Міністерство освіти і науки України
Харківський національний університет радіоелектроніки

Факультет комп'ютерних наук
(повна назва)

Кафедра програмної інженерії
(повна назва)

КВАЛІФІКАЦІЙНА РОБОТА
Пояснювальна записка

Рівень вищої освіти другий (магістерський)

Порівняльний аналіз оптимізації архітектури обробки сповіщень на стороні Back-end: синхронного REST API та асинхронного підходу з Apache Kafka
(тема)

Виконав:
здобувач другого року навчання
групи ІПЗм-23-4
Владислав СТАРІЧЕНКО
(власне ім'я, прізвище)

Спеціальність 121 – Інженерія програмного забезпечення
(код і повна назва спеціальності)

Тип програми освітньо-наукова

Керівник доц. Олексій ЛАНОВИЙ
(посада, власне ім'я, прізвище)

Допускається до захисту

Зав. кафедри _____ Кирило СМЕЛЯКОВ
(підпис) (прізвище, ініціали)

2025 р.

Харківський національний університет радіоелектроніки

Факультет комп'ютерних наук
Кафедра програмної інженерії
Рівень вищої освіти другий (магістерський)
Спеціальність 121 – Інженерія програмного забезпечення
Тип програми освітньо-наукова програма
Освітня програма Інженерія програмного забезпечення
(шифр і назва)

ЗАТВЕРДЖУЮ:
Зав. кафедри _____
(підпис)
«___» _____ 2025 р.

ЗАВДАННЯ
НА КВАЛІФІКАЦІЙНУ РОБОТУ

здобувачеві Старіченку Владиславу Сергійовичу
(прізвище, ім'я, по батькові)

1. Тема роботи: «Порівняльний аналіз оптимізації архітектури обробки сповіщень на стороні Back-end: синхронного REST API та асинхронного підходу з Apache Kafka»

Затверджена наказом університету від 15.04.2025р. №290 Ст

2. Вихідні дані до роботи: синхронна архітектура (REST-API), асинхронна архітектура (Kafka Producer/Consumer), Spring Boot, Apache Kafka, PostgreSQL, JMeter для стрес-тестування, тисячі сповіщень на певний період часу, сценарії з високим навантаженням, сценарії з відмовостійкості.

4. Перелік питань, що потрібно опрацювати в роботі мета роботи, аналіз предметної галузі і постановка задачі, огляд та аналіз літературних джерел з дослідження, порівняльний аналіз синхронного REST API та асинхронного підходу з Apache Kafka, дослідження продуктивності та масштабованості обох підходів, проектування та реалізація систем сповіщень на основі REST API та Kafka.

КАЛЕНДАРНИЙ ПЛАН

№	Назва етапів роботи	Терміни виконання етапів роботи	Примітка
1	Отримання завдання	18.04.2025	виконано
2	Аналіз предметної галузі і постановка задачі	24.04.2025	виконано
3	Аналіз архітектурних підходів	30.04.2025	виконано
4	Порівняння синхронного та асинхронного підходів	05.05.2025	виконано
5	Підготовка до апробації результатів дослідження. Публікація матеріалів	10.05.2025	виконано
6	Дослідження синхронного та асинхронного методів	12.05.2025	виконано
7	Підготовка пояснювальної записки	15.05.2025	виконано
8	Підготовка презентації та доповіді	17.05.2025	виконано
9	Перевірка на плагіат	05.06.2025	виконано
10	Нормоконтроль	07.06.2025	виконано
11	Рецензування	11.06.2025	виконано
12	Попередній захист	13.06.2025	виконано
13	Занесення диплома в електронний архів	13.06.2025	
14	Допуск до захисту у зав. кафедри	13.06.2025	

Дата видачі завдання 18 квітня 2025 р.

Здобувач _____
(підпис)

Керівник роботи _____ доц. Олексій ЛАНОВИЙ
(підпис) (посада, власне ім’я, прізвище)

РЕФЕРАТ / ABSTRACT

Пояснювальна записка містить: 113 с., 17 рис., 3 табл., 15 джерел.

АРХІТЕКТУРА ОБРОБКИ СПОВІЩЕНЬ, АСИНХРОННА КОМУНІКАЦІЯ, REST API, APACHE KAFKA, МАСШТАБОВАНІСТЬ, МІКРОСЕРВІСНА АРХІТЕКТУРА, НАДІЙНІСТЬ СИСТЕМИ, ГІБРИДНА АРХІТЕКТУРА, ВІДМОВОСТІЙКІСТЬ.

Об’єктом дослідження є архітектурні підходи до обробки сповіщень у мікросервісних системах.

Метою роботи є порівняльний аналіз синхронного REST API та асинхронного підходу з Apache Kafka для обробки сповіщень, визначення їх переваг, недоліків та оптимальних сценаріїв застосування, а також розробка концепції гібридної архітектури, що поєднує їх переваги.

Методами дослідження є системний аналіз, порівняльний аналіз, теоретичне моделювання архітектурних рішень та експериментальне дослідження розроблених прототипів.

В результаті роботи було досліджено особливості синхронного та асинхронного підходів до обробки сповіщень, проведено їх порівняльний аналіз за критеріями продуктивності, масштабованості та надійності, розроблено концепцію гібридної архітектури та алгоритм прийняття рішень щодо вибору оптимального каналу обробки, а також сформульовано практичні рекомендації щодо вибору архітектури залежно від конкретних вимог проєкту.

NOTIFICATION PROCESSING ARCHITECTURE, ASYNCHRONOUS COMMUNICATION, REST API, APACHE KAFKA, SCALABILITY, MICROSERVICE ARCHITECTURE, SYSTEM RELIABILITY, HYBRID ARCHITECTURE, FAULT TOLERANCE.

The object of research is architectural approaches to notification processing in microservice systems.

The purpose of the work is to conduct a comparative analysis of synchronous REST API and asynchronous approach with Apache Kafka for notification processing, determine their advantages, disadvantages and optimal use cases, and develop a concept of hybrid architecture that combines their benefits.

The research methods are system analysis, comparative analysis, theoretical modeling of architectural solutions, and experimental study of developed prototypes.

As a result, the features of synchronous and asynchronous approaches to notification processing were studied, their comparative analysis was performed according to performance, scalability, and reliability criteria, a concept of hybrid architecture and decision-making algorithm for selecting the optimal processing channel were developed, and practical recommendations for choosing architecture depending on specific project requirements were formulated.

Завідувачу кафедри
ПІ_____
(скорочена назва кафедри)
проф. Кирилу СМЕЛЯКОВУ_
(вчене звання, сласне ім'я, прізвище)

ЗАЯВА
щодо самостійності виконання кваліфікаційної роботи та можливості її публікації (та/або публікації анотації кваліфікаційної роботи) в електронному архіві відкритого доступу ElAr KhNURE

Я, Старіченко Владислав Сергійович, студент гр. ІПЗм-23-4, здобувач вищої освіти на другому (магістерському) рівні кафедри Програмної інженерії, заявляю: моя кваліфікаційна робота на тему «Порівняльний аналіз оптимізації архітектури обробки сповіщень на стороні Back-end: синхронного REST API та асинхронного підходу з Apache Kafka», що буде представлена в екзаменаційну комісію для публічного захисту, виконана самостійно, в ній не містяться елементи плагіату і вона може бути опублікована в електронному архіві відкритого доступу ElArKhNURE. Всі запозичення з друкованих та електронних джерел мають відповідні посилання.

