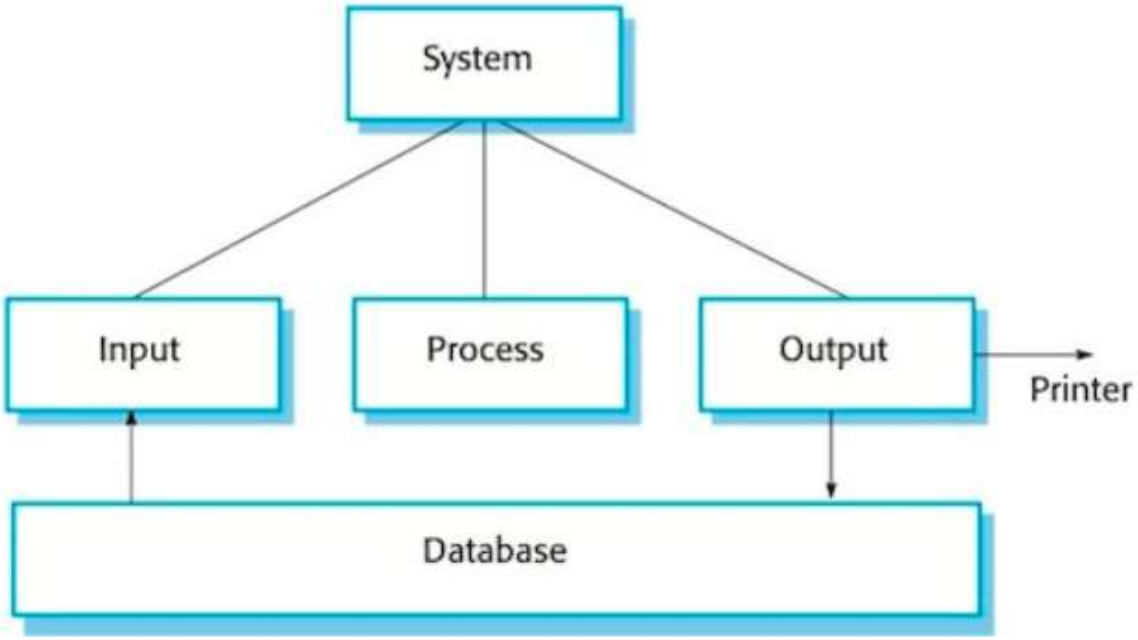

Рис. 2.7. Компоненти архітектури з пакетною обробкою [15]

Архітектура систем пакетної обробки має три основні компоненти, як показано на рисунку 2.7. Вхідний компонент збирає вхідні дані з одного або декількох джерел; компонент обробки робить обчислення, використовуючи ці входи; і вихідний компонент генерує результати, які слід записати назад у базу даних та роздрукувати. Наприклад, телефонна система виставлення рахунків приймає записи клієнта та показання лічильника телефону (входи) з комутатора, обчислює витрати для кожного клієнта (процес), а потім друкує рахунки (виходи) для кожного клієнта. Вхідні, обробні та вихідні компоненти можуть самі бути розкладені на структуру вхід-процес-вихід. Наприклад: Вхідний компонент може зчитувати деякі дані (вхідні дані) із файлу або бази даних, перевіряти достовірність цих даних та виправляти деякі помилки (процес), а потім ставити в чергу дійсні дані для обробки (виведення). Компонент обробки може взяти транзакцію з черги (введення), виконати деякі обчислення даних і створити новий запис даних, що реєструє результати обчислення (процесу), а потім поставити цей новий запис у чергу для друку (виведення). Іноді обробка здійснюється в системі бази даних, а іноді це окрема програма. Вихідний компонент може зчитувати записи з черги (введення), форматувати їх відповідно до вихідної форми (процесу), потім надсилати їх на принтер або записувати нові записи назад у базу даних (вихід).

Характер систем обробки даних, де записи або транзакції обробляються послідовно, без необхідності підтримувати стан транзакцій, означає, що ці системи природно орієнтовані на функції, а не на об'єкти. Функції - це компоненти, які не підтримують інформацію про внутрішній стан від одного виклику до іншого. Діаграми потоків даних - це хороший спосіб описати архітектуру систем обробки бізнес-даних.

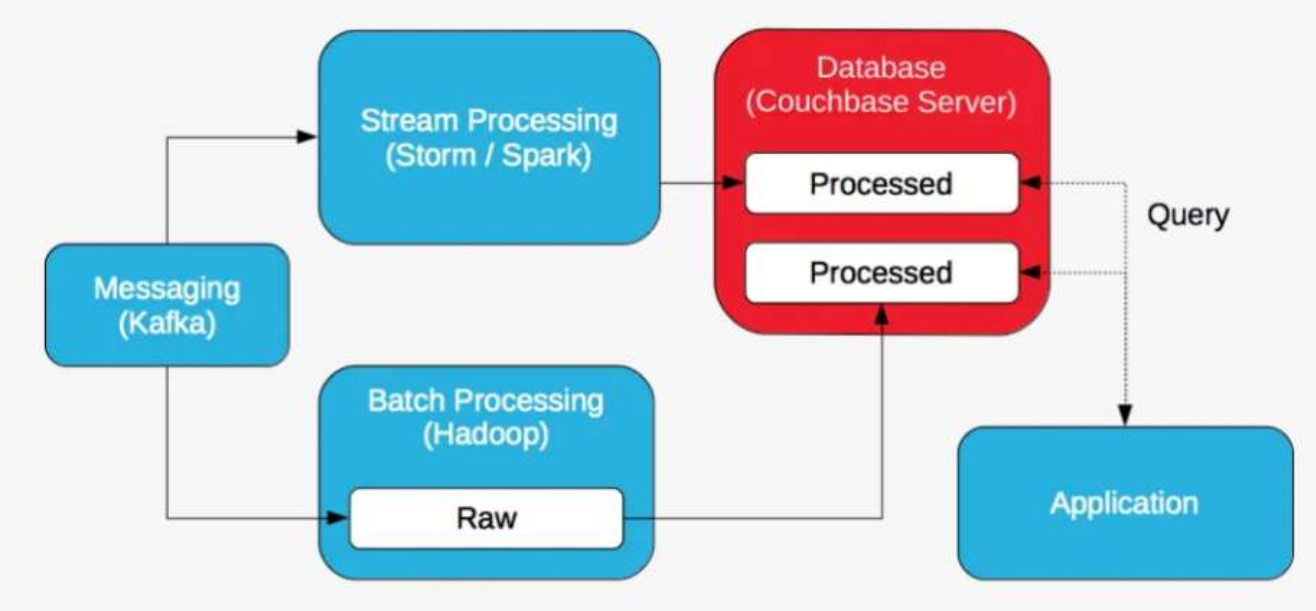

Рис. 2.8. Застосунок з розподіленою чергою та пакетною обробкою [14]

При комбінації розглянутих прийомів та компонентів результатом буде оптимізована мікросервісна архітектура. Оптимізованими характеристиками є RPS, кількість запитів на секунду які може обслужити веб сервіс та показники затримки відповіді.

Висновки

Розглянувши основні характеристики монолітної та мікросервісної архітектур можна зробити висновок що саме мікросервісна архітектура має більше переваг для високонавантажених веб сервісів та систем.

Для оптимізації була вибрана розподілена черга як компонент архітектури, оскільки розподілена черга дозволяє добитися виконання ключових вимог для покращення показників затримки відповіді та спроможності обробляти сотні

запитів в секунду. Іншим необхідним компонентом є пакетна обробка (*batch processing*). Застосувавши обидва цих компонента, можна значно покращити характеристики веб сервісу який їх застосовує.

РОЗДІЛ 3
РОЗРОБКА ОПТИМІЗОВАНОЇ МІКРОСЕРВІСНОЇ АРХІТЕКТУРИ

3.1. Структура системи

Система створюється за розподіленою технологією яка хоститься в хмарі *Azure* на кластері віртуальних машин в сервісі *Azure Web Apps*.

На рисунку 3.1 представлена архітектура, котра показує перевірені практики для веб-додатків, які використовують службу програм *Azure* та базу даних *SQL Azure*.

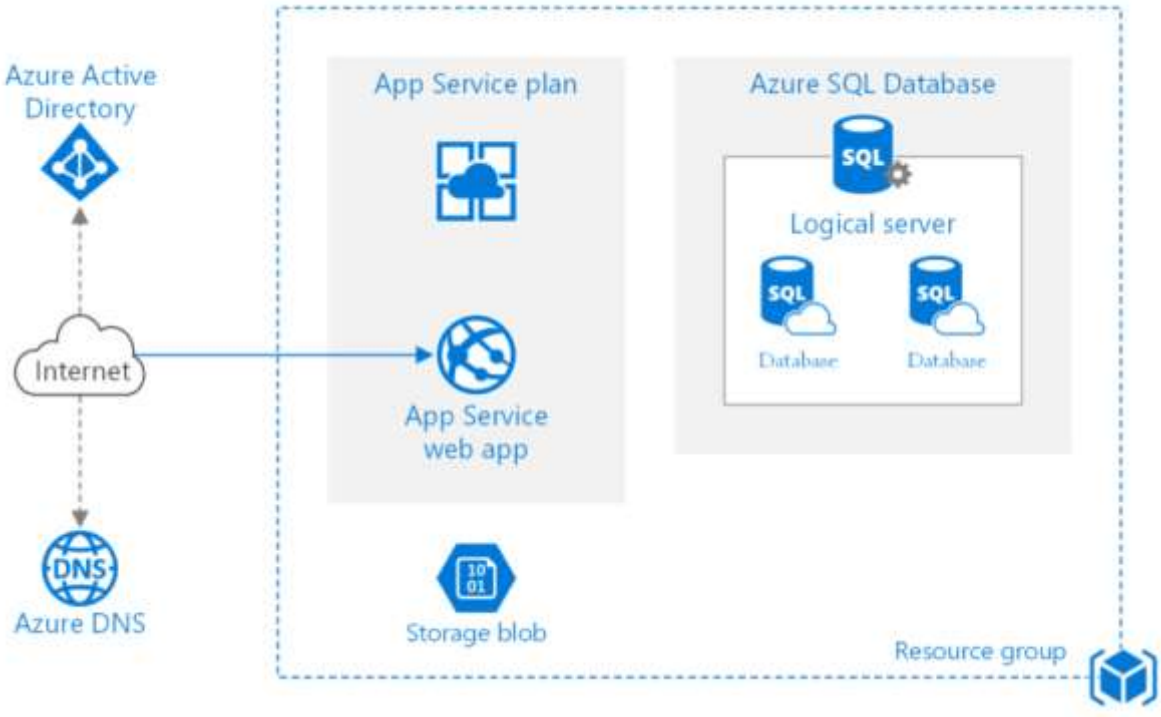

Рис.3.1 Представлення архітектури *Azure* [8]

Архітектура має такі компоненти:

Ресурсна група. Група ресурсів - це логічний контейнер для ресурсів *Azure*.

Служба додатків. Служба програм *Azure* - це повністю керована платформа для створення та розгортання хмарних програм.

План служби додатків. План служби додатків надає керовані віртуальні машини (*BM*), на яких розміщується ваш додаток. Усі програми, пов'язані з планом, працюють на одних і тих же екземплярах *BM*.[15]

Слоти для розгортання. Слот розгортання дозволяє вам інсценувати розгортання, а потім поміняти його на виробниче розгортання. Таким чином, можна уникнути розгортання безпосередньо на виробництві.

IP-адреса. Додаток *App Service* має загальнодоступну *IP*-адресу та доменне ім’я. Доменне ім’я є субдоменом azurewebsites.net, таким як *contoso.azurewebsites.net*.[16]

Azure DNS. *Azure DNS* - це послуга хостингу для доменів *DNS*, що забезпечує роздільну здатність імен за допомогою інфраструктури *Microsoft Azure*. Розмістивши свої домени в *Azure*, можна керувати своїми записами *DNS*, використовуючи ті самі облікові дані, *API*, інструменти та виставлення рахунків, що й інші служби *Azure*. Для використання власного доменного імені (наприклад, contoso.com) створіть записи *DNS*, які відображають ім’я власного домену на *IP*-адресу. Щоб отримати додаткові відомості, див. Розділ Налаштування власного доменного імені в службі програм *Azure*.

База даних *SQL* Azure. База даних *SQL* - це реляційна база даних як послуга в хмарі. База даних *SQL* ділиться своєю базою коду з механізмом баз даних Microsoft *SQL* Server. Залежно від вимог до вашої програми, також можна використовувати базу даних Azure для *MySQL* або базу даних *Azure* для *PostgreSQL*. Це повністю керовані служби баз даних, засновані на відкритих кодах *MySQL Server* та *Postgres*, відповідно.

Логічний сервер. У базі даних *SQL Azure* логічний сервер розміщує бази даних. На логічному сервері можна створити кілька баз даних.

Сховище *Azure*. Створено обліковий запис сховища *Azure* за допомогою контейнера *BLOB*-об'єктів для зберігання журналів діагностики.[17]

Azure Active Directory (*Azure AD*). Для автентифікації використовуйте *Azure AD* або іншого постачальника ідентифікаційних даних.

Логічна група серверів спрощує адміністративні завдання. Кожна база даних у групі розгортається з певним сервісним рівнем. У межах кожної групи бази даних не можуть спільно використовувати ресурси. Для сервера немає обчислювальних витрат, але для кожної бази даних потрібно вказати рівень. Отже, завдяки виділеним ресурсам продуктивність може бути кращою, але вартість може бути вищою.

Існує два способи масштабування програми *App Service*:

Збільшити масштаб, що означає зміну розміру екземпляра. Розмір екземпляра визначає пам’ять, кількість ядер та пам’ять на кожному екземплярі *ВМ*. Можна масштабувати вручну, змінюючи розмір екземпляра або рівень плану.

Масштабуйте, що означає додавання екземплярів для обробки підвищеного навантаження. Кожен рівень цін має максимальну кількість екземплярів.

Можна масштабувати вручну, змінюючи кількість екземплярів, або використовувати автомасштабування, щоб *Azure* автоматично додавав або видаляв екземпляри на основі розкладу та / або показників продуктивності. Кожна операція з масштабуванням відбувається швидко - як правило, за лічені секунди.

Щоб увімкнути автомасштабування, створив профіль автомасштабування, який визначає мінімальну та максимальну кількість екземплярів. Профілі можна запланувати.

Масштабування веб-програми:

Наскільки це можливо, уникнув масштабування вгору та вниз, оскільки це може спричинити перезапуск програми. Натомість обрав рівень та розмір, які відповідають вашим вимогам до продуктивності під типовим навантаженням, а потім масштабуйте екземпляри, щоб обробляти зміни в обсязі трафіку.