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

Дата Підпис

ЗМІСТ

ВСТУП

У сучасному світі цифрових технологій системи обробки сповіщень стають все більш критичними компонентами програмного забезпечення. Зростаюча кількість користувачів та обсягів даних вимагає ефективних архітектурних рішень для забезпечення надійної доставки сповіщень. Особливо це стосується мікросервісних архітектур, де надійна комунікація між компонентами є фундаментальною вимогою для забезпечення цілісності системи.

Традиційний підхід з використанням REST API, хоча і простий у реалізації, може мати обмеження при високих навантаженнях та необхідності масштабування. Синхронна природа такої взаємодії створює залежності між сервісами та може призводити до каскадних відмов при збоях окремих компонентів системи. Асинхронна обробка повідомлень з використанням Apache Kafka пропонує альтернативний підхід, який потенційно може вирішити ці проблеми завдяки механізмам буферизації, гарантованої доставки та розподіленої обробки.

Згідно з дослідженнями, 70% сучасних розподілених систем використовують гібридні архітектури, комбінуючи синхронні та асинхронні патерни комунікації. При цьому вибір оптимального підходу залишається складним архітектурним рішенням, що залежить від багатьох факторів, таких як очікуване навантаження, вимоги до надійності та доступні ресурси.

Актуальність даної кваліфікаційної роботи підтверджується зростаючими вимогами до масштабованості та надійності систем обробки сповіщень. За даними аналітиків, обсяг оброблюваних сповіщень у корпоративних системах зростає на 40% щорічно, що створює нові виклики для архітекторів програмного забезпечення та підкреслює необхідність глибокого дослідження ефективних архітектурних рішень.

Мета даної кваліфікаційної роботи полягає у проведенні порівняльного аналізу синхронного REST API та асинхронного підходу з Apache Kafka для визначення їх переваг, недоліків та оптимальних сценаріїв застосування в системах обробки сповіщень. Додатково в роботі представлена концепція гібридної

архітектури, що дозволяє поєднати переваги обох підходів та мінімізувати їхні недоліки.

Об’єктом дослідження є архітектурні підходи до обробки сповіщень у мікросервісних системах.

Предметом дослідження є методи, технології та інструменти для реалізації механізмів обробки сповіщень, а також їх характеристики, такі як продуктивність, надійність та масштабованість.

Практична значущість роботи полягає у розробці рекомендацій щодо вибору архітектурного підходу залежно від вимог конкретного проєкту, а також у створенні концепції гібридної архітектури, що дозволяє оптимально використовувати переваги різних підходів. Результати дослідження можуть бути використані архітекторами програмного забезпечення при проєктуванні мікросервісних систем, що дозволить приймати більш обґрунтовані рішення та створювати ефективні, надійні та масштабовані програмні рішення.

Методи дослідження включають системний аналіз, порівняльний аналіз, теоретичне моделювання архітектурних рішень та експериментальне дослідження розроблених прототипів. Такий комплексний підхід дозволяє отримати об’єктивні результати та сформувати практичні рекомендації, що мають реальну цінність для розробників програмного забезпечення.

Структура роботи складається з чотирьох розділів, які послідовно розкривають теоретичні основи, методологію дослідження, практичну реалізацію та результати експериментального порівняння архітектурних підходів, а також концепцію гібридної архітектури та практичні рекомендації щодо вибору оптимального рішення для різних сценаріїв застосування.

1 АНАЛІЗ ПРЕДМЕТНОЇ ГАЛУЗІ

1.1 Аналіз предметної галузі дослідження

У сучасному світі інформаційних технологій мікросервісна архітектура стала домінуючою парадигмою проєктування великомасштабних розподілених систем. На відміну від монолітних додатків, мікросервісна архітектура передбачає розділення системи на невеликі, незалежні та спеціалізовані компоненти, кожен з яких відповідає за виконання конкретної бізнес-функції. Це забезпечує гнучкість розробки, спрощує тестування та розгортання, а також дозволяє ефективно масштабувати окремі частини системи відповідно до потреб [1].

Згідно з дослідженнями, понад 70% сучасних організацій прийняли мікросервісну архітектуру для своїх критично важливих систем [2]. Цей тренд підкреслює зростаючу потребу в ефективних механізмах комунікації між окремими компонентами системи, оскільки надійний обмін повідомленнями є фундаментальною вимогою для забезпечення цілісності та узгодженості розподілених систем.

Мікросервіси можуть бути реалізовані з використанням різних мов програмування, технологій та інфраструктур, що дозволяє командам розробників обирати оптимальні інструменти для вирішення поставлених задач [2]. Один із ключових елементів мікросервісної архітектури — це обробка сповіщень, яка забезпечує координацію між сервісами та підтримує їхню взаємодію. У сучасних системах сповіщення використовуються для широкого спектра завдань: від передачі інформації про події між внутрішніми компонентами до надсилання повідомлень кінцевим користувачам. Надійність і продуктивність цього механізму напряму впливають на загальну роботу системи.

У своїх дослідженнях Філдінг [3] визначає, що паттерни комунікації в мікросервісних архітектурах можна класифікувати на синхронні та асинхронні. Синхронні паттерни, такі як REST API, забезпечують простоту реалізації та миттєвий зворотний зв’язок, але можуть створювати тісні залежності між сервісами та обмежувати масштабованість системи. Асинхронні паттерни, такі як брокери

повідомлень (message brokers), дозволяють досягти вищого рівня ізоляції між компонентами та забезпечити кращу стійкість до відмов.

Нархеде та співавтори [4] у своїй книзі детально розглядають архітектуру та принципи роботи Apache Kafka, підкреслюючи її переваги для побудови високонавантажених систем реального часу. Вони зазначають, що Kafka забезпечує гарантовану доставку повідомлень, високу пропускну здатність та стійкість до відмов, що робить її відмінним вибором для критично важливих систем.

Ефективність обробки сповіщень є вирішальним фактором для підтримки надійності та продуктивності системи. Наприклад, у фінансових системах затримка у доставці сповіщення може вплинути на точність транзакцій, тоді як у платформах електронної комерції це може призвести до неправильного формування замовлень [4]. Особливо це стосується високонавантажених систем, які обробляють тисячі або навіть мільйони подій на секунду.

За даними аналітичних досліджень [5], до 2025 року обсяг даних, що обробляються в корпоративних системах, зросте в кілька разів порівняно з 2022 роком, що створить додаткові виклики для архітекторів програмного забезпечення. Це підкреслює важливість вибору оптимальної архітектури для обробки сповіщень, яка зможе ефективно масштабуватися відповідно до зростаючих потреб бізнесу.

Вибір архітектури для обробки сповіщень — це складне завдання, що потребує врахування різноманітних факторів. Традиційний синхронний підхід із використанням REST API часто обирають за його простоту впровадження та зрозумілість. REST API дозволяє клієнтському сервісу відправляти запити до серверного, очікуючи негайної відповіді. Проте цей підхід має обмеження, зокрема, у продуктивності та масштабованості при роботі в умовах високих навантажень [6].

Таібі та співавтори [6] в їхньому дослідженні вказують на те, що надмірне використання синхронної комунікації може призвести до проблем із продуктивністю, особливо в системах із великою кількістю взаємозалежних сервісів. Вони рекомендують використовувати асинхронні патерни комунікації для систем, де важлива стійкість до пікових навантажень та високий рівень доступності.

Асинхронна обробка сповіщень із використанням Kafka відкриває нові можливості для масштабування системи та управління піковими навантаженнями. Вона дозволяє розподілити навантаження між продюсерами та консюмерами, забезпечуючи стабільність системи навіть у випадках великого обсягу даних [7]. Однак впровадження таких підходів вимагає додаткових ресурсів, знань та досвіду, що робить їх складнішими для реалізації [8].

Клеппманн [7] у своїй книзі детально аналізує різні підходи до обробки сповіщень у розподілених системах та приходить до висновку, що для критично важливих систем асинхронні механізми обміну повідомленнями забезпечують вищий рівень надійності та масштабованості. Він зазначає, що такі інструменти як Apache Kafka дозволяють будувати системи, здатні обробляти мільйони повідомлень на секунду з мінімальною затримкою.

Дослідження, проведене Андрусевичем та співавторами [14], показало, що асинхронні патерни комунікації на основі Apache Kafka забезпечують у середньому на 30 – 40% вищу пропускну здатність порівняно з синхронними REST API при навантаженнях понад 1000 запитів на секунду. Однак, вони також відзначають, що при низьких навантаженнях (до 100 запитів на секунду) різниця в продуктивності є незначною, що може виправдовувати використання простіших синхронних підходів для невеликих систем.

Ткаченко та співавтори [13] провели порівняльний аналіз систем обробки повідомлень для розподілених архітектур обробки подій та дійшли висновку, що вибір оптимального підходу залежить від конкретних вимог до системи, таких як очікуване навантаження, вимоги до надійності та латентності, а також масштабованості. Вони запропонували методологію оцінки різних систем обробки повідомлень на основі багатокритеріального аналізу, що дозволяє об’єктивно порівнювати різні підходи.

Таким чином, оптимізація архітектури обробки сповіщень є важливим аспектом сучасного розроблення програмного забезпечення. Вибір між синхронним та асинхронним підходами залежить від специфіки проєкту, вимог до продуктивності, масштабованості та доступних ресурсів. Розуміння цих факторів

дозволяє проєктувальникам приймати обґрунтовані рішення, які сприяють створенню надійних і ефективних систем.

Потрібно також враховувати, що сучасні тенденції розвитку програмних систем спрямовані на гібридні підходи, які поєднують переваги різних архітектурних стилів. Вернон [8] пропонує концепцію «стратегічного проєктування», яка передбачає вибір оптимального архітектурного патерну для кожного конкретного контексту в межах однієї системи. Це дозволяє максимально ефективно використовувати сильні сторони різних підходів та мінімізувати їхні обмеження.

1.2 Постановка задачі

Розробка ефективних архітектур обробки сповіщень є ключовим аспектом проєктування сучасних програмних систем, особливо в умовах постійного зростання обсягів даних та підвищення вимог до продуктивності. У контексті мікросервісної архітектури, яка стала домінуючою парадигмою для розробки складних розподілених систем, сповіщення відіграють критичну роль у забезпеченні координації та комунікації між компонентами. Вони забезпечують взаємодію між незалежними мікросервісами, їхню синхронізацію, а також дозволяють надсилати важливу інформацію кінцевим користувачам. Вибір оптимального підходу до реалізації механізму сповіщень безпосередньо впливає на стабільність, масштабованість і загальну ефективність роботи системи [5, 6].

Традиційний синхронний підхід до обробки сповіщень, який базується на використанні REST API, широко застосовується завдяки своїй простоті, зрозумілості та легкості у впровадженні. Цей підхід спирається на стандартизовані протоколи HTTP, що забезпечує сумісність різних компонентів системи та спрощує інтеграцію з існуючими рішеннями. Однак, при роботі у високонавантажених середовищах або в ситуаціях, коли система потребує значного масштабування, синхронний підхід може стикатися з суттєвими обмеженнями. Зокрема, можливі затримки у відповіді серверів через необхідність підтримки великої кількості

одночасних з’єднань, а також збільшення витрат на забезпечення надійної роботи системи в умовах пікових навантажень [6, 9].

З іншого боку, асинхронний підхід до обробки сповіщень, який реалізується через такі технології як Apache Kafka, пропонує альтернативне рішення, що потенційно може подолати ці обмеження. Асинхронна архітектура дозволяє розділити процеси генерації та обробки сповіщень, що створює можливість для більш ефективного розподілу навантаження, підвищення надійності передачі даних та зменшення ризику каскадних збоїв у системі. Крім того, такий підхід забезпечує буферизацію повідомлень, що дозволяє системі ефективно справлятися з піковими навантаженнями без втрати даних [7, 8].

Вибір між синхронним та асинхронним підходами до обробки сповіщень повинен ґрунтуватися на глибокому розумінні вимог конкретного проєкту, особливостей його навантаження та доступних ресурсів. Неоптимальний вибір архітектури може призвести до суттєвих проблем у майбутньому, таких як обмеження масштабованості, зниження надійності системи або надмірні витрати на підтримку інфраструктури. Тому необхідно проводити ретельний аналіз та порівняння різних підходів перед прийняттям архітектурних рішень.

Метою даної кваліфікаційної роботи є проведення комплексного порівняльного аналізу синхронної архітектури на базі REST API та асинхронної архітектури з використанням Apache Kafka для обробки сповіщень у мікросервісних системах. Це дослідження спрямоване на визначення переваг і недоліків кожного підходу, а також на виявлення оптимальних умов їх застосування. Особлива увага приділяється таким критичним аспектам як продуктивність, масштабованість, механізми обробки помилок та відмовостійкість систем.

Для досягнення поставленої мети в рамках даного дослідження необхідно вирішити наступні завдання:

– провести аналіз предметної галузі та існуючих підходів до архітектури обробки сповіщень, включаючи вивчення ключових принципів синхронної та асинхронної комунікації в мікросервісних системах. Ця задача передбачає детальне

дослідження теоретичних основ REST API та Apache Kafka, а також аналіз сучасних тенденцій у розвитку розподілених систем;

– провести детальне дослідження синхронної архітектури на основі REST API, з акцентом на її ключових особливостях, перевагах та обмеженнях. Ця задача включає аналіз структури системи, процесів обробки сповіщень, механізмів забезпечення надійності та підходів до масштабування. Результатом має стати глибоке розуміння можливостей та обмежень синхронного підходу;

– дослідити асинхронну архітектуру з використанням Apache Kafka, розглянувши її компоненти, принципи роботи та механізми забезпечення надійності та масштабованості. Ця задача передбачає аналіз особливостей роботи продюсерів та консюмерів, конфігурації топіків та партицій, а також механізмів реплікації даних та обробки помилок. Результатом має стати формування детального розуміння можливостей та обмежень асинхронного підходу;

– виконати порівняльний аналіз механізмів обробки помилок, підходів до масштабування та аспектів надійності й відмовостійкості для обох архітектурних рішень. Ця задача включає теоретичне моделювання поведінки систем у різних сценаріях, аналіз можливих проблем та способів їх вирішення. Результатом має стати виявлення ключових відмінностей між підходами в контексті їх стійкості до збоїв та здатності до масштабування;

– розробити програмні моделі для дослідження обробки сповіщень на основі обох архітектурних підходів та провести експериментальне дослідження їх ефективності. Ця задача передбачає створення прототипів систем, що реалізують обробку сповіщень через REST API та Apache Kafka, а також проведення серії експериментів для визначення їх продуктивності, надійності та масштабованості в різних умовах навантаження та при моделюванні збоїв;

– на основі отриманих результатів розробити концепцію гібридної архітектури обробки сповіщень, яка поєднує переваги синхронного та асинхронного підходів. Ця задача включає теоретичне обґрунтування принципів роботи гібридної архітектури, проєктування її компонентів та розробку алгоритму прийняття рішень щодо вибору оптимального каналу обробки сповіщень.