Увімкнув автомасштабування. Якщо у додатку передбачене регулярне навантаження, створіть профілі, щоб запланувати підрахунок екземплярів заздалегідь. Якщо навантаження не передбачуване, використовуйте автоматичне масштабування на основі правил, щоб реагувати на зміни навантаження в міру їх виникнення. Можна поєднати обидва підходи.

Використання центрального процесора, як правило, є хорошим показником для правил автоматичного масштабування. Однак слід завантажити тест програми, визначити потенційні вузькі місця та на основі цих даних засновувати правила автомасштабування.

Правила автомасштабування включають період охолодження, тобто інтервал очікування після завершення дії масштабу перед початком нової дії масштабу. Період охолодження дозволяє системі стабілізуватися перед повторним масштабуванням. Встановив коротший період охолодження для додавання примірників та довший період охолодження для видалення примірників. Встановив 5 хвилин, щоб додати екземпляр, але 60 хвилин, щоб видалити екземпляр. Краще швидко додавати нові екземпляри під великим навантаженням, щоб обробляти додатковий трафік, а потім поступово зменшувати масштаб.

У випадку втрати даних, база даних *SQL* забезпечує відновлення та геовосстановлення за часом. Ці функції доступні на всіх рівнях і вмикаються автоматично

Використав відновлення за часом, щоб відновити помилку людини, повернувши базу даних у попередній момент часу.

Використав геовідновлення для відновлення після відключення служби, відновлюючи базу даних із резервної георезервованої резервної копії.

У цій архітектурі було використано шаблон *Azure Resource Manager* для надання ресурсів *Azure* та їх залежностей. Оскільки це єдиний веб-додаток, усі ресурси ізольовані в одному основному робочому навантаженні, що полегшує зв’язування конкретних ресурсів робочого навантаження з командою, щоб команда могла самостійно керувати всіма аспектами цих ресурсів. Ця ізоляція дозволяє команді *DevOps* здійснювати безперервну інтеграцію та безперервну доставку (*CI* / *CD*). Крім того, використав різні шаблони *Azure Resource Manager* та інтегрувати їх із службами *Azure DevOps* для надання різних середовищ за лічені хвилини, наприклад для реплікації виробництва, наприклад сценаріїв, або середовищ тестування навантажень лише за потреби, заощаджуючи витрати.

Розгортання.

Розгортання передбачає два етапи:

Надання ресурсів *Azure*. Для цього кроку використав шаблони *Azure Resource Manager*. Шаблони спрощують автоматизацію розгортання за допомогою *PowerShell* або *Azure CLI*.

Розгортання програми (код, двійкові файли та файли вмісту). У вас є кілька варіантів, включаючи розгортання з локального сховища *Git*, використання *Visual* Studio або безперервне розгортання з керованого хмарного джерела.

Додаток Служби додатків завжди має один слот розгортання з назвою production, який представляє дійсний робочий сайт. Було створено проміжний слот для розгортання оновлень. Переваги використання слота для постановки включають:

Розгортання в проміжний слот гарантує, що всі екземпляри будуть розгорнуті перед тим, як перейти на виробництво. Багато програм мають значний час розминки та холодного запуску.

Також було створено третій слот для проведення останнього відомого розгортання. Після того, поміняно місцями інсталяцію та виробництво, перенесено попереднє розгортання виробництва (яке зараз перебуває у стадії інсталяції) в останній відомий добрий слот. Таким чином, якщо виявлено проблему пізніше, можна швидко повернутися до останньої відомої хорошої версії.

Конфігурація.

Було збережено параметри конфігурації як налаштування програми. Визначено параметри програми у шаблонах диспетчера ресурсів або за допомогою PowerShell. Під час виконання налаштування програми доступні додатку як змінні середовища.

Паролі не перевірялись, ключі доступу чи рядки підключення в елементі керування джерелом. Натомість було передано їх як параметри сценарію розгортання, який зберігає ці значення як налаштування програми.

3.2. Модель розгортання веб сервісу на хмару

Служба програм на *Linux* надає високо масштабовану, самовиправлення послугу веб-хостингу за допомогою операційної системи *Linux*. Цей швидкий

старт використав для створення програми *.NET Core* на службі програм у *Linux*. Створено програму за допомогою *Azure CLI* з використанням *Git* для розгортання коду *.NET Core* у програмі.

Мікросервіси - це популярний архітектурний стиль для побудови пристосувань, які є стійкими, масштабованими, незалежно розгортаються і здатні швидко розвиватися. Але успішна архітектура мікросервісів вимагає іншого підходу до проектування та побудови додатків. Архітектура мікросервісу складається з колекції невеликих автономних служб. Кожна послуга є автономною і повинна реалізовувати єдиний бізнес-потенціал. Перехід на сучасні процеси *CI* / *CD* забезпечує багато переваг для побудови додатків, розгортання, тестування та моніторингу. Використовуючи *Azure DevOps* разом з іншими службами, такими як Служба додатків, організації можуть зосередитись на розробці своїх програм, а не на управлінні допоміжною інфраструктурою.

Налаштувавши конвеєр за допомогою *Azure DevOps Projects* і завершивши збірку, після відповідних змін коду, робочі елементи та результати тестування стає помітно, що результати тесту не відображаються, оскільки код не містить тестів для запуску. Конвеєр створює визначення випуску та тригер тривалого розгортання, розгортаючи наш додаток у середовищі *Dev*. Як частина безперервного процесу розгортання, можна побачити випуски, що охоплюють кілька середовищ. Випуск може охоплювати як інфраструктуру (використовуючи такі методи, як інфраструктура як код), а також може розгортати необхідні пакети програм разом із будь-якими завданнями після конфігурації. *Azure DevOps* виставляється за кожного користувача щомісяця. Залежно від необхідних одночасних конвеєрів, на додаток до будь-яких додаткових тестових користувачів або базових ліцензій, можуть стягуватися додаткові збори.

3.3. Система обміну повідомленнями

Архітектура месенджеру. *Messenger* розробляється щоб бути одночасно і захищеним на рівні *Tox, BitMessage*, і зручним на рівні *Telegram* і *WhatsApp*. Як

виглядає архітектура і які рішення були використані щоб досягти поставлених цілей.

Переваги сучасних месенджерів зібрано в даному сервісі:

Спілкування:

1. Відправка тексту, фото, файлів, голосових повідомлень;
2. Відправлення повідомлень, захищених наскрізним шифруванням (включаючи фото і файли);
3. Реалізація діалогів, розрахованих на багато користувачів чатів і каналів;
4. Отримання push-повідомлень про нові повідомлення;
5. Висока швидкість відправки повідомлень;
6. Можливість відображення онлайну користувачів і індикації набору тексту;
7. Ієрархія ролей користувачів в групових чатах і каналах;

Реєстрація та авторизація користувачів:

1. Можливість анонімної реєстрації без номера телефону / пошту;
2. Можливість реєстрації за номером телефону / поштою;
3. Можливість приховати персональні дані або обмежити видимість певних груп;
4. Можливість переглядати список сесій користувача;
5. Відсутність жорсткої прив'язки користувача до сервера і можливість перенесення даних користувача між серверами;

Робота з користувачами:

1. Пошук користувачів за доступними даними (ім'я, пошта, телефон);
2. Пошук користувачів за ідентифікатором;
3. Зберігання списку контактів і груп контактів;

Адміністрування:

1. Можливість розгортання власного сервера;
2. Автопоновлення серверного додатка;
3. Автоматичне створення резервних копій;

4. Налаштування під місцеві вимоги (законів і внутрішніх політик власників);
5. Можливість налаштування доступності шифрування для користувачів сервера;
6. Можливість перегляду незашифрованих діалогів адміністратором сервера;
7. Зберігання статичних файлів в хмарі.

Мережева архітектура. Взаємодія всіх учасників мережі ми будуємо на базі протоколу *WebSocket*. Файли передаються по протоколу *HTTP*. Опис протоколу є в нашому репозиторії на *GitHub*.

Цілі і завдання:

1. Створити децентралізувати месенджер.
2. Забезпечити рівність учасників децентралізрованной мережі.
3. Забезпечити відмовостійкість збоїв окремих серверів.
4. Оптимізувати навантаження на мобільні пристрої користувачів.
5. Забезпечити захист призначених для користувача даних.
6. Забезпечити можливість спілкування між користувачами різних серверів.
7. Мінімізувати залежність користувачів від інших елементів мережі
8. Забезпечити можливість реалізації функцій сучасних централізованих месенджерів.

Рішення. Було розглядено різні варіанти і прийшов до висновку, що мережева архітектура, здатна найкращим чином задовольнити всім вимогам - "федералізована децентралізація".

Федералізована децентралізація передбачає створення спільних точок обміну даними (хабів). Хаби є довіреними представниками користувачів, і користувачі дозволяють хабам обробляти частина інформації про себе в обмін на зниження навантаження на свій пристрій.

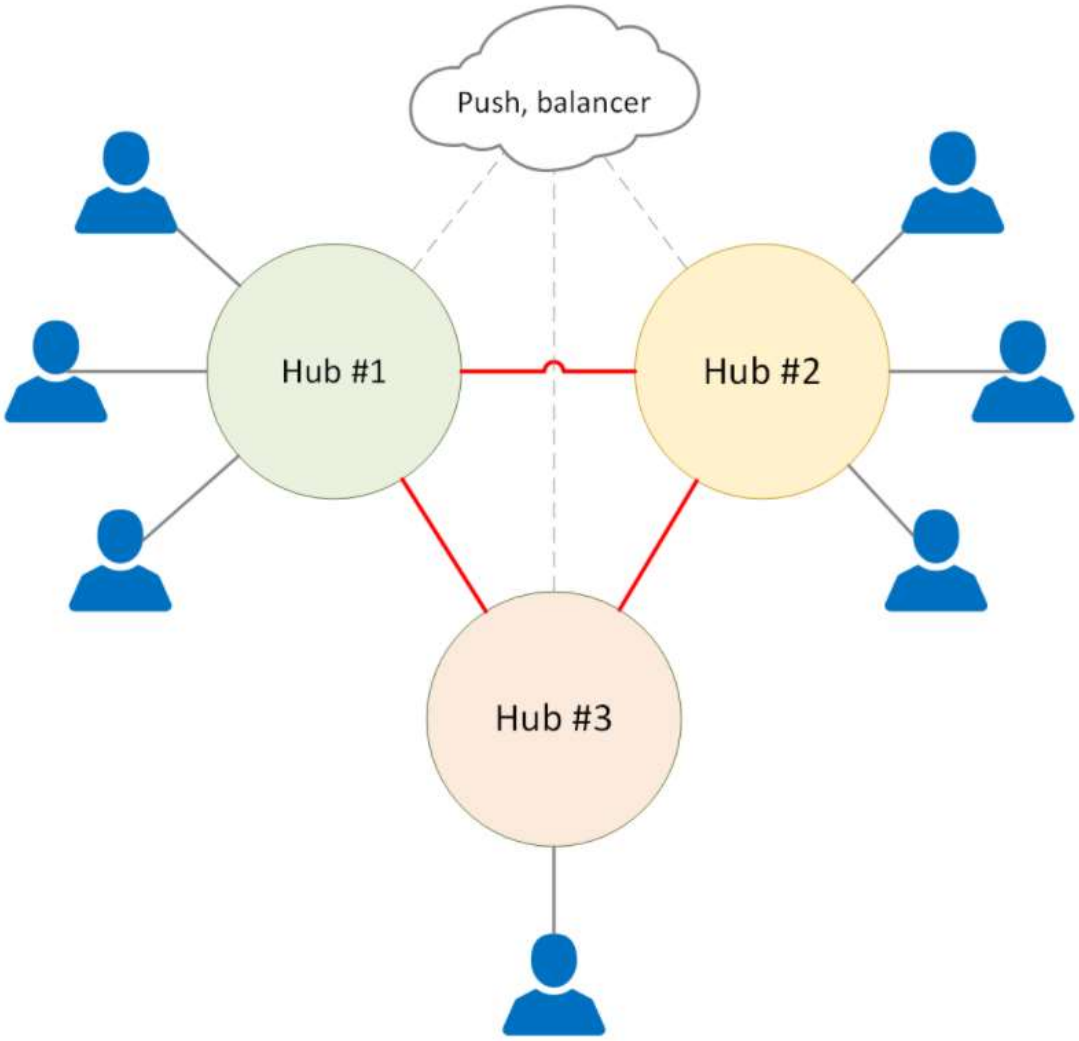

Рис. 3.2. Мережа хабів

Як видно з рисунку вище, користувач підключаються до свого хабу. Все хаби рівнозначні і утворюють тимчасову мережу.[18]

Аналогічну мережеву архітектуру утворюють поштові сервера (*E-mail*).

Хаби. Завдання хабів:

1. Авторизація, реєстрація та верифікація користувачів.
2. Зберігання даних про користувача (профіль, повідомлення, контакти, настройки).
3. Забезпечення передачі даних між користувачами.
4. Надання інформації про користувача відповідно до його налаштуваннями конфіденційності.

5. Маршрутизація запитів від інших хабів і користувачів.
6. Зберігання файлів (у себе або в *S3*-сумісному сховище).
7. Підтримка розподіленого сховища *Blockchain*.
8. Резервне копіювання даних.
9. Проксінг запитів між користувачів інших хабів при збої хаба, до якого вони належать.
10. Адміністрування своїх користувачів.

Хаб може бути запущений ким завгодно, подібно сервера електронної пошти. Команда розробників *Y messenger* не має доступу до хабам користувачів і не може читати / записувати / змінювати / видаляти дані в цих хабах.