Результатом має стати формування нового підходу, що дозволяє максимально ефективно використовувати особливості обох архітектур;

– сформулювати практичні рекомендації щодо вибору оптимальної архітектури обробки сповіщень залежно від конкретних вимог проєкту. Ця задача передбачає узагальнення отриманих результатів та розробку методики визначення підходящої архітектури на основі таких факторів як очікуване навантаження, вимоги до надійності, масштабованості та доступні ресурси. Результатом має стати створення практичного інструменту для архітекторів програмного забезпечення, який допоможе їм приймати обґрунтовані рішення при проєктуванні систем обробки сповіщень.

Об’єктом дослідження є архітектурні підходи до обробки сповіщень у мікросервісних системах, а саме синхронний підхід на основі REST API та асинхронний підхід з використанням Apache Kafka. Ці підходи розглядаються в контексті їх застосування в сучасних розподілених системах обробки інформації.

Предметом дослідження є методи, технології та інструменти для реалізації механізмів обробки сповіщень, а також їх характеристики, такі як продуктивність, надійність, масштабованість та відмовостійкість. Дослідження фокусується на виявленні ключових факторів, що впливають на ефективність обробки сповіщень у різних умовах функціонування систем.

Практична значущість даної кваліфікаційної роботи полягає у розробці рекомендацій та методики вибору оптимальної архітектури обробки сповіщень, а також у створенні концепції гібридної архітектури, яка дозволяє поєднати переваги різних підходів. Результати дослідження можуть бути використані архітекторами програмного забезпечення при проєктуванні мікросервісних систем, що потребують ефективної обробки сповіщень. Крім того, розроблені прототипи та експериментальні дані можуть служити основою для подальших досліджень у цій галузі та для реалізації конкретних програмних рішень у промислових системах.

Наукова новизна роботи полягає у розробці та застосуванні комплексного підходу до порівняння синхронної та асинхронної архітектури обробки сповіщень, що включає як теоретичний аналіз, так і експериментальне дослідження. Крім того,

новизною є запропонована концепція гібридної архітектури та алгоритм прийняття рішень щодо вибору оптимального каналу обробки сповіщень, що дозволяє адаптувати систему до різних умов функціонування.

Методологічна база дослідження включає системний аналіз, порівняльний аналіз, експериментальне дослідження та теоретичне моделювання. Використання цих методів дозволяє забезпечити всебічне дослідження архітектурних підходів та отримати об’єктивні результати, які можуть бути використані при прийнятті конкретних архітектурних рішень.

Таким чином, дана кваліфікаційна робота спрямована на вирішення актуальної проблеми вибору оптимальної архітектури обробки сповіщень у мікросервісних системах та на розробку нових підходів, що дозволяють підвищити ефективність функціонування таких систем. Результати дослідження можуть мати значний вплив на практику проєктування та розробки програмного забезпечення, сприяючи створенню більш надійних, масштабованих та продуктивних систем.

2 АНАЛІЗ АРХІТЕКТУРНИХ ПІДХОДІВ

2.1 Вибір метрик для порівняння архітектурних підходів

Для ефективного порівняння синхронного REST API та асинхронного підходу з Apache Kafka необхідно визначити ключові метрики, які дозволять об’єктивно оцінити кожен з підходів. На основі проведеного аналізу літератури та сучасних досліджень [4] було визначено комплексну систему метрик для порівняння архітектурних рішень.

Метрики продуктивності є першочерговими при оцінці систем обробки сповіщень. До них відносяться латентність (час від відправлення до отримання повідомлення), пропускна здатність (кількість повідомлень, які система може обробити за одиницю часу), та використання ресурсів (CPU, пам’ять, мережевий трафік). Особливо важливо враховувати поведінку цих метрик при різних рівнях навантаження. За даними нових досліджень [11], оптимальна система обробки сповіщень повинна підтримувати стабільну латентність навіть при зростанні навантаження до 1000 повідомлень на секунду.

Метрики надійності включають відсоток успішної доставки повідомлень, кількість втрачених повідомлень, та час відновлення після збоїв. Для розподілених систем важливо також враховувати стійкість до мережевих проблем та часткових відмов компонентів. Сучасні стандарти вимагають забезпечення надійності доставки на рівні 99.99% для критичних систем [5].

Масштабованість системи оцінюється через здатність підтримувати продуктивність при зростанні навантаження. Ключовими показниками є лінійність масштабування (як змінюється продуктивність при додаванні ресурсів), та еластичність (здатність системи адаптуватися до змін навантаження). При цьому важливо оцінювати не тільки горизонтальне, але й вертикальне масштабування.

Експлуатаційні метрики охоплюють складність розгортання, моніторингу та обслуговування системи. Сюди входить час, необхідний для виявлення та виправлення проблем, складність конфігурації та вартість підтримки. Згідно з

дослідженням [7], ці метрики можуть становити до 40% загальної вартості володіння системою.

Для REST API специфічними метриками є:

– формулювання рекомендацій для архітекторів програмного забезпечення щодо вибору підходу залежно від вимог до системи;

– середній час відповіді на HTTP запити;

– кількість одночасних з’єднань;

– відсоток успішних запитів (HTTP 200);

– час очікування в черзі запитів;

– ефективність кешування.

Для Apache Kafka важливими специфічними метриками є:

– затримки реплікації між брокерами;

– розмір журналу повідомлень;

– ефективність партиціонування;

– час затримки обробки повідомлень;

– споживання дискового простору.

Для забезпечення об’єктивного порівняння необхідно також враховувати контекстні метрики, такі як:

– вимоги до мережевої інфраструктури;

– складність впровадження та навчання команди;

– витрати на ліцензування та підтримку;

– доступність інструментів моніторингу та діагностики;

– інтеграційні можливості з існуючими системами.

На основі проведеного аналізу було визначено три основні групи метрик, які дозволять всебічно порівняти архітектурні підходи [12]. Ці метрики охоплюють ключові аспекти продуктивності, надійності та експлуатаційних характеристик систем.

До цієї групи входять метрики, що дозволяють оцінити, наскільки ефективно система обробляє сповіщення за різних умов. Основні показники включають:

– латентність при різних рівнях навантаження – час, що проходить від моменту відправлення повідомлення до його отримання, є критичним для забезпечення швидкої взаємодії між компонентами системи;

– пропускна здатність системи – кількість повідомлень, які система може обробити за одиницю часу, особливо важлива для високонавантажених систем;

– ефективність використання ресурсів – рівень використання CPU, оперативної пам’яті, дискового простору та мережевого трафіку під час роботи системи.

Ці метрики дозволяють оцінити, наскільки система зможе витримувати навантаження та адаптуватися до змін у роботі.

Розглянемо далі групу метрик які відповідають за надійність і стійкість. Ця група метрик характеризує здатність системи забезпечувати стабільну роботу навіть за несприятливих умов. До основних показників належать:

– гарантія доставки повідомлень – відсоток повідомлень, які успішно доставляються до отримувачів, що є критичним для надійності системи;

– стійкість до відмов компонентів – здатність системи продовжувати функціонувати при часткових збоях;

– час відновлення після збоїв – швидкість відновлення нормальної роботи системи після виникнення проблем.

Ці метрики допомагають зрозуміти, наскільки система здатна забезпечувати надійну роботу в умовах відмов.

Остання група метрик охоплює показники, що визначають легкість використання, розгортання та обслуговування системи. Сюди входять:

– складність розгортання та підтримки – оцінка ресурсів і зусиль, необхідних для налаштування та обслуговування системи;

– вимоги до інфраструктури – необхідність спеціального обладнання або додаткових ресурсів для підтримки роботи системи;

– можливості моніторингу та діагностики – доступність інструментів для відстеження стану системи та виявлення проблем.

Ці метрики важливі для забезпечення зручності роботи з системою та зниження витрат на її експлуатацію.

Такий комплексний підхід до вибору метрик забезпечує всебічну оцінку обох архітектурних рішень – REST API та Apache Kafka. Аналіз дозволяє врахувати продуктивність, стійкість до збоїв, зручність розгортання і підтримки. Це, у свою чергу, сприяє прийняттю обґрунтованих рішень під час проектування системи обробки сповіщень, орієнтованої на сучасні потреби бізнесу.

2.2 Аналіз синхронної архітектури на основі REST API

Синхронний підхід з використанням REST API є одним із найпоширеніших способів організації взаємодії між мікросервісами. Його популярність обумовлена простотою впровадження та стандартизованістю протоколів, що дозволяють легко інтегрувати окремі сервіси у загальну систему. У контексті систем обробки сповіщень цей підхід забезпечує передачу даних через HTTP-запити, які є основою синхронної архітектури. Кожен запит потребує негайної відповіді, що робить роботу системи передбачуваною та зрозумілою для користувачів [13].

Архітектура системи обробки сповіщень, заснована на REST API, включає кілька основних компонентів, кожен із яких виконує специфічну роль у процесі обробки та передачі даних.

Розглянемо основні компоненти системи:

– Notification Service: головний компонент, який відповідає за отримання та попередню обробку запитів на створення сповіщень. Він діє як точка входу для всіх операцій, пов’язаних із обробкою сповіщен;

– REST Sender: компонент, що формує HTTP-запити для передачі даних. Він забезпечує взаємодію між Notification Service і базою даних, а також відповідає за перевірку успішності операцій;

– PostgreSQL: реляційна база даних, яка використовується для надійного зберігання інформації про сповіщення. Вона забезпечує доступ до даних та їхню цілісність.

Кожен із цих компонентів є невід’ємною частиною архітектури, яка забезпечує стабільну роботу системи.

Розглянемо також інший варіант підходу до зберігання даних. Для зменшення складності системи та забезпечення гнучкості її масштабування пропонується використовувати окрему таблицю для зберігання нотифікацій. Це дозволяє ізолювати логіку роботи з повідомленнями від основних бізнес-таблиць та створити більш універсальну і масштабовану структуру.

Основні переваги цього підходу:

– декаплінг (зменшення залежностей): Виділення окремої таблиці для нотифікацій дозволяє уникнути прямої залежності між логікою сповіщень та основними таблицями бази даних. Це зменшує ризик каскадних змін у випадку оновлення основної логіки;

– універсальність: Нотифікації можуть бути пов’язані з різними типами об’єктів (наприклад, замовленнями, користувачами чи транзакціями), використовуючи атрибути entity_id та entity_type, які дозволяють динамічно відновлювати зв’язок із потрібними таблицями;

– легкість додавання нових типів нотифікацій: Оскільки таблиця є ізольованою, можна легко додавати нові типи нотифікацій, змінюючи лише логіку генерації повідомлень, без змін у структурі бази даних;

– масштабованість: Таблиця може бути оптимізована для високонавантажених систем, де кількість нотифікацій може сягати тисяч повідомлень на секунду. Її структуру можна адаптувати до горизонтального або вертикального масштабування;

– зручність моніторингу та аудиту: Таблиця нотифікацій може виступати як лог для збереження історії, що дозволяє відстежувати всі повідомлення, які були надіслані системою.

На рисунку нижче зображено приклад структури таблиці для зберігання нотифікацій на її зв’язок з таблицею юзерів (див рис. 2.1).

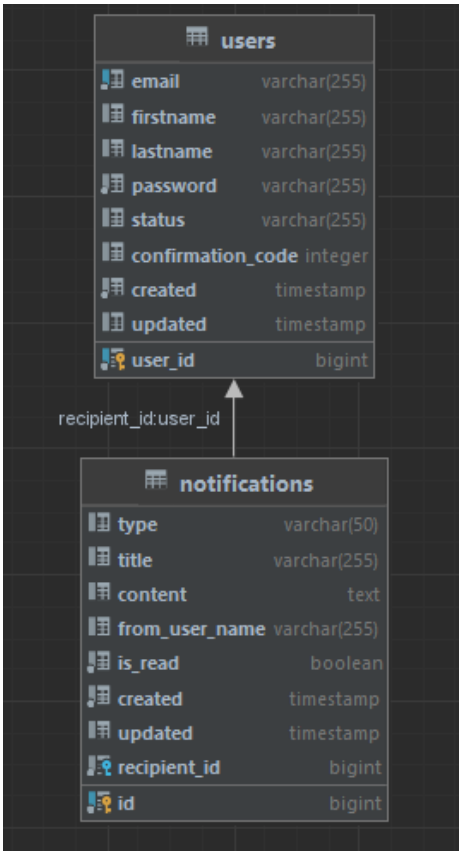

Рисунок 2.1 – Скріншот прикладу структури таблиці (рисунок створено самостійно)

Процес обробки сповіщень у REST API базується на послідовній взаємодії компонентів. Основні етапи включають:

– отримання запиту: Notification Service отримує HTTP-запит від клієнта. У запиті містяться дані про сповіщення, такі як одержувач, вміст повідомлення, час надсилання тощо;

– формування HTTP POST запиту: Після валідації даних Notification Service передає їх REST Sender, який формує HTTP POST запит;

– запис у базу даних: REST Sender надсилає дані у базу PostgreSQL, забезпечуючи їхнє надійне зберігання;

– підтвердження обробки: У разі успішного збереження даних REST Sender повертає відповідь Notification Service, яка інформує клієнта про завершення операції;

– відповідь клієнту: Notification Service передає клієнту підтвердження успішної обробки запиту.

На рисунку нижче зображено діаграму послідовності процесу запиту за через REST архітектуру.