3.4. Архітектура з використанням розподіленої черги

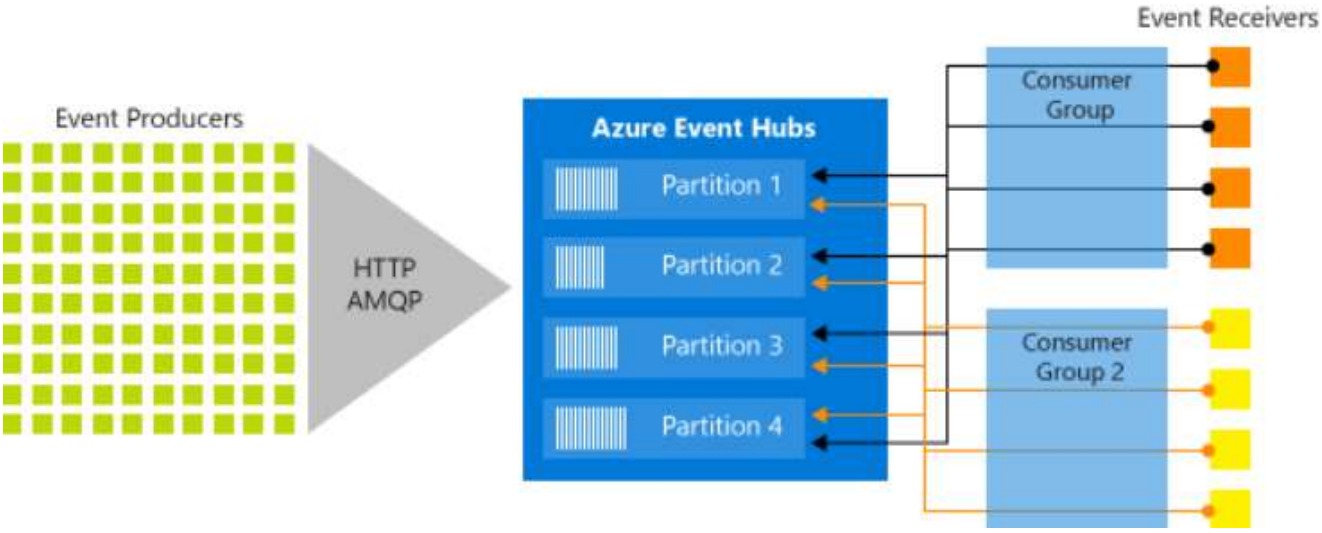

Рис. 3.3. Архітектура *Azure Event Hubs*

Azure Event Hub - це велика платформа для потокового передавання даних та служба передачі подій. Вона може приймати і обробляти мільйони подій в секунду. Дані, що надсилаються до центру подій, можуть бути перетворені та збережені за допомогою будь-якого постачальника аналітики в режимі реального часу або адаптерів пакетного зберігання / зберігання.

Дані цінні лише тоді, коли існує простий спосіб їх обробки та отримання своєчасної інформації з джерел даних. Концентратори подій надають платформу

обробки розподіленого потоку з низькою затримкою та безперебійною інтеграцією, а також служби передачі даних та аналітики всередині та за межами *Azure* для створення вашого повного конвеєру великих даних. Концентратори подій представляють "вхідні двері" для конвеєру подій, який часто називають організатор подій в архітектурних рішеннях. Організатор подій - це компонент або послуга, яка знаходиться між видавцями подій та споживачами подій, щоб відокремити виробництво потоку подій від споживання цих подій. *Event Hubs* забезпечує уніфіковану потокову платформу з буфером збереження часу, відокремлюючи виробників подій від споживачів подій.

Можна завантажувати, буферизувати, зберігати та обробляти свій потік у режимі реального часу, щоб отримати статистичну інформацію. Концентратори подій використовують розділену споживчу модель, що дозволяє кільком програмам одночасно обробляти потік і дозволяє контролювати швидкість обробки. Можна захоплювати свої дані майже в реальному часі в сховищі *Azure Blob* або сховищі даних *Azure Data Lake* для тривалого зберігання або обробки в пакетному режимі. Можна досягти цієї поведінки в тому самому потоці, який використовується для отримання аналітики в режимі реального часу. Налаштування збору даних про події відбувається швидко. Немає адміністративних витрат на його запуск, і він автоматично масштабується за допомогою одиниць конфігурації. Концентратори подій дозволяють зосередитись на обробці даних, а не на збір даних.

Завдяки широкій екосистемі, доступній різними мовами *.NET*, *Java*, *Python*, JavaScript, можливо легко розпочати обробку своїх потоків із центрів подій. Усі підтримувані мови клієнта забезпечують низькорівневу інтеграцію. Екосистема також забезпечує безперебійну інтеграцію із службами *Azure*, такими як *Azure Stream Analytics* та *Azure Functions*, і таким чином дозволяє створювати безсерверні архітектури.

3.5. Архітектура за принципами REST

REST (*Representational state transfer*) - це стиль архітектури програмного забезпечення для розподілених систем, таких як *World Wide Web,* який, як правило, використовується для побудови веб-служб. Термін *REST* був введений у 2000 році Роєм Філдінгом, одним з авторів *HTTP*-протоколу. Системи, що підтримують *REST*, називаються *RESTful*-системами.[19]

У загальному випадку *REST* є дуже простим інтерфейсом управління інформацією без використання якихось додаткових внутрішніх прошарків. Кожна одиниця інформації однозначно визначається глобальним ідентифікатором, таким як *URL*. Кожна *URL* в свою чергу має строго заданий формат.

Відсутність додаткових внутрішніх прошарків означає передачу даних в тому ж вигляді, що і самі дані. Тобто ми не загортаємо дані в *XML*, як це робить *SOAP* і *XML-RPC*, не використовуємо *AMF*, як це робить *Flash* і т.д. Просто віддаємо самі дані.

Кожен ресурс однозначно визначається *URL* - це значить, що *URL* по суті є первинним ключем для одиниці даних. Тобто наприклад третя книга з книжкової полиці матиме вигляд / book / 3, а 35 сторінка в цій книзі - / book / 3 / page / 35. Звідси і виходить строго заданий формат. Причому абсолютно не має значення, в якому форматі знаходяться дані за адресою / book / 3 / page / 35 - це може бути і *HTML*, і відсканована копія у вигляді jpeg-файлу, і документ *Microsoft Word*.

Як відбувається управління інформацією сервісу - це цілком і повністю ґрунтується на протоколі передачі даних. Найбільш поширений протокол звичайно ж *HTTP*. Так ось, для *HTTP* дію над даними задається за допомогою методів: *GET* (отримати), *PUT* (додати, замінити), *POST* (додати, змінити, видалити), *DELETE* (видалити). Таким чином, дії *CRUD* (*Create-Read-Updtae-Delete*) можуть виконуватися як з усіма 4-ма методами, так і тільки за допомогою *GET* і *POST*.

Як видно, в архітектура *REST* дуже проста в плані використання. По виду прийшов запиту відразу можна визначити, що він робить, не розбираючись в форматах (на відміну від *SOAP, XML-RPC*). Дані передаються без застосування додаткових шарів, тому *REST* вважається менш ресурсномістких, оскільки не треба

парсити запит щоб зрозуміти що він повинен зробити і не треба переводити дані з одного формату в інший.

Найголовніше гідність сервісів в тому, що з ними працювати може яка завгодно система, будь то сайт, flash, програма та ін. Так як методи парсинга *XML* і виконання запитів *HTTP* присутні майже всюди.

Архітектура *REST* дозволяє серйозно спростити цю задачу. Звичайно в реальності, того що описано мало, адже не можна кому завгодно давати можливість змінювати інформацію, тобто потрібна ще авторизація та аутентифікація. Але це досить просто дозволяється за допомогою різного типу сесій або просто *HTTP Authentication*.

Кожна одиниця інформації однозначно визначається *URL* - це значить, що *URL* по суті є первинним ключем для одиниці даних. Тобто наприклад третя книга з книжкової полиці матиме вигляд / book / 3, а 35 сторінка в цій книзі - / book / 3 / page / 35.

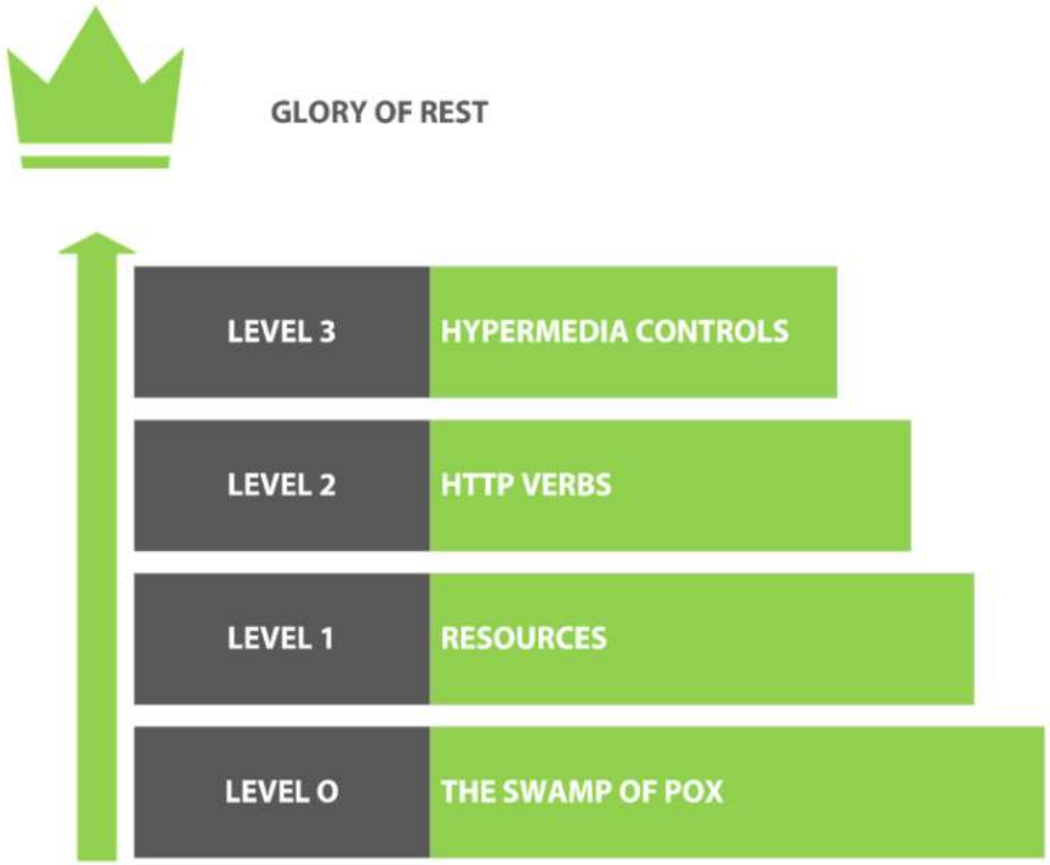

Рис. 3.4. *HATEOAS*: реалізація функцій в *RESTful API* [20]

Викликаючи той чи інший *REST API* додаток отримує не тільки дані про ресурс - сукупність характеристик в форматі 'ключ = значення' або навіть більш складну структуру, наприклад, у вигляді *JSON* дерева, а й набір гіперпосилань. Можна вважати, що ваш додаток - це кінцевий автомат. Його станом поточним станом є *URL* програмного інтерфейсу, який він тільки що викликав, а набором допустимих переходів в інші стани - безліч гіперпосилань, отриманих у відповіді. Кожна отримана гіперпосилання - це дієслово, саме відповідну операцію для перекладу вашого додатка в будь-яке інше стан.

3.6. Діаграми класів

Даний модуль в собі містить код загального призначення, який використовується в інших модулях проекту та не несе якоїсь бізнес логіки.