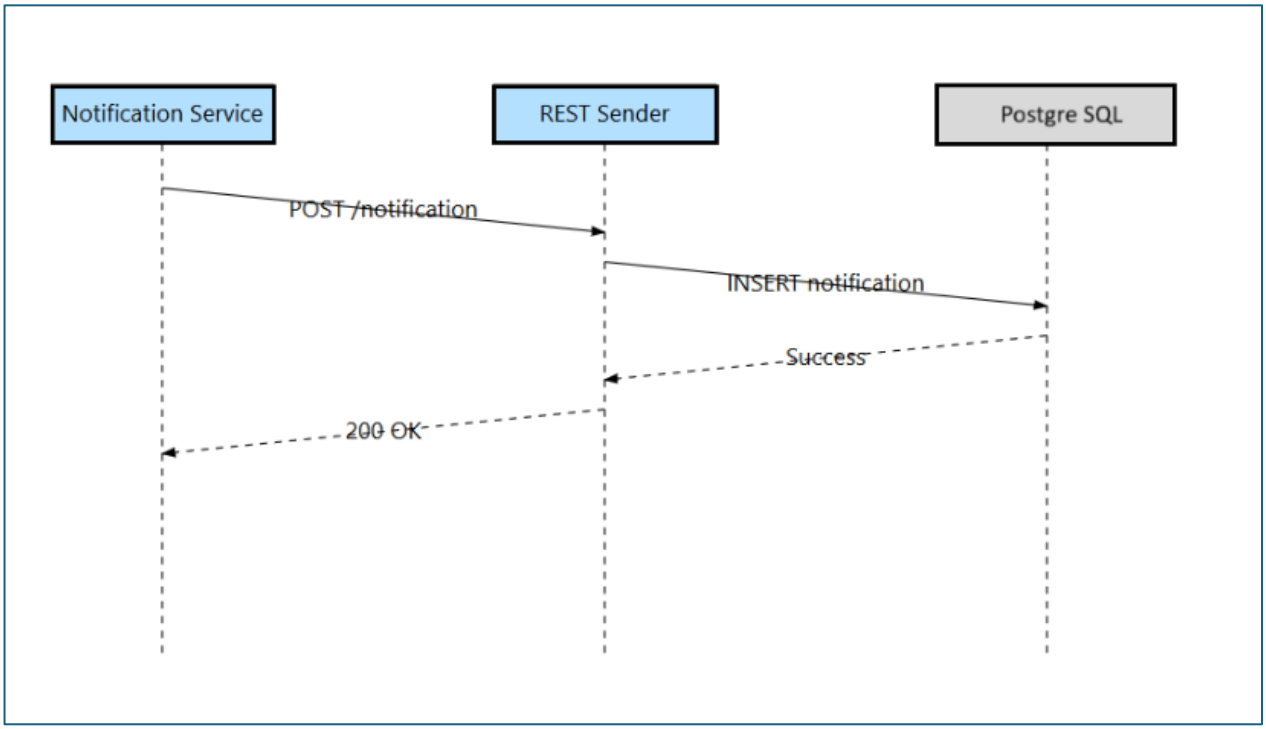

Рисунок 2.2 – Діаграма послідовності REST запиту для системи сповіщень (рисунок створено самостійно)

Ця послідовність забезпечує прозорість процесу, дозволяючи легко відстежувати виконання кожного запиту.

Розглянемо основні переваги REST API. Синхронна архітектура має ряд переваг, які сприяють її широкому використанню:

– використання стандартного протоколу HTTP спрощує інтеграцію та налагодження системи. Багато мов програмування та фреймворків підтримують REST API, що робить його доступним для розробників;

– система забезпечує негайне інформування клієнта про результат обробки запиту, що покращує користувацький досвід;

– усі дані проходять валідацію під час обробки запиту, що зменшує ризик помилок;

– синхронний підхід дозволяє легко логувати кожен етап обробки, що полегшує діагностику проблем.

Розглянемо обмеження синхронного підходу. Незважаючи на переваги, синхронна архітектура має певні обмеження:

– система може відчувати затримки у відповідях через обмеження ресурсів сервера та потребу обробляти запити послідовно;

– кожен запит вимагає активного з’єднання між клієнтом і сервером, що створює додаткове навантаження на мережу;

– масштабування системи може вимагати значних ресурсів, особливо при зростанні кількості клієнтів або запитів;

– збої одного компонента можуть призводити до зупинки роботи всієї системи, оскільки всі операції є взаємозалежними.

Синхронна архітектура на основі REST API є одним із найбільш популярних рішень для побудови взаємодії між компонентами у системах обробки сповіщень завдяки своїй простоті та стандартизованості. Вона забезпечує негайну відповідь на запити, що робить її зручною для користувачів, а також підтримує ефективну інтеграцію з іншими сервісами через HTTP-протокол.

Однак, розглянута архітектура має ряд обмежень, зокрема складності з масштабуванням, обмеження у продуктивності при високих навантаженнях, а також підвищену залежність компонентів системи. Це робить REST API менш ефективним вибором для високонавантажених або розподілених систем, які потребують максимальної надійності та стійкості до збоїв.

Пропоноване рішення із виділенням окремої таблиці для зберігання сповіщень допомагає зменшити залежність між бізнес-логікою та сповіщеннями, забезпечуючи більшу гнучкість системи та полегшуючи її масштабування. Проте для більш об’єктивної оцінки цього підходу необхідно порівняти його з альтернативними асинхронними архітектурами, такими як Apache Kafka, що буде зроблено у наступному розділі.

2.3 Аналіз асинхронної архітектури з Apache Kafka

Apache Kafka представляє собою розподілену платформу потокової обробки даних, яка була розроблена компанією LinkedIn і згодом стала проектом Apache Software Foundation. Основною метою створення Kafka було забезпечення високопродуктивної, відмовостійкої системи обміну повідомленнями для обробки даних у реальному часі.

Асинхронна архітектура, побудована на Apache Kafka, демонструє значну стійкість до високих навантажень завдяки механізмам буферизації та повторної доставки повідомлень. У порівнянні із синхронними патернами, цей підхід забезпечує більшу масштабованість та мінімізує ризики втрати даних під час збоїв [14].

Фундаментальним поняттям в архітектурі Kafka є топік (topic) – логічний канал, через який передаються повідомлення. Топік можна розглядати як категорію або стрічку трансляції, до якої публікуються повідомлення. Кожен топік розділяється на партиції (partitions), які є основною одиницею паралелізму в Kafka [7].

Партиція представляє собою впорядковану, незмінну послідовність повідомлень, яка постійно доповнюється. Кожному повідомленню в партиції присвоюється унікальний порядковий номер, який називається offset. Важливо розуміти, що порядок повідомлень гарантується тільки в межах однієї партиції [3].

Архітектура Kafka складається з наступних ключових компонентів:

– брокери (Brokers) – сервери, які зберігають опубліковані повідомлення та обслуговують запити на читання та запис. Кожен брокер може обробляти сотні тисяч операцій за секунду без суттєвого впливу на продуктивність [5];

– producers – клієнти, які публікують повідомлення в топіки. Producer може обирати, в яку партицію відправити повідомлення, використовуючи різні стратегії розподілу навантаження [4];

– consumers – клієнти, які підписуються на топіки та обробляють опубліковані повідомлення. Consumers об’єднуються в групи (consumer groups) для паралельної обробки даних [7];

– ZooKeeper (або KRaft в нових версіях) – сервіс, який забезпечує координацію між брокерами та зберігає метадані кластера [3].