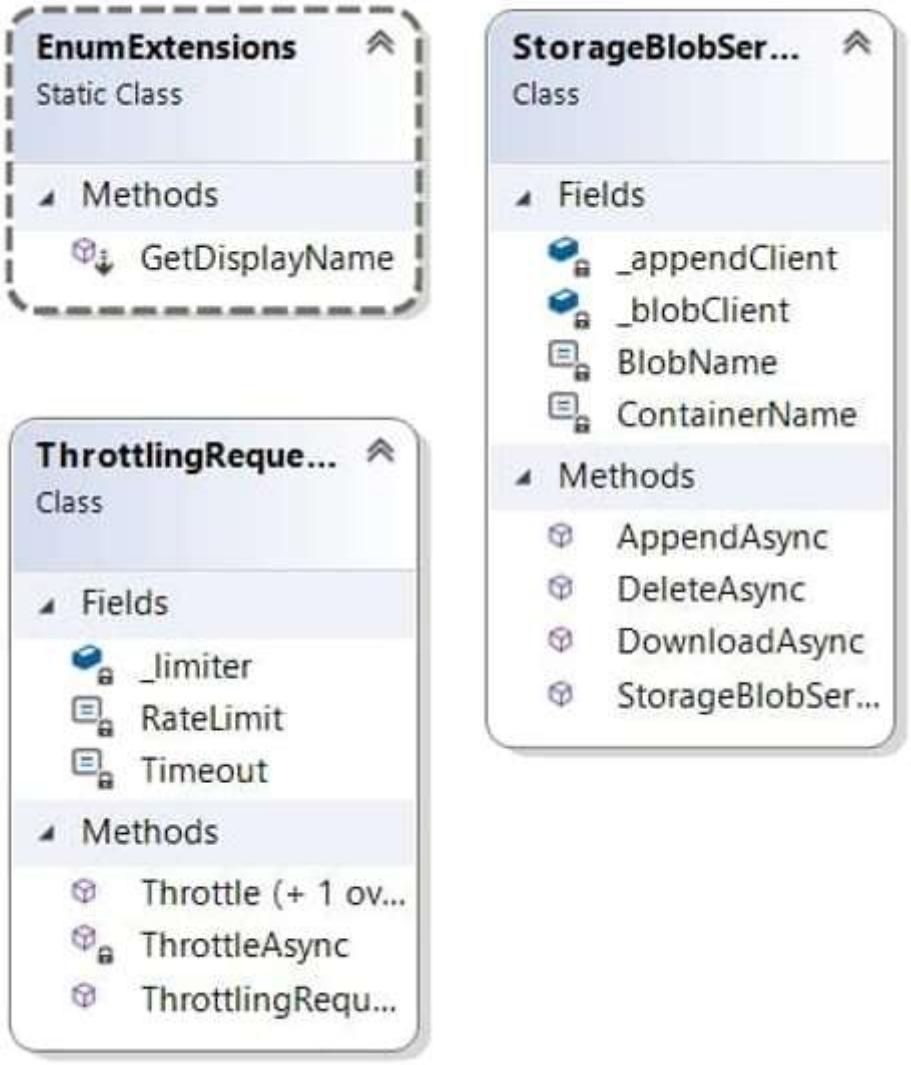

Рис. 3.5. Модуль *Common*

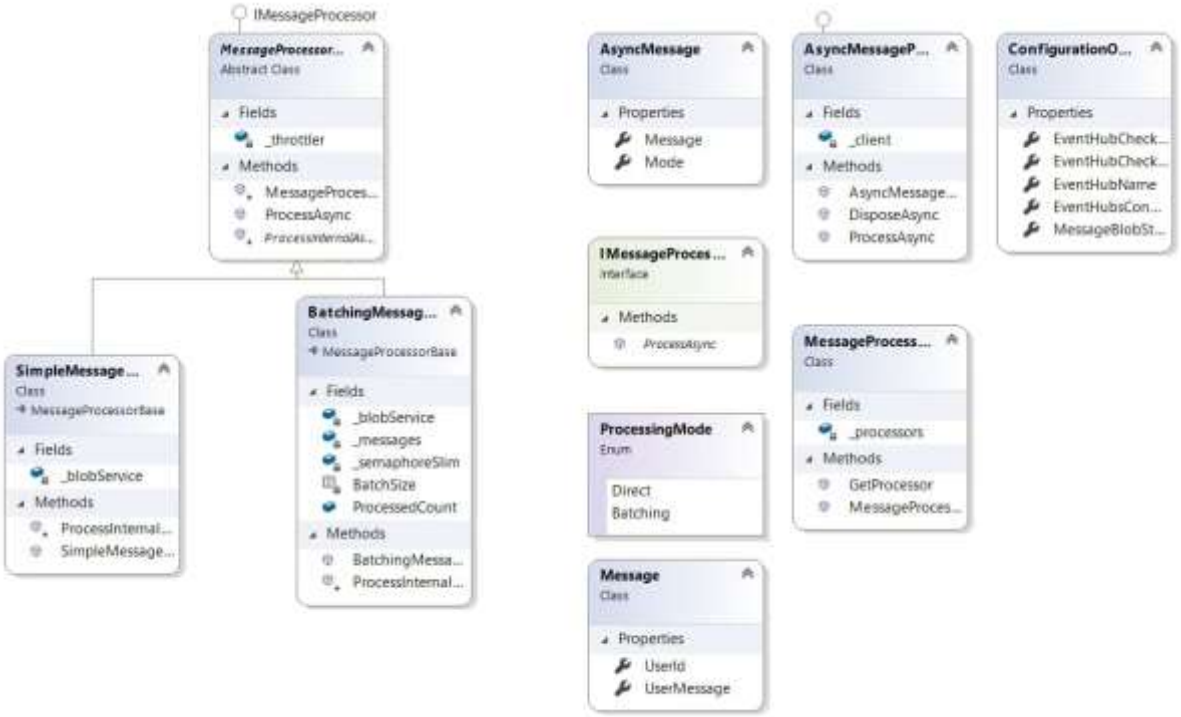

Рис. 3.6. Модуль *MessageProcessor*

Модуль *MessageProcessor* містить в собі спільну бізнес логіку для обробки повідомлень. Даний код було винесено в спільний модуль для легшого перевикористання між застосунками public *API* та *async processor*.

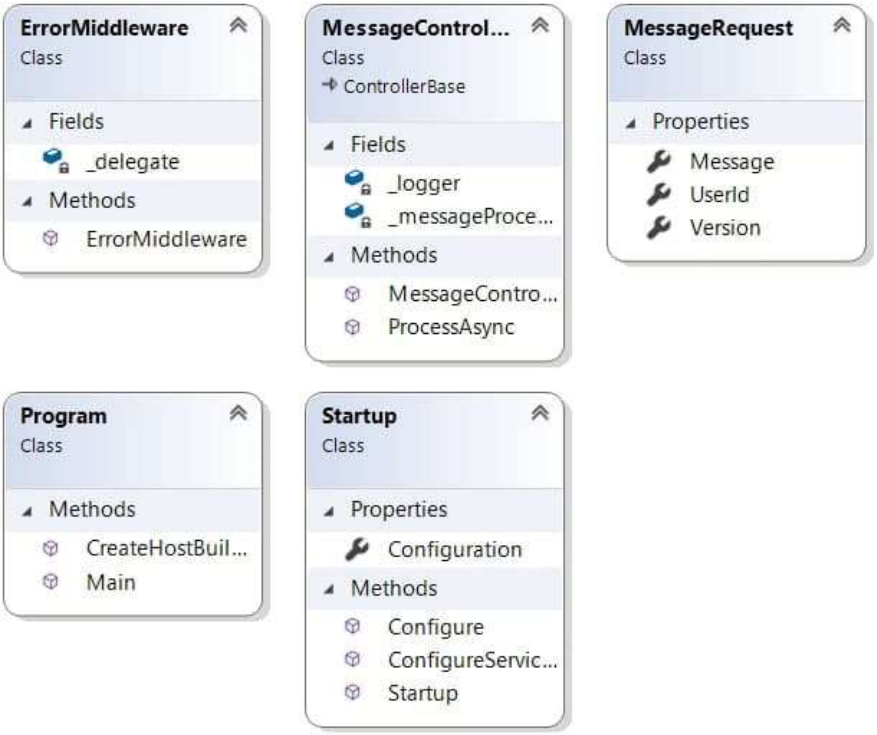

Рис. 3.7. Модуль *Publisher*

Модуль *Publisher* це веб застосунок який представляє публічне *API* яким можуть користуватися клієнти для користування месенджером.

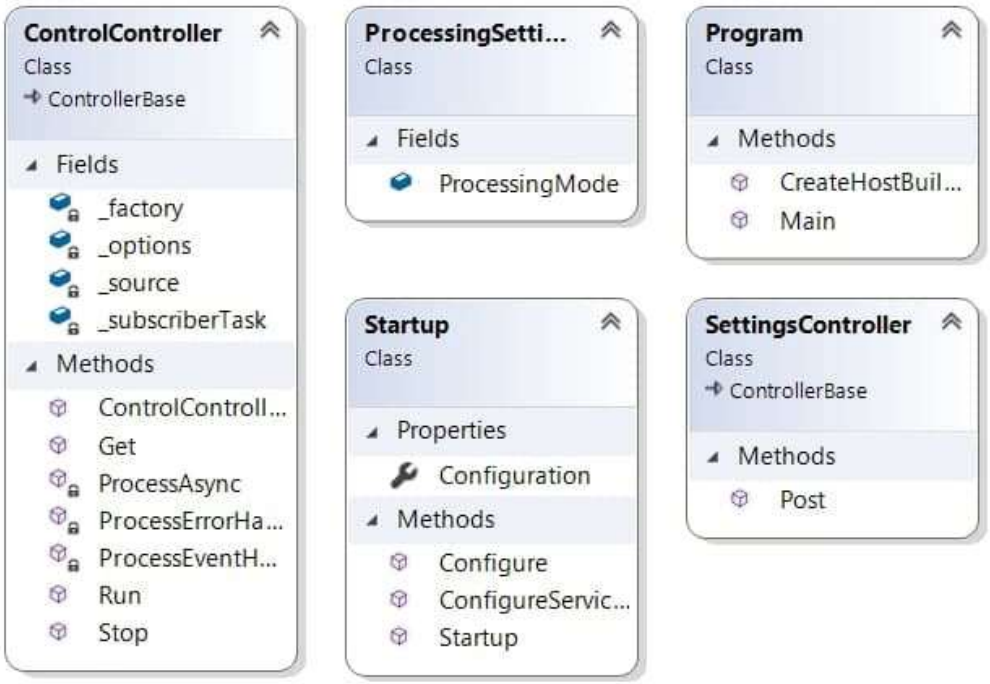

Рис. 3.8. Модуль *Subscriber*

Модуль *Subscriber* реалізує собою підсистему асинхронного процесінгу, шляхом зчитування повідомленнь з розподіленої черги і обробки в пакетному режимі.

Висновки

Для оптимізації мікросервісної архітектури з використанням розподіленої черги та пакетної обробки була вибрана хмара *Microsoft Azure*.

В ній було розгорнуто застосунки побудовані за допомогою мікросервісної архітектури та веб фреймворку *ASP.NET Core*. В ролі розподіленої черги був вибраний сервіс *Azure Event Hubs*, який реалізує протокол *AMQP*, та є реалізацією гарантій *ACID* та at-least-once доставки. Для зберігання результатів було вибрано *Azure Blob Storage*.

Процес обробки данних асинхронний, та полягає в потоку даних від публічного *API* до акумуляції в розподіленій черзі для подальшої обробки модулем Consumer та зберігання результатів обчислення над пакетами даних в персистентному сховищі.

РОЗДІЛ 4
АНАЛІЗ РЕЗУЛЬТАТІВ ОПТИМІЗАЦІЇ АСИНХРОННОЇ ОБРОБКИ ДАНИХ В РОЗПОДІЛЕНИХ СИСТЕМАХ

4.1 Підготовка тестового середовища

1. Веб-додатки в *Azure* дозволяють легко опублікувати веб-сайт і керувати ним, не піклуючись про базові серверах, сховище або мережевих ресурсах. Замість цього можна зосередитися на функціоналі сайту, поклавшись на надійну платформу *Azure*, яка забезпечить безпечний доступ до нього. Вимоги, які маються на увазі або перетворені з високорівневого вимоги. Наприклад, вимога для більшого радіусу дії або високій швидкості може привести до вимоги низької ваги.
2. Реєстр контейнерів *Azure* - це керована служба реєстру *Docker*, заснована на відкритому коді *Docker Registry* 2.0. Реєстр контейнерів є приватним, розміщується в *Azure* і дозволяє створювати, зберігати та керувати зображеннями для всіх типів розгортання контейнерів. Дізнайтеся, як створювати та зберігати образи контейнерів за допомогою реєстру контейнерів *Azure*.
3. Служба додатків на платформі *Linux* - це високомасштабована служба розміщення з самостійною установкою виправлень на основі операційної системи *Linux*. У цьому короткому посібнику показано, як створити додаток *.NET Core* в службі додатків на платформі Linux. Створіть програму за

Список усіх *CI*/*CD* пайплайнів які використовуються для збірки й розвертання системи в хмару. На даний момент це 2 пайплайна для публічного *API* та підсистеми асинхронної обробки.

Рис. 4.1. Налаштування *CI/CD*

2. Типове налаштування пайплайнів

Для збірки використовуються кроки *Nuget Restore* для отримання пакетів *NuGet*, власне компіляція та завантаження в хмару.

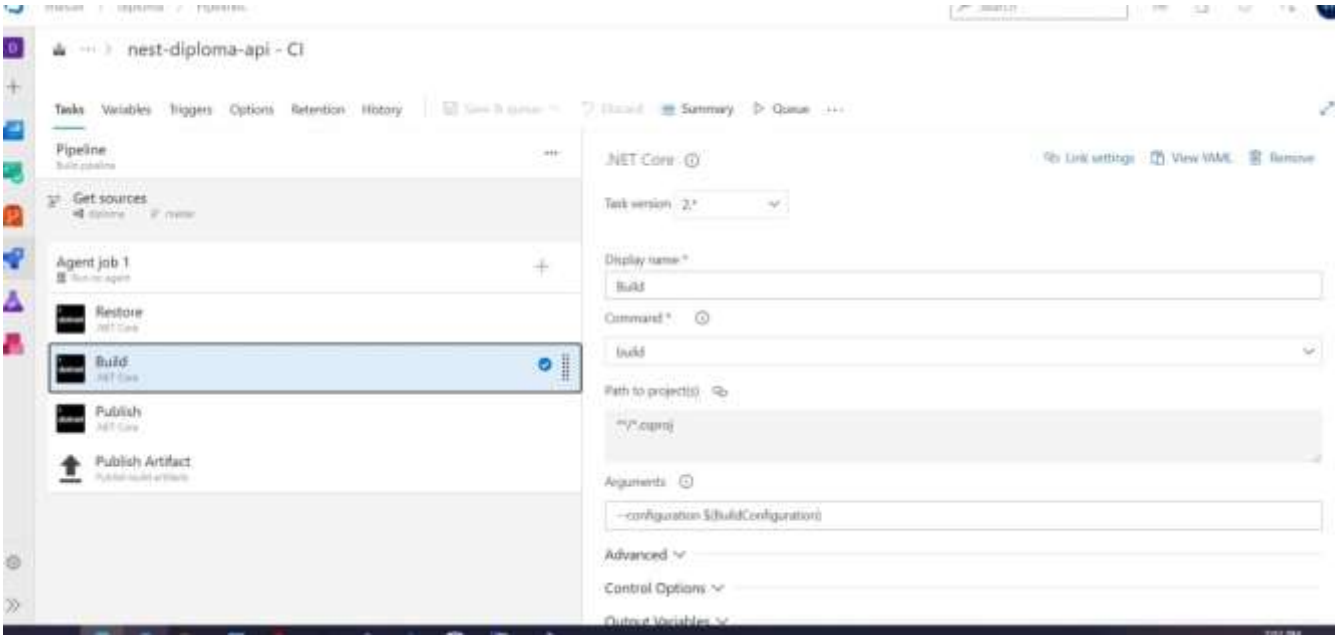

Рис. 4.2. Налаштування пайплайну збірки та розгортання сервісу

3. Логування

Завдяки сервісу *Azure DevOps* ми маємо змогу бачити логи всіх кроків збірки та деплоя кожної підсистеми та мати змогу швидко та ефективно локалізувати та вирішити проблему.

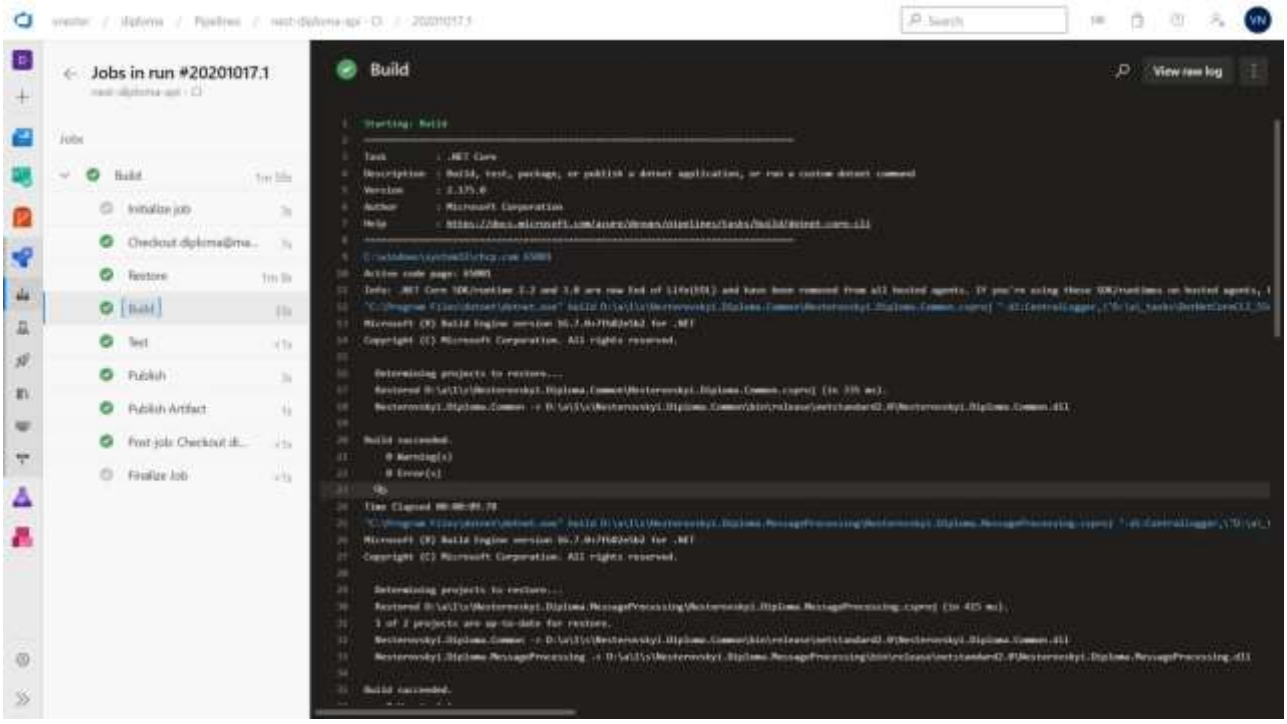

Рис. 4.3. Логування збірки та розгортання веб сервісу

4. Всі ресурси в *Azure*

Для підтримки роботи системи було створено окрему групу ресурсів в хмарі, і створено в ній:

План веб застосунків для балансування бюджету та конфігурації по цих сервісах.