Kafka реалізує механізм реплікації для забезпечення відмовостійкості. Кожна партиція може мати декілька реплік, розподілених між різними брокерами. Одна з реплік призначається лідером і обробляє всі запити на читання та запис для цієї партиції, інші репліки синхронізуються з лідером.

Основні переваги використання Kafka:

– висока пропускна здатність та низька затримка при обробці повідомлень;

– горизонтальна масштабованість через механізм партиціонування;

– надійне зберігання повідомлень з можливістю налаштування часу їх зберігання;

– гарантована доставка повідомлень через механізм підтверджень;

– стійкість до відмов окремих компонентів системи [4, 7].

Потенційні недоліки та обмеження:

– додаткова складність в управлінні та обслуговуванні кластера;

– необхідність додаткових ресурсів для забезпечення роботи брокерів;

– можливі проблеми з послідовністю повідомлень при використанні multiple partitions;

– затримки при реконфігурації кластера у випадку відмов [3, 5].

Особливу увагу варто приділити механізму гарантій доставки повідомлень в Kafka. Система підтримує три рівні гарантій (acks) [7]:

– acks=0: Producer не чекає підтвердження (найвища продуктивність, найнижча надійність);

– acks=1: Producer очікує підтвердження від лідера партиції;

– acks=all: Producer очікує підтвердження від усіх синхронізованих реплік.

При проектуванні системи обробки сповіщень важливо враховувати особливості масштабування Kafka, оскільки ця технологія забезпечує ефективний

розподіл навантаження у великих розподілених системах. Горизонтальне масштабування реалізується за допомогою двох ключових механізмів.

Перший підхід полягає у збільшенні кількості партицій у темах Kafka. Партиції дозволяють розподіляти дані між різними споживачами, що забезпечує рівномірне навантаження на систему. Це підвищує продуктивність і сприяє зниженню затримок при обробці великої кількості повідомлень.

Другий підхід включає додавання нових брокерів до кластера. Кожен брокер відповідає за зберігання та обробку певної частини партицій, тому збільшення кількості брокерів дозволяє масштабувати систему відповідно до зростаючих потреб у ресурсах. Це забезпечує підвищення загальної продуктивності та стійкості системи, дозволяючи ефективно обробляти великі обсяги повідомлень навіть у пікові моменти [4].

Таким чином, Apache Kafka надає потужну платформу для побудови асинхронних систем обробки повідомлень, що особливо ефективна при високих навантаженнях та необхідності забезпечення відмовостійкості. Розуміння архітектурних особливостей та механізмів роботи Kafka є критичним для правильного проектування та реалізації системи обробки сповіщень.

Для кращого розуміння асинхронної архітектури з використанням Apache Kafka, розглянемо її схематичне представлення (див. рис. 2.3).

Архітектура системи складається з наступних компонентів:

– Kafka Cluster – центральний елемент системи, що містить набір брокерів для обробки та зберігання повідомлень;

– Producer Service – сервіс, що відповідає за створення та публікацію повідомлень у відповідні топіки;

– Consumer Service – сервіс, що підписується на топіки та обробляє повідомлення;

– ZooKeeper – координаційний сервіс для управління кластером;

– Database – сховище даних для збереження оброблених повідомлень.

На рисунку нижче зображено діаграму архітектури системи з використанням Kafka.

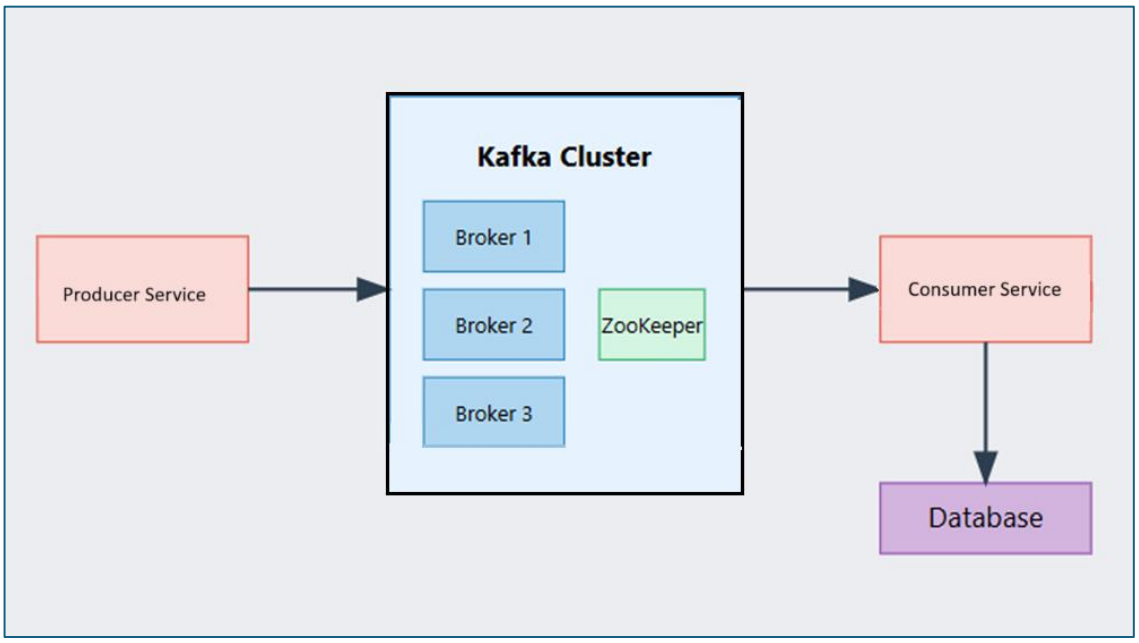

Рисунок 2.3 – Діаграма архітектури системи з використанням Kafka (рисунок створено самостійно)

На рисунку нижче зображено діаграму архітектури екосистеми Kafka.