Azure Event Hubs сервіс, який використовується як розподілена черга для зберігання повідомлень.

Власне *Azure Web App* який хостить публічне *API*.

Application Insights сервіс який дозволяє ефективно збирати й візуалізувати телеметрію та логи з застосунку публічного *API*.

Azure Web App який хостить підсистему асинхронної обробки повідомлень.

Екземпляр *Application Insights* для підсистеми обробки.

Azure Blob Storage для довгострокового зберігання користувацьких повідомлень.

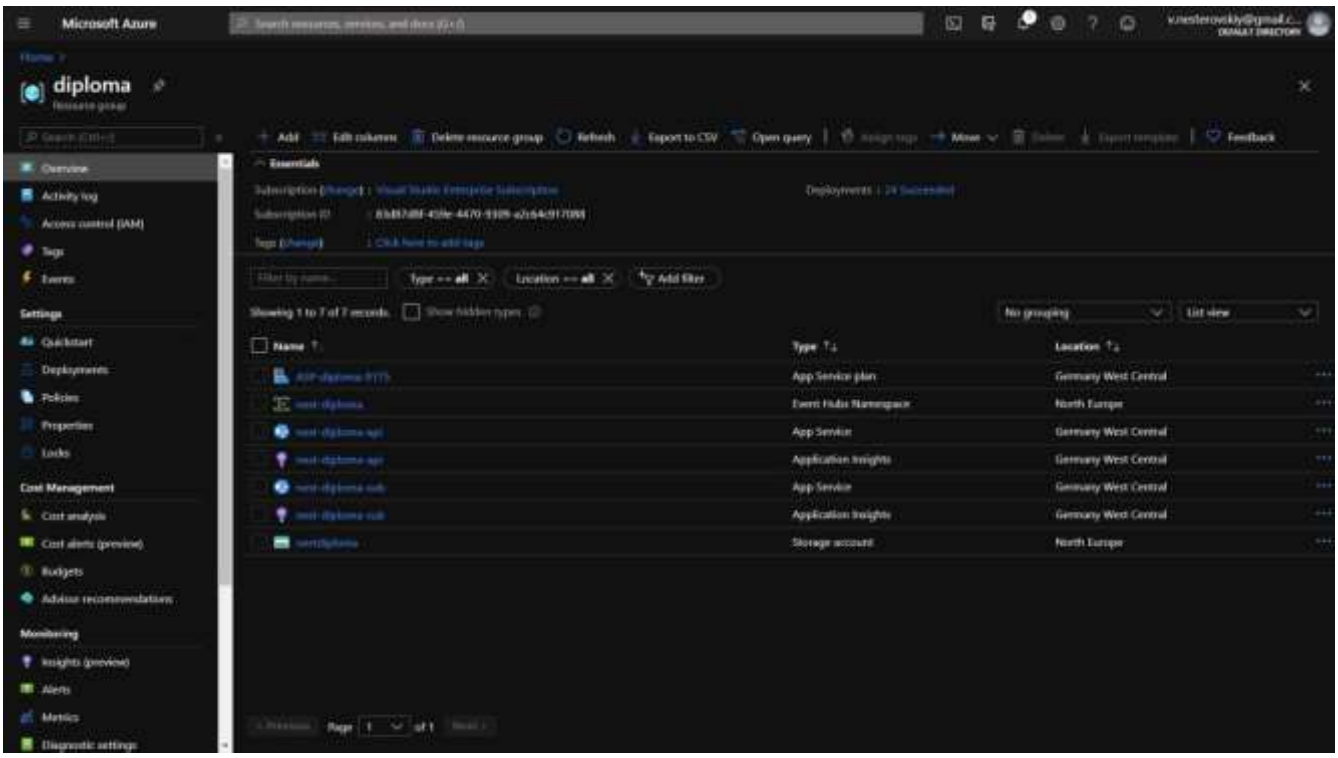

Рис. 4.4. Загальний список ресурсів які було розгорнуто для замірів

5. Панель керування публічним *API*

Панель керування дозволяє здійснювати основні операції по конфігурації та підтримки сервісу публічного *API*, такі як телеметрія, діагностика, деплой, конфігурація, автентифікація та авторизація, скейлінг та ін.

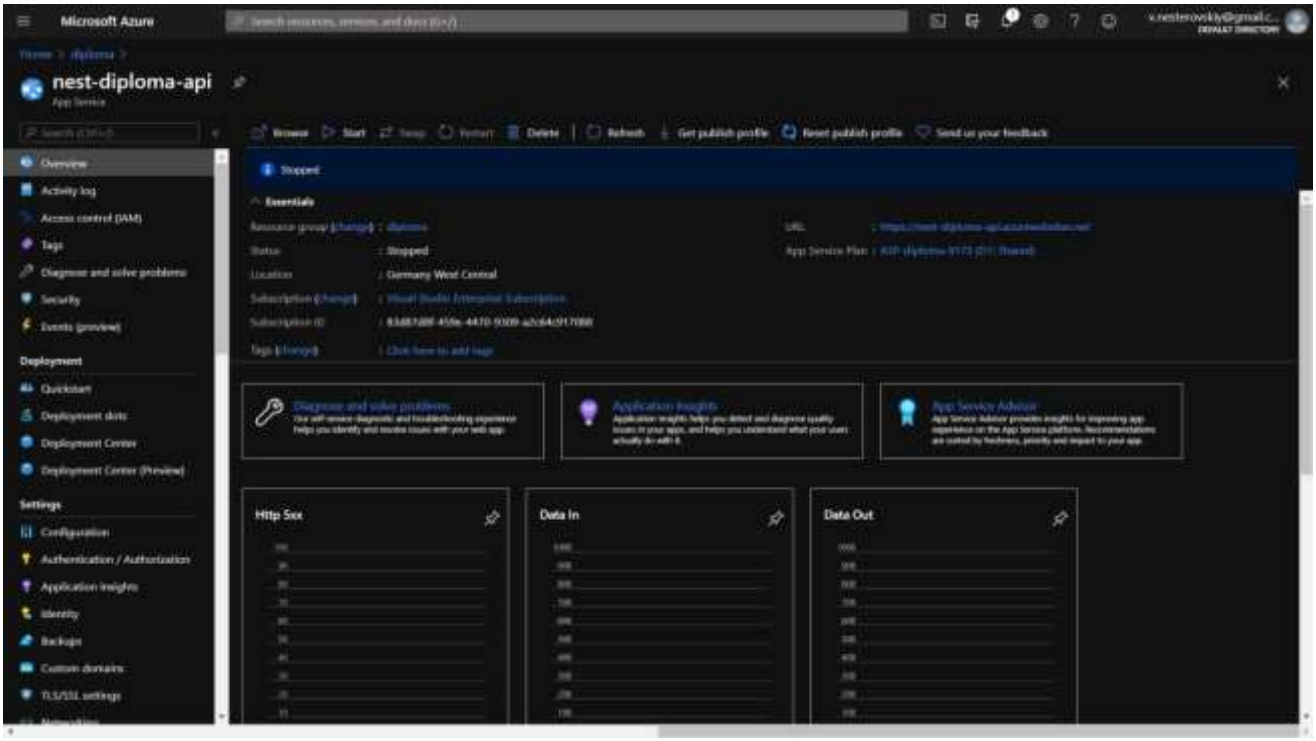

Рис. 4.5. Конфігурація публічного веб *API*

6. Вміст *Blob storage*

Azure Blob Storage має зручний інтерфейс для роботи з даними які знаходяться в цьому сховищі, також дозволяє гнучке керування роботою сховища.

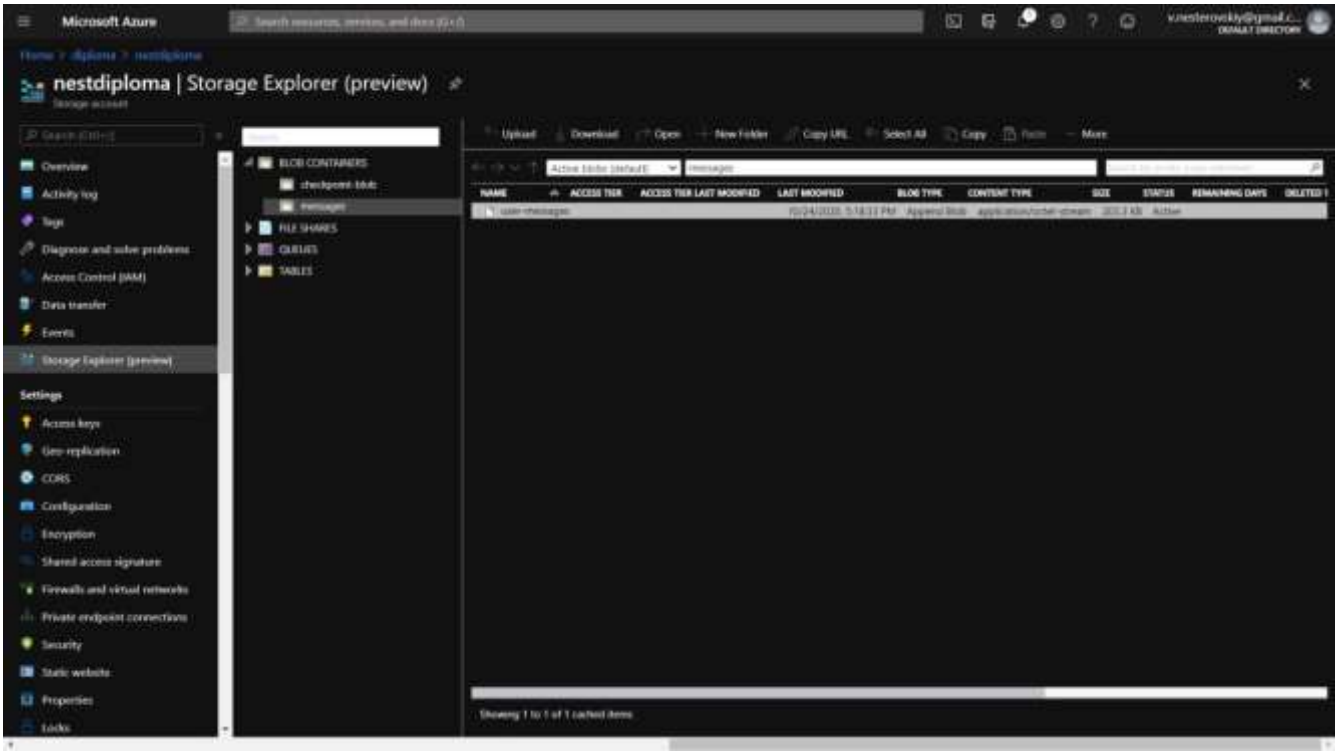

Рис. 4.6. Конфігурація та вміст *Azure Blob Storage*

7.Коміти

Для розробки системи застосовано систему версій *Git*, який дозволяє зручно та гнучко контролювати історію розробки, дивитися список комітів (змін) та їх авторство. *Git* в даному випадку використовується як підсистема *Azure DevOps*.

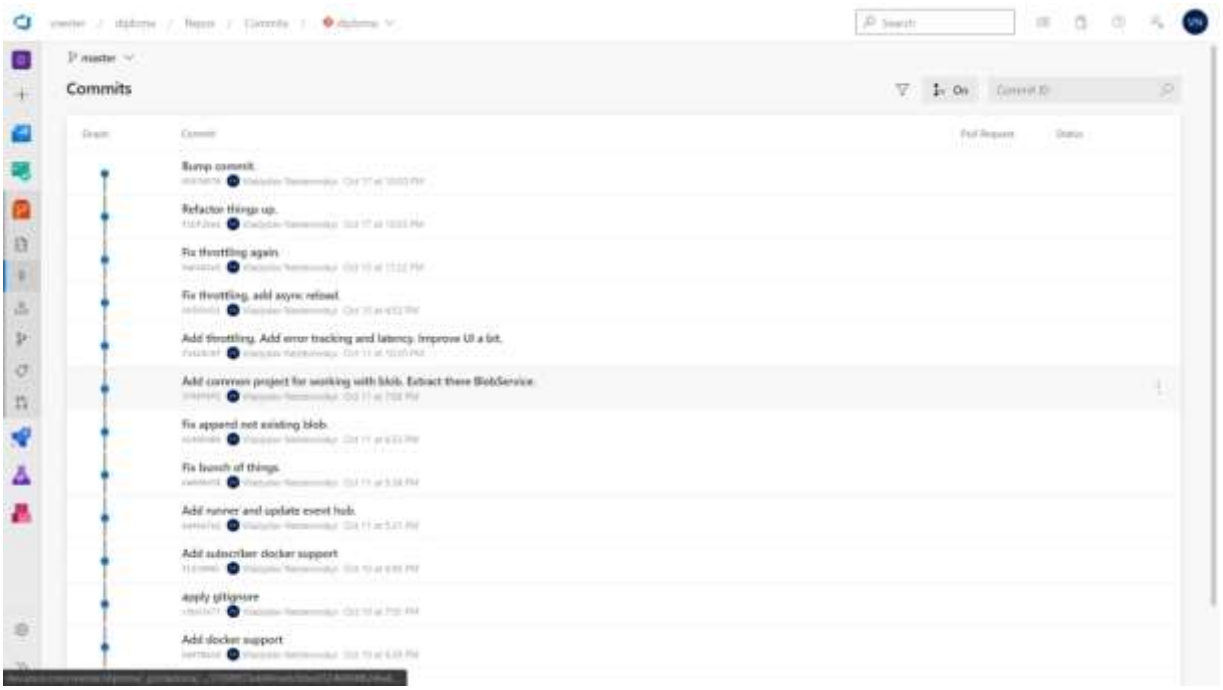

Рис. 4.7. Використані коміти для реалізації демо сервісу

8.Можливість скейлінгу

Панель Azure дозволяє на льоту скейлити застосунки як горизонтально, так і вертикально, що дає гнучкість в підтримці ефективності та продуктивності системи.

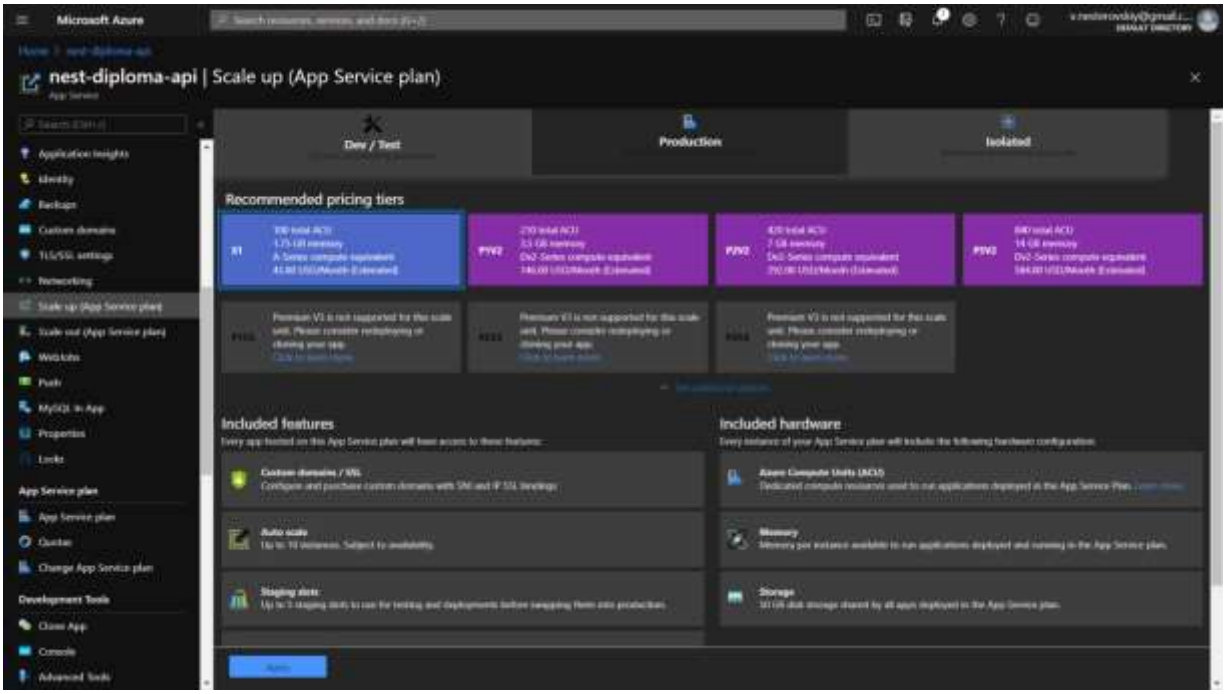

Рис. 4.8. Конфігурація вертикального та горизонтального скейлінгу

4.2 Аналіз результатів оптимізації

Після прогону тестів продуктивності веб сервісу була знята наступна телеметрія користуючись фреймворком *Azure AppInsights* та фреймворком візуалізації *Locust.*

Була знята телеметрія по ключовим нефункціональним показникам роботи веб сервісу, а саме по показникам кількості успішно опрацьованих запитів, кількості невдало опрацьованих запитів, показникам *RPS* тобто кількості запитів на секунду, показникам затримки відповіді.

До оптимізації показники кількості запитів на секунду в середньому складали 30-40 *RPS,* з показником до 80% неуспішно оброблених запитів на піку навантаження. Такий високий показник спричинений як надмірно високим навантаженням (500 користувачів одночасно), так і слабким ресурсам виділеним на роботу веб сервісу. На рис. 4.9 продемонстровано показники *RPS* для неоптимізованого веб сервісу.

Рис. 4.9. Показники *RPS* для неоптимізованого веб сервісу

Після оптимізації показники кількості запитів на секунду в середньому складали 60-80 *RPS*, з показником в 0% неуспшіно оброблених запитів на піку навантаження. Такий низький показник спричинений переносом навантаження на розподілену чергу, яка реалізує *lock-free append*. На рис. 4.10 продемонстровано показники *RPS* для оптимізованого веб сервісу.

Рис. 4.10. Показники *RPS* для оптимізованого веб сервісу

До оптимізації показники затримки відповіді в середньому складали 4-6 секунд, з піками до 20 секунд на обробку запитів на піку навантаження. Такий високий показник спричинений слабким ресурсом веб сервісу, оскільки багато паралельних запитів спричинили *thread pool* starvation. На рис. 4.11 продемонстровано показники latency для неоптимізованого веб сервісу.

Рис. 4.11. Показники *latency* для неоптимізованого веб сервісу

Після оптимізації показники затримки відповіді в середньому менше 2 секунд, з піками до 4-6 секунд на обробку запитів на піку навантаження. Такий низький показник спричинений тим, що thread pool starvation який вплинув на неоптимізований сервіс тут не діє, завдяки тому що корисне робота по обробці запиту була асинхронно відкладена в розподілену чергу, а, як відомо [25], асинхронність на рівні веб застосунку вирішує проблеми перевикористання пулу потоків. На рис. 4.12 продемонстровано показники latency для оптимізованого веб сервісу.

Рис. 4.12. Показники *latency* для неоптимізованого веб сервісу

Таким чином можна сказати про трикратне покращення показників latency та 80% покращення показників *RPS*. Кількість неуспішно оброблених запитів впала до нуля завдяки тому що клієнтам більше немає необхідності чекати поки буде виконана корисна робота, *HTTP* відповідь відразу надходить після успішного поміщення в розподілену чергу, а оскільки *SLA Azure Event Hubs* складає 99,99% то абсолютна більшість запитів буде оброблено успішно.

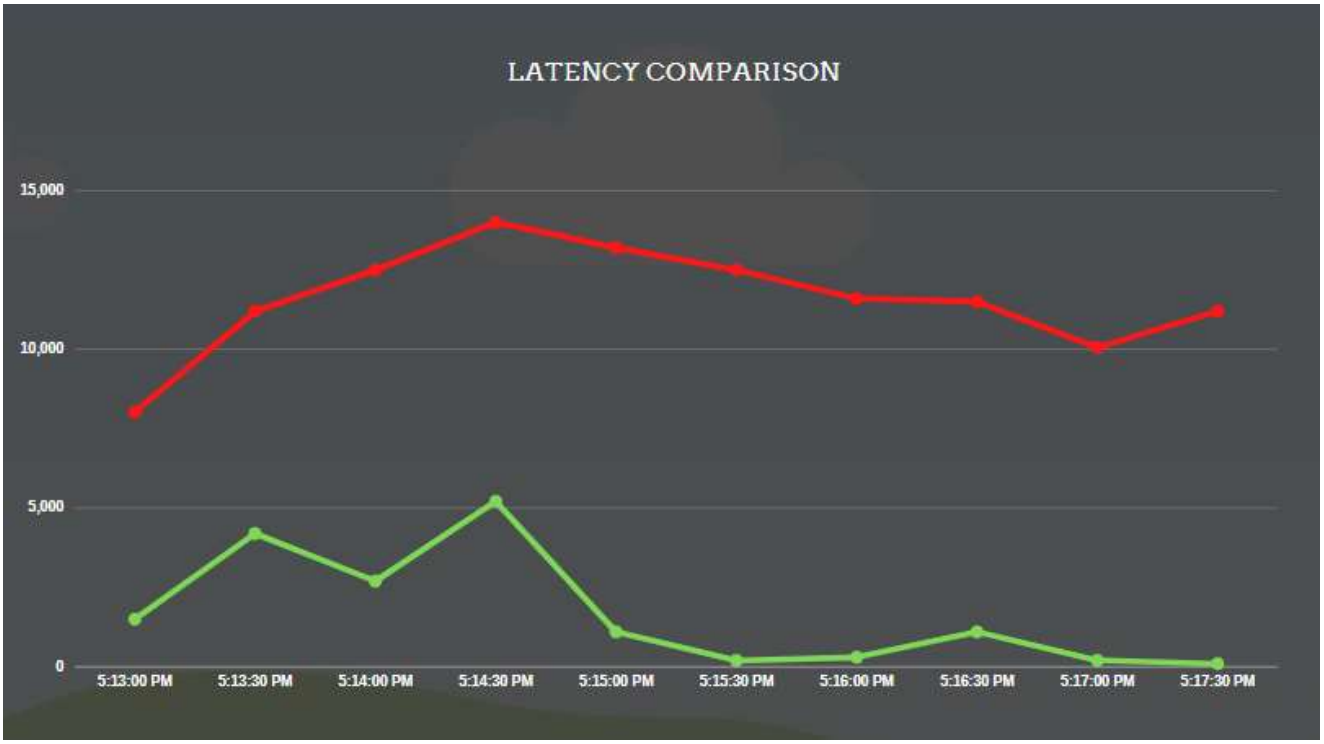

Рис. 4.13. Порівняння *latency* веб сервісу

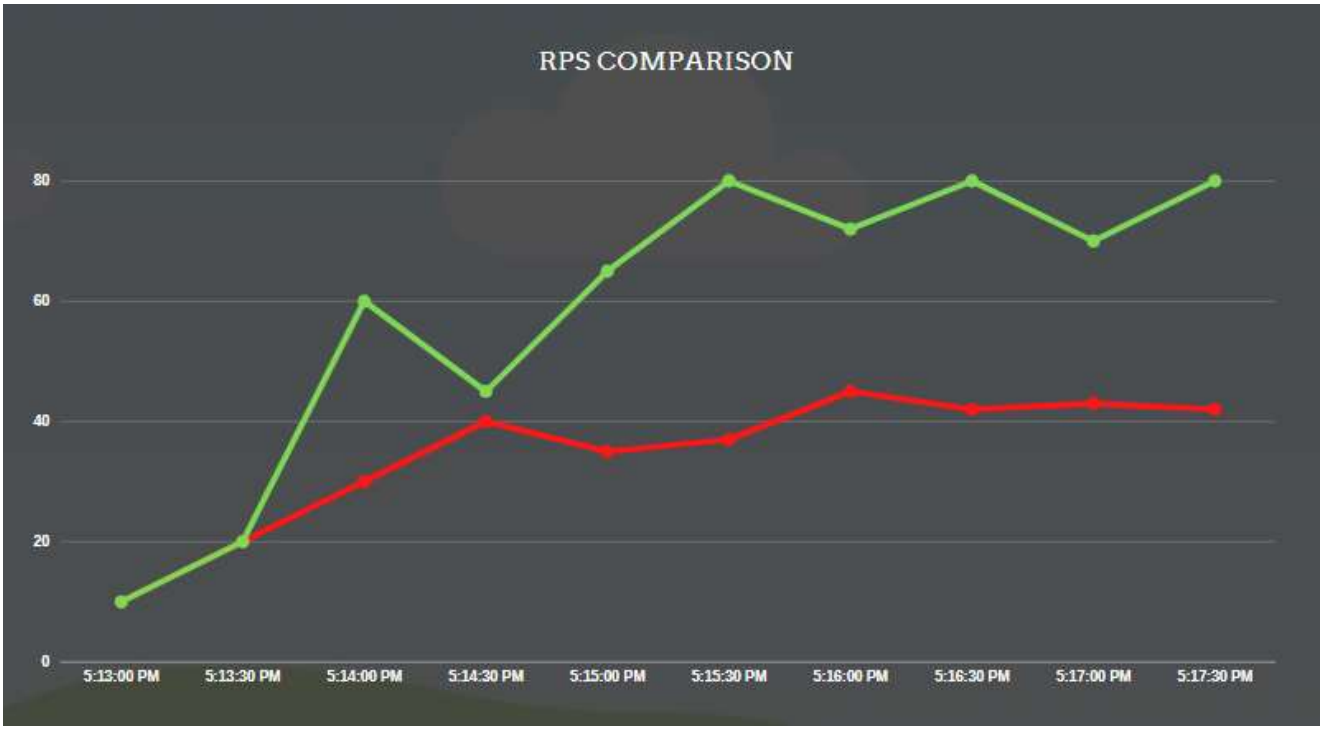

Рис. 4.14. Порівняння *RPS* веб сервісу

Висновки

Для замірів ефективності оптимізації, дослідженої в цій роботі, було розгорнуто 2 веб сервіси в хмарі *Azure*, та створено емуляцію навантаження від одного до п’ятиста одночасних користувачів.

Результати замірів показують значне покращення показників *RPS* та кількості успішно оброблених запитів. Введення в архітектуру асинхронності та розподіленої черги дозволило уникнути ситуації коли capacity веб сервісу напряму залежить від ресурсів які були на нього виділені. Завдяки гарантіям *SLA* розподіленої черги абсолютна більшість запитів була оброблена набагато швидше, оскільки корисна робота не виконувалася публічним веб *API*.

ВИСНОВКИ

Сучасні технології стрімко розвиваються, вирішуючи проблеми швидко зростаючої кількості користувачів мережі Інтернет. Проте це також породжує нові виклики в сфері розподілених та хмарних обчислень. Веб ресурси мають справу з нерівномірним навантаженням, яке спричиняє в одному випадку недостатню навантаженість веб сервісу частину часу, б'ючи по собівартості роботи ресурсу, а з іншого боку під час пікового навантаження веб сервісу спричиняючи відмову сервісу, погіршення характеристик *QoS* та ключових нефункціональних показників роботи веб сервісу, таких як збільшення кількості помилок під час оброблення запитів, таймаути, збільшення часу відповіді, та інші. Хмарні обчислення - це система, що складається переважно з трьох служб: програмне забезпечення як послуга (*SaaS),* інфраструктура як послуга (*IaaS*) та платформа як послуга (*PaaS*).

Вимоги щодо якості обслуговування, що стосуються веб-служб, стосуються якості як функціональних, так і нефункціональних характеристик веб-служб. Кожна категорія повинна мати набір кількісних параметрів або вимірювань. Вони включають час обслуговування, надійність, ціну виконання, доступність, масштабованість, доступність, продуктивність, ємність, цілісність, міцність / гнучкість, повнота, регулювання, репутація, вартість, точність, транзакційність, стабільність та *QoS*, пов'язані з безпекою. Проблеми веб-служб полягають у підвищенні вимог до якості обслуговування, щоб обґрунтувати їх обладнання для якості веб-послуг, і ця вимога повинна відповідати вибраним веб-службам.

Для вирішення вищевказаної проблеми було розглянуто існуючі підходи та архітектури веб застосунків.

Монолітна архітектура існувала з більш ранніх часів та має наступні переваги: Проста у розробці - мета сучасних засобів розробки та *IDE* полягає в підтримці розробки монолітних додатків. Просте для розгортання - просто потрібно розгорнути файл (або ієрархію каталогів) у відповідному середовищі

виконання. Просте у масштабі - можна масштабувати сервіс, запускаючи кілька копій застосунку за балансувальником навантаження. Проте недостатки переважають: велика монолітна база коду відлякує розробників, додаток може бути важко зрозуміти, немає жорстких меж модулів, якість коду з часом знижується, перевантажена *IDE*, перевантажений веб-контейнер, втрата часу на очікування запуску контейнера, а постійний деплой ускладнений.

Зазначену проблему вирішує розподілена мікросервісна архітектура – її переваги такі як забезпечення постійної доставки та розгортання великих складних підсистем, покращена підтримуваність, краща тестованість, краща можливість розгортання, послуги можна розгорнути незалежно, кожна команда може розробляти, тестувати, застосовувати та масштабувати свої послуги незалежно від усіх інших команд. Також кожен мікросервіс порівняно невеликий, тому розробнику легше зрозуміти, а *IDE* швидше робить розробників більш продуктивними. В той час основними недоліками розподілених систем є складне усунення несправностей, менша підтримка програмного забезпечення, високі витрати на мережеву інфраструктуру, проблеми безпеки.

На відміну від централізованих систем, розподілені програмні системи додають новий рівень складності до і без того складної проблеми проектування програмного забезпечення, але вирішують багато інших проблем та краще пристосовані до ситуації з нерівномірним навантаженням завдяки нативній можливості скейлінгу.

Для оптимізації була вибрана розподілена черга як компонент архітектури, оскільки розподілена черга дозволяє добитися виконання ключових вимог для покращення показників затримки відповіді та спроможності обробляти сотні запитів в секунду. Іншим необхідним компонентом є пакетна обробка. Застосувавши обидва цих компонента, можна значно покращити характеристики веб сервісу який їх застосовує. Черги повідомлень дозволяють різним частинам системи спілкуватися та обробляти операції асинхронно, таким чином знімаючи навантаження з частин системи які бачить користувач (*customer facing APIs*).

Також перевага розподіленої черги полягає в тому, що вона дозволяє розподілити обробку даних на декілька екземплярів споживача, що дозволяє масштабувати процес обробки під час зростання навантаження.

Для оптимізації мікросервісної архітектури з використанням розподіленої черги та пакетної обробки була вибрана хмара *Microsoft Azure*.

В ній розгорнуто застосунки побудовані за допомогою мікросервісної архітектури та веб фреймворку *ASP.NET Core*. Застосунок являє собою месенджер, тобто застосунок обміну повідомленнями між користувачами. В ролі розподіленої черги був вибраний сервіс *Azure Event Hubs*, який реалізує протокол *AMQP*, та є реалізацією гарантій *ACID* та *at-least-once* доставки. Для зберігання результатів було вибрано *Azure Blob Storage*. Оскільки месенджер це сервіс яким користувачі користуються відносно часто, це тип сервісу який найбільш чутливий до проблеми яка вирішена в даній роботі.

Процес обробки данних асинхронний, та полягає в потоку даних від публічного *API* до акумуляції в розподіленій черзі для подальшої обробки модулем Consumer та зберігання результатів обчислення над пакетами даних в персистентному сховищі. Таким чином зберігаються високий рівень гарантій при використанні черги та можливість як скейлити обробку даних так і чекати спаду навантаження для кращої утилізації ресурсів хмари.

Для замірів ефективності оптимізації, дослідженої в цій роботі, було розгорнуто 2 веб сервіси в хмарі *Azure*, та створено емуляцію навантаження сотень одночасних користувачів.

Результати замірів показують значне покращення ключових показників *RPS* та кількості успішно оброблених запитів. Після введення в архітектуру аспекту асинхронності та компоненту розподіленої черги дозволило уникнути ситуації коли capacity веб сервісу напряму залежить від ресурсів які були на нього виділені. Завдяки гарантіям *SLA* розподіленої черги абсолютна більшість запитів була оброблена набагато швидше, оскільки корисна робота не виконувалася публічним веб *API*. Запропонована пакетна обробка дозволила обробити більшу ширину потоку даних без збільшення потужностей обробника повідомлень з черги.

Оптимізації запропоновані в даній роботі рекомендується застосовувати при розробці веб сервісів, потреба в функціоналі яких корелює з часом доби або року, наприклад онлайн-магазини, месенджери, стрімінгові платформи, медіа платформи та інші. Завдяки використанню запропонованих підходів до системного дизайну архітектури веб сервісів очікується покращення ключових характеристик роботи сервісу що напряму впливає на задоволення користувачів роботи сервісу, *COGS* (*cost of goods sold*, собівартість інфраструктури), та таким чином на прибутковість та популярність сервісу.

ПОСИЛАННЯ

1. https://coderoad.ru/14623173
2. https://infostart.ru/1c/articles/583403/
3. https://coderoad.ru/123533344
4. Vince Barnes Where Do I Put My Application?/ Vince Barnes // Non-Technical Introduction. – 2009. – #2. – 1
5. https://searchunifiedcommunications.techtarget.com/definition/QoS-Quality-of-Service
6. Pete Zaborszky The 5 Best Ways to Build a Web app in 2014 [Web resource]. – Access mode: http://www.make-a-web-site.com/5-best-ways-build-website-2014/
7. https://www.networkcomputing.com/networking/basics-qos
8. https://www.cloudflare.com/learning/ddos/glossary/open-systems-interconnection-model-osi/
9. Joe Burns To Use or Not to Use a Database? / Joe Burns // Database Basics. – 2009. – #1. – 4
10. Curtis Dicken Top 5 Databases for Web Developers / Curtis Dicken // Database/SQL Primer. – 2011. – 1
11. https://www.forcepoint.com/cyber-edu/osi-model
12. https://www.imperva.com/learn/application-security/osi-model/
13. Glenn Fowler A Flat File Database Query Language / Glenn Fowler // AT&T Labs Research. – 1994. – 10
14. SQL Server [Web resource]. – Microsoft, 2014. – Access mode: http://www.microsoft.com/en-us/server-cloud/products/sql-server/
15. Introduction to the Postgres Database [Web resource]. – Postgres Corporation. – Access mode: http://docs.oracle.com/cd/B19306_01/server.102/b14220/intro.htm
16. Content management system [Web resource]. – DocForge, 2010. – Access mode: http://docforge.com/wiki/Content_management_system
17. https://azure.microsoft.com/en-us/

18. https://docs.microsoft.com/en-us/azure/?product=featured
19. https://restfulapi.net/
20. merriam-webster.com/dictionary/rest
21. https://itglobal.com/ru-ru/company/glossary/qos/
22. https://lectureswww.readthedocs.io/6.www.sync/3.framework/pyramid/5.1.rest.html
23. https://webkyrs.info/page/chto-takoe-rest-api
24. http://www.restapitutorial.ru/
25. https://habr.com/ru/hub/azure/
26. John Richer Azure in Simple // Labs Research. – 2008. – 55
27. Scott Klein, Herve Roggero Pro SQL Database for Windows Azure. - 2013 - 283
28. Cloud Computing Principles and Paradigms Rajkumar Byya, James Broberg, Andrzej Goscinsk. – 2010. – 542
29. Cloud Computing: Concepts, Technology & Architecture Thomas Erl. – 2013. – 528
30. Mastering Zoho CRM Shabdar Ali. – 2017. – 328
31. SugarCRM Building on John Martic. – 2012. – 325
32. Microsoft Dynamics CRM Administration Mathew Witterman, Geoff Ables. – 2011. – 712
33. Microsoft Dynamics by Marc Wolenik. – 2013. – 1176
34. The Guide for Business and Technology Managers by Vivek Kale. – 2013. – 1176
35. Scott Kostojohn,Mathew Johnso and Brian Paulen CRM Fundamentals. – 2013. – 213
36. Martin, D., Burstein, M., Hobbs, J., Lassila, O., McDermott, D., McIlraith. – 2004. – 121
37. Joel Scott Michael DelIsa Microsoft. – 2015. – 11
38. Sriram Krishnan Windows Azure O'Reilly. – 2011. – 556
39. Опис хмарних сервісів та роботи з ними https://en.wikipedia.org/wiki/Cloud_computing

40. Cloud Computing Principles and Paradigms Rajkumar Byya, James Broberg, Andrzej Goscinski . – 2008. – 123
41. https://docs.microsoft.com/en-us/azure/event-hubs/event-hubs-about

ДОДАТОК А.

ТЕКСТ ПРОГРАМИ ДЛЯ ЗАМІРІВ ЕФЕКТИВНОСТІ ОПТИМІЗАЦІЇ

Файл з специфікаєю проекту.

```
<Project Sdk="Microsoft.NET.Sdk">

  <PropertyGroup>
    <TargetFramework>netstandard2.0</TargetFramework>
    <LangVersion>8.0</LangVersion>
  </PropertyGroup>

  <ItemGroup>
    <PackageReference Include="Azure.Storage.Blobs" Version="12.6.0" />
    <PackageReference Include="System.ComponentModel.Annotations" Version="4.7.0" />
  </ItemGroup>

</Project>
```

Extension method для отримання значення DisplayAttribute з об’єктів типу enum.

```
        public static string GetDisplayName(this Enum enumValue)
        {
            return enumValue.GetType()
                            .GetMember(enumValue.ToString())
                            .First()
                            .GetCustomAttribute<DisplayAttribute>()
                            .GetName();
        }
    }
```

Методи для роботи з Azure Blob Storage, використовується і в public API і в asynchronous processing.

```
        private const string ContainerName = "messages";
        private const string BlobName = "user-messages";

        private readonly BlobClient _blobClient;
        private readonly AppendBlobClient _appendClient;

        public StorageBlobService(string connectionString)
        {
            var containerClient = new BlobContainerClient(connectionString, ContainerName);
            containerClient.CreateIfNotExists();
```

```
            _blobClient = containerClient.GetBlobClient(BlobName);
            _appendClient = containerClient.GetAppendBlobClient(BlobName);
        }

        public async IAsyncEnumerable<string> DownloadAsync([EnumeratorCancellation] Cancella
tionToken token)
        {
            var exists = await _blobClient.ExistsAsync();
            if(!exists)
            {
                await Task.Delay(1000);
                yield break;
            }

            var resp = await _blobClient.DownloadAsync(token);
            if (resp.GetRawResponse().Status >= 400)
                throw new Exception("Invalid request to blob storage");

            using var reader = new StreamReader(resp.Value.Content);

            while (!reader.EndOfStream)
            {
                yield return await reader.ReadLineAsync();
            }
        }

        public async Task AppendAsync(string val)
        {
            await _appendClient.CreateIfNotExistsAsync();

            var bytes = Encoding.UTF8.GetBytes(val);
            var stream = new MemoryStream(bytes);

            await _appendClient.AppendBlockAsync(stream);
        }
    }
```

Емулятор обмеження ресурсів веб сайту – для наглядності демо було обмежено одночасне навантаження в 20 запитів, і додав таймаут в 5 секунд для всіх інших запитів.

```
        private const int RateLimit = 20;
        private const int Timeout = 5000;

        private readonly SemaphoreSlim _limiter;

        public ThrottlingRequestHandler()
        {
            _limiter = new SemaphoreSlim(RateLimit, RateLimit);
        }

        public async Task Throttle(Func<Task> action)
        {
            await ThrottleAsync();
```

```
            try
            {
                await action();
            }
            finally
            {
                _limiter.Release();
            }
        }

        public async Task<T> Throttle<T>(Func<Task<T>> action)
        {
            await ThrottleAsync();

            try
            {
                return await action();
            }
            finally
            {
                _limiter.Release();
            }
        }

        private async Task ThrottleAsync()
        {
            var ok = await _limiter.WaitAsync(Timeout);
            if (!ok)
                throw new TimeoutException();
        }
    }
}
```

Модель об’єкту користувацького повідомлення з необхідними метаданими.

```
namespace Nesterovskyi.Diploma.MessageProcessing
{
    public class AsyncMessage
    {
        public Message Message { get; set; }
        public ProcessingMode Mode { get; set; }

        public enum ProcessingMode
        {
            Direct = 1,
            Batching = 2
        }
    }
}
```

Метод який обробляє повідомлення шляхом розміщення повідомленнь в розподілену чергу.

```
        private readonly EventHubProducerClient _client;

        public AsyncMessageProcessor(IOptions<ConfigurationOptions> options)
        {
             _client = new EventHubProducerClient(options.Value.EventHubsConnectionString, opt
ions.Value.EventHubName);
        }

        public async Task ProcessAsync(Message message)
        {
            var json = JsonConvert.SerializeObject(message);
            var bytes = Encoding.UTF8.GetBytes(json);

            await _client.SendAsync(new[] { new EventData(bytes) });
        }

    }
```

Метод який обробляє повідомлення в пакетному режимі – буферизуючи повідомлення в пакети перед обробкою.

```
        private const int BatchSize = 20;
        private readonly BlockingCollection<Message> _messages;
        private readonly StorageBlobService _blobService;
        static readonly SemaphoreSlim _semaphoreSlim = new SemaphoreSlim(1, 1);

        public static int ProcessedCount = 0;

        public BatchingMessageProcessor(IOptions<ConfigurationOptions> options)
        {
            _messages = new BlockingCollection<Message>(BatchSize);

             _blobService = new StorageBlobService(options.Value.MessageBlobStorageConnectionS
tring);
        }

        protected override async Task ProcessInternalAsync(Message message)
        {
            bool added = false;
            while (!added)
            {
                added = _messages.TryAdd(message);

                if (!added)
                {
                    var result = new StringBuilder();

                    await _semaphoreSlim.WaitAsync();

                    try
                    {
                        if (_messages.Count < _messages.BoundedCapacity)
                            continue;

                        while (_messages.TryTake(out var msg))
                        {
                            var line = JsonConvert.SerializeObject(msg);
```

```
                    result.AppendLine(line);
                }

                await _blobService.AppendAsync(result.ToString());

                ProcessedCount += _messages.BoundedCapacity;
            }
            finally
            {
                _semaphoreSlim.Release();
            }
        }
```

Модель об’єкту конфігурації, з необхідними метаданими для з’єднання з Azure Blob Storage, Azure Event Hubs.

```
namespace Nesterovskyi.Diploma.MessageProcessing
{
    public class ConfigurationOptions
    {
        public string EventHubsConnectionString { get; set; }
        public string EventHubName { get; set; }
        public string MessageBlobStorageConnectionString { get; set; }
        public string EventHubCheckpointBlobConnectionString { get; set; }
        public string EventHubCheckpointBlobName { get; set; }
    }
}
```

Інтерфейс для обробки повідомленнь.

```
namespace Nesterovskyi.Diploma.MessageProcessing
{
    public interface IMessageProcessor
    {
        Task ProcessAsync(Message message);
    }
}
```

Модель користувацького повідомлення.

```
namespace Nesterovskyi.Diploma.MessageProcessing
{
    public class Message
    {
        public Guid UserId { get; set; }
        public string UserMessage { get; set; }
    }
}
```

Базовий клас для всіх обробників повідомлень який містить в собі логіку тротлінгу повідомленнь.

```
namespace Nesterovskyi.Diploma.MessageProcessing
{
    public abstract class MessageProcessorBase : IMessageProcessor
    {
        private ThrottlingRequestHandler _throttler;

        protected MessageProcessorBase()
        {
            _throttler = new ThrottlingRequestHandler();
        }

        public Task ProcessAsync(Message message)
        {
            return _throttler.Throttle(() => ProcessInternalAsync(message));
        }

        protected abstract Task ProcessInternalAsync(Message message);
    }
}
```

Клас який реалізує патерн Factory для вибору обробника повідомлень на основі версії, яку передає клієнт.

```
    public class MessageProcessorFactory
    {
        private readonly Dictionary<int, IMessageProcessor> _processors;

        public MessageProcessorFactory(IOptions<ConfigurationOptions> options)
        {
            _processors = new Dictionary<int, IMessageProcessor>()
            {
                [1] = new SimpleMessageProcessor(options),
                [2] = new AsyncMessageProcessor(options),
                [3] = new BatchingMessageProcessor(options)
            };
        }

        public IMessageProcessor GetProcessor(int version)
        {
            if (_processors.TryGetValue(version, out var processor))
                return processor;

            throw new ArgumentException($"Processor version {version} not found");
        }
    }
}
```

Файл опису проекту.

```
<Project Sdk="Microsoft.NET.Sdk">
```

```
    <PropertyGroup>
      <TargetFramework>netstandard2.0</TargetFramework>
      <LangVersion>8.0</LangVersion>
    </PropertyGroup>

    <ItemGroup>
      <PackageReference Include="Azure.Messaging.EventHubs" Version="5.2.0" />
      <PackageReference Include="Microsoft.Extensions.Logging.Abstractions" Version="3.1.8" />
      <PackageReference Include="Microsoft.Extensions.Options" Version="3.1.8" />
      <PackageReference Include="Newtonsoft.Json" Version="12.0.3" />
    </ItemGroup>

    <ItemGroup>
       <ProjectReference Include="..\Nesterovskyi.Diploma.Common\Nesterovskyi.Diploma.Common.csproj" />
    </ItemGroup>


  </Project>
```

Обробник повідомлень який власне реалізує бізнес логіку обробки і складає результат в Azure Blob Storage.

```
        public SimpleMessageProcessor(IOptions<ConfigurationOptions> options)
        {
             _blobService = new StorageBlobService(options.Value.MessageBlobStorageConnectionString);
        }

        protected override async Task ProcessInternalAsync(Message message)
        {
            // Simulate long running job.
            await Task.Delay(3000);

            var line = JsonConvert.SerializeObject(message) + Environment.NewLine;

            await _blobService.AppendAsync(line);
        }
    }
```

Клас який реалізує патерн MVC і є публічним ендпоінтом для відправлення повідомлень.

```
namespace Nesterovskyi.Diploma.Publisher.Controllers
{
    [ApiController]
    [Route("[controller]")]
    public class MessageController : ControllerBase
    {
```

```
            private readonly ILogger<MessageController> _logger;
            private readonly MessageProcessorFactory _messageProcessorFactory;

            public MessageController(
                ILogger<MessageController> logger,
                MessageProcessorFactory factory)
            {
                _logger = logger;
                _messageProcessorFactory = factory;
            }

            [HttpPost]
            public Task ProcessAsync([FromBody] MessageRequest request)
            {
                _logger.LogInformation("New message from user {userId}", request.UserId);

                var processor = _messageProcessorFactory.GetProcessor(request.Version);

                 return processor.ProcessAsync(new Message { UserId = request.UserId, UserMessage
= request.Message});
            }
        }
    }
```

Файл конфігурації публічного API.

```
{
  "iisSettings": {
    "windowsAuthentication": false,
    "anonymousAuthentication": true,
    "iisExpress": {
      "applicationUrl": "http://localhost:10483"
    }
  },
  "$schema": "http://json.schemastore.org/launchsettings.json",
  "profiles": {
    "IIS Express": {
      "commandName": "IISExpress",
      "launchBrowser": true,
      "launchUrl": "weatherforecast",
      "environmentVariables": {
        "ASPNETCORE_ENVIRONMENT": "Development"
      }
    },
    "Nesterovskyi.Diploma.Publisher": {
      "commandName": "Project",
      "launchBrowser": true,
      "launchUrl": "weatherforecast",
      "environmentVariables": {
        "ASPNETCORE_ENVIRONMENT": "Development"
      },
      "applicationUrl": "http://localhost:5001;http://localhost:5000"
    },
    "Docker": {
      "commandName": "Docker",
      "launchBrowser": true,
      "launchUrl": "{Scheme}://{ServiceHost}:{ServicePort}/weatherforecast",
      "publishAllPorts": true
```

```
    }
  }
}
```

Модель запиту який відправляє клієнт.

```
namespace Nesterovskyi.Diploma.Publisher
{
    public class MessageRequest
    {
        public int Version { get; set; }
        public Guid UserId { get; set; }
        public string Message { get; set; }
    }
}
```

Клас в якому міститься метод Main, стартова точка програми.

```
namespace Nesterovskyi.Diploma.Publisher
{
    public class Program
    {
        public static void Main(string[] args)
        {
            CreateHostBuilder(args).Build().Run();
        }

        public static IHostBuilder CreateHostBuilder(string[] args) =>
            Host.CreateDefaultBuilder(args)
                .ConfigureWebHostDefaults(webBuilder =>
                {
                    webBuilder.UseStartup<Startup>();
                });
    }
}
```

Файл конфігурації ASP.NET Core фреймворку.

```
namespace Nesterovskyi.Diploma.Publisher
{
    public class Startup
    {
        public Startup(IWebHostEnvironment env)
        {
            var builder = new ConfigurationBuilder()
                .SetBasePath(env.ContentRootPath)
                .AddJsonFile("appsettings.json", optional: true, reloadOnChange: true);

            Configuration = builder.Build();
        }

        public IConfiguration Configuration { get; }
```

```
            // This method gets called by the runtime. Use this method to add services to the con
tainer.
            public void ConfigureServices(IServiceCollection services)
            {
                services.AddControllers();

                services.AddSingleton<MessageProcessorFactory>();

                // Add functionality to inject IOptions<T>
                services.AddOptions();

                // Add our Config object so it can be injected
                services.Configure<ConfigurationOptions>(Configuration.GetSection("Connections"));
            }

            // This method gets called by the runtime. Use this method to configure the HTTP requ
est pipeline.
            public void Configure(IApplicationBuilder app, IWebHostEnvironment env)
            {
                if (env.IsDevelopment())
                {
                    app.UseDeveloperExceptionPage();
                }

                app.UseHttpsRedirection();

                app.UseRouting();

                app.UseAuthorization();

                app.UseEndpoints(endpoints =>
                {
                    endpoints.MapControllers();
                });
            }
        }

        public class ErrorMiddleware
        {
            private RequestDelegate _delegate;

            public ErrorMiddleware(RequestDelegate request)
            {
                _delegate = request;
            }
        }
    }
```

Методи які реалізують MVC патерн і є ендпоінтом для контролю підсистеми обробки.

```
            private Task _subscriberTask;
            private readonly MessageProcessorFactory _factory;
            private CancellationTokenSource _source;
            private readonly IOptions<ConfigurationOptions> _options;
```

```
        public ControlController(MessageProcessorFactory factory, IOptions<ConfigurationOptio
ns> options)
        {
            _factory = factory;
            _options = options;
        }

        [HttpGet]
        public int Get()
        {
            return BatchingMessageProcessor.ProcessedCount;
        }

        [HttpPost]
        public void Run()
        {
            _source = new CancellationTokenSource();

            _subscriberTask = Task.Factory.StartNew(ProcessAsync);
        }

        [HttpDelete]
        public void Stop()
        {
            if (_subscriberTask == null)
                return;

            _source.Cancel();

            _subscriberTask = null;
        }

        private async Task ProcessAsync()
        {
            // Read from the default consumer group: $Default
            string consumerGroup = EventHubConsumerClient.DefaultConsumerGroupName;

            // Create a blob container client that the event processor will use
            var storageClient = new BlobContainerClient(_options.Value.EventHubCheckpointBlob
ConnectionString, _options.Value.EventHubCheckpointBlobName);

            await storageClient.CreateIfNotExistsAsync();

            // Create an event processor client to process events in the event hub
            var processor = new EventProcessorClient(storageClient, consumerGroup, _options.V
alue.EventHubsConnectionString, _options.Value.EventHubName);

            // Register handlers for processing events and handling errors
            processor.ProcessEventAsync += ProcessEventHandler;
            processor.ProcessErrorAsync += ProcessErrorHandler;

            // Start the processing
            await processor.StartProcessingAsync();

            try
            {
                await Task.Delay(Timeout.Infinite, _source.Token);
            }
            finally
            {
                await processor.StopProcessingAsync();
```

```
            }
        }


        private async Task ProcessEventHandler(ProcessEventArgs eventArgs)
        {
            var data = eventArgs.Data.Body;
            var json = Encoding.UTF8.GetString(data.ToArray());
            var msg = JsonConvert.DeserializeObject<Message>(json);

            var processor = _factory.GetProcessor(3);
            await processor.ProcessAsync(msg);

            // Update checkpoint in the blob storage so that the app receives only new events
the next time it's run
            await eventArgs.UpdateCheckpointAsync(eventArgs.CancellationToken);
        }
```

Файл ендпоінту для конфігурації обробки.

```
[ApiController]
[Route("[controller]")]
public class SettingsController : ControllerBase
{
    [HttpPost]
    public void Post(int mode)
    {
        ProcessingSettings.ProcessingMode = (AsyncMessage.ProcessingMode)mode;
    }
}
```

Файл конфігурації веб апі.

```
{
  "$schema": "http://json.schemastore.org/launchsettings.json",
  "iisSettings": {
    "windowsAuthentication": false,
    "anonymousAuthentication": true,
    "iisExpress": {
      "applicationUrl": "http://localhost:11373",
      "sslPort": 44360
    }
  },
  "profiles": {
    "IIS Express": {
      "commandName": "IISExpress",
      "launchBrowser": true,
      "launchUrl": "weatherforecast",
      "environmentVariables": {
        "ASPNETCORE_ENVIRONMENT": "Development"
      }
    },
    "Nesterovskyi.Diploma.Subscriber": {
```

```
      "commandName": "Project",
      "launchBrowser": true,
      "launchUrl": "weatherforecast",
      "applicationUrl": "https://localhost:5001;http://localhost:5000",
      "environmentVariables": {
        "ASPNETCORE_ENVIRONMENT": "Development"
      }
    }
  }
}
{
  "Logging": {
    "LogLevel": {
      "Default": "Information",
      "Microsoft": "Warning",
      "Microsoft.Hosting.Lifetime": "Information"
    }
  }
}
{
  "Logging": {
    "LogLevel": {
      "Default": "Information",
      "Microsoft": "Warning",
      "Microsoft.Hosting.Lifetime": "Information"
    }
  }
```

Файл опису проекту.

```
<Project Sdk="Microsoft.NET.Sdk.Web">

  <PropertyGroup>
    <TargetFramework>netcoreapp3.1</TargetFramework>
  </PropertyGroup>

  <ItemGroup>
     <ProjectReference Include="..\Nesterovskyi.Diploma.MessageProcessing\Nesterovskyi.Diploma
.MessageProcessing.csproj" />
  </ItemGroup>

  <ItemGroup>
    <PackageReference Include="Azure.Messaging.EventHubs.Processor" Version="5.2.0" />
  </ItemGroup>
</Project>
```

Файл статичних налаштуваннь обробки.

```
namespace Nesterovskyi.Diploma.Subscriber
{
    public class ProcessingSettings
    {
        public static AsyncMessage.ProcessingMode ProcessingMode = AsyncMessage.ProcessingMod
e.Direct;
    }
```

Стартовий файл програми з методом Main.

```
namespace Nesterovskyi.Diploma.Subscriber
{
    public class Program
    {
        public static void Main(string[] args)
        {
            CreateHostBuilder(args).Build().Run();
        }

        public static IHostBuilder CreateHostBuilder(string[] args) =>
            Host.CreateDefaultBuilder(args)
                .ConfigureWebHostDefaults(webBuilder =>
                {
                    webBuilder.UseStartup<Startup>();
                });
    }
}
```

Файл налаштування ASP.NET Core фреймворку.

```
    public class Startup
    {
        public Startup(IConfiguration configuration)
        {
            Configuration = configuration;
        }

        public IConfiguration Configuration { get; }

        // This method gets called by the runtime. Use this method to add services to the con
tainer.
        public void ConfigureServices(IServiceCollection services)
        {
            services.AddControllers();

            services.AddSingleton<MessageProcessorFactory>();

            // Add functionality to inject IOptions<T>
            services.AddOptions();

            // Add our Config object so it can be injected
            services.Configure<ConfigurationOptions>(Configuration.GetSection("Connections"));
        }


        // This method gets called by the runtime. Use this method to configure the HTTP requ
est pipeline.
        public void Configure(IApplicationBuilder app, IWebHostEnvironment env)
        {
            if (env.IsDevelopment())
            {
                app.UseDeveloperExceptionPage();
            }

            app.UseHttpsRedirection();
```

```
            app.UseRouting();

            app.UseAuthorization();

            app.UseEndpoints(endpoints =>
            {
                endpoints.MapControllers();
            });
        }
    }
}
```