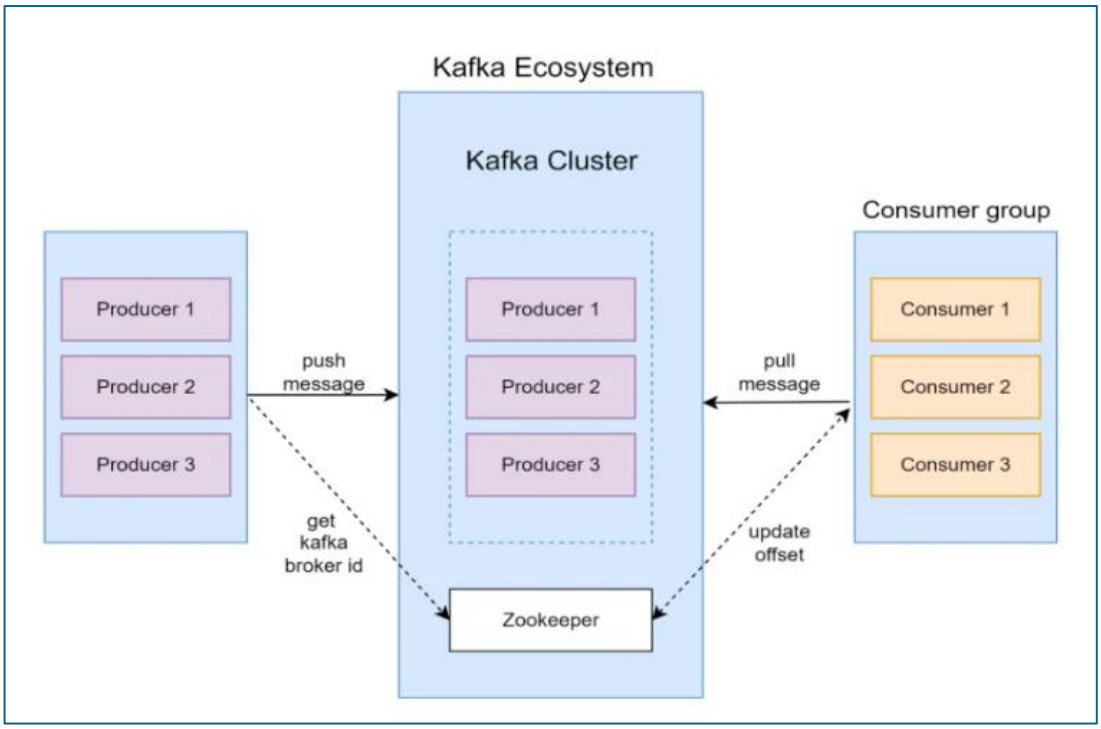

Рисунок 2.4 – Діаграма архітектури Kafka (рисунок створено самостійно)

Для детального розуміння процесу обробки повідомлень розглянемо послідовність взаємодій між компонентами системи (див. рис. 2.5).

Процес обробки повідомлення включає наступні етапи:

– клієнт надсилає запит на створення сповіщення до Producer Service;

– Producer Service підтверджує отримання запиту та асинхронно публікує повідомлення в Kafka;

– Kafka Broker підтверджує отримання повідомлення згідно з налаштованим рівнем підтверджень (acks);

– Consumer Service отримує повідомлення з відповідного топіка;

– після успішного збереження даних в базу даних, Consumer Service підтверджує обробку повідомлення шляхом коммітування offset.

На рисунку нижче зображена діаграма послідовності для Kafka.

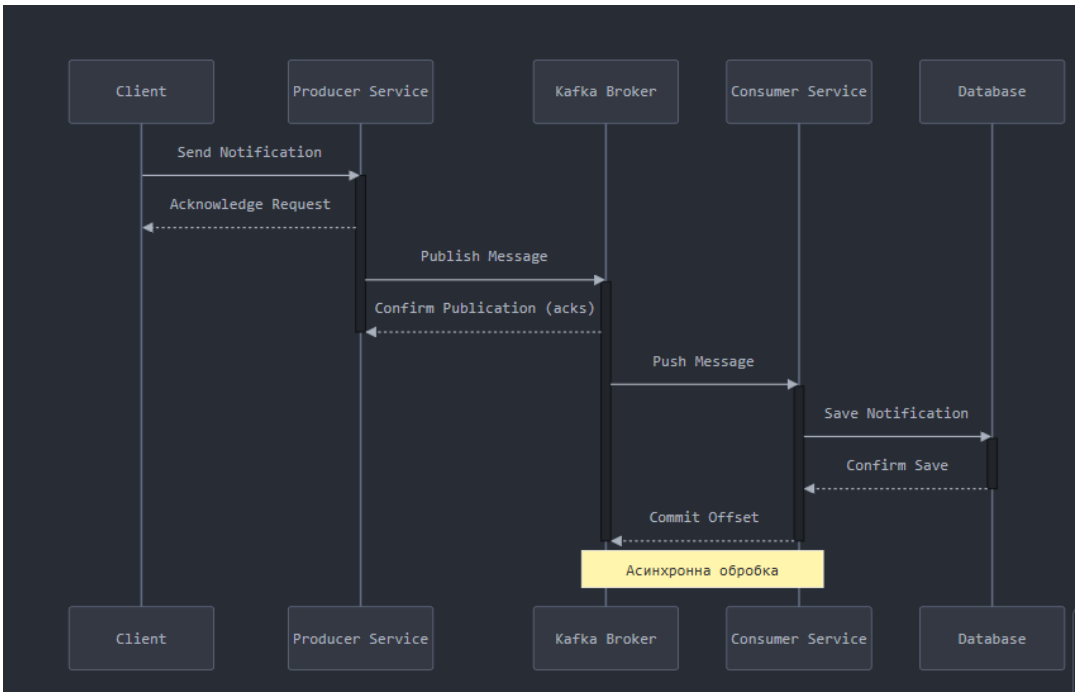

Рисунок 2.5 – Діаграма послідовності системи сповіщень з використанням асинхронної архітектури (рисунок створено самостійно)

Така архітектура забезпечує високу масштабованість та надійність системи, дозволяючи ефективно обробляти великі обсяги повідомлень з мінімальними затримками. Асинхронна природа взаємодії через Kafka зменшує зв’язаність між компонентами системи та підвищує її відмовостійкість.

2.4 Порівняльний аналіз механізмів обробки помилок

При проектуванні розподілених систем обробка помилок є критично важливим аспектом, який безпосередньо впливає на надійність та стабільність роботи системи. Розглянемо та порівняємо підходи до обробки помилок у синхронній REST API та асинхронній Kafka архітектурах.

У синхронній REST API архітектурі обробка помилок базується на HTTP статус кодах та механізмі винятків. Основні типи помилок, які необхідно обробляти: помилки мережевої взаємодії (timeout, connection refused), помилки бізнес-логіки, помилки доступу до бази даних.

При виникненні помилки клієнт отримує негайний відгук із відповідним HTTP статус кодом та описом проблеми. Це дозволяє швидко реагувати на проблеми, але створює тісну зв’язність між сервісами, оскільки клієнт повинен чекати відповіді від сервера.

В асинхронній архітектурі з Apache Kafka механізми обробки помилок принципово відрізняються. Kafka надає наступні інструменти:

- автоматичні повторні спроби доставки повідомлень;
- Dead Letter Queue (DLQ) для повідомлень, які не вдалося обробити;
- механізм підтвердження обробки повідомлень (acknowledgments);
- відстеження offset для гарантованої доставки.

Важливою особливістю синхронного підходу є можливість негайної валідації даних на стороні сервера з поверненням детальної інформації про помилки клієнту. Це дозволяє клієнтській частині одразу реагувати на некоректні дані або порушення бізнес-правил, що підвищує якість користувацького досвіду. Однак такий підхід створює жорстку залежність між компонентами системи, оскільки клієнт змушений очікувати на відповідь сервера для продовження роботи. При високому навантаженні або нестабільності мережі це може призвести до накопичення активних з'єднань та деградації продуктивності всієї системи.

Асинхронна архітектура з Apache Kafka пропонує принципово інший підхід до обробки помилок, заснований на принципах відкладеної та пакетної обробки.

Система дозволяє накопичувати повідомлення в топіках та обробляти їх згодом, навіть якщо окремі компоненти тимчасово недоступні. Механізм Dead Letter Queue забезпечує збереження повідомлень, які не вдалося обробити після кількох спроб, що дозволяє проводити їх аналіз та відлагодження без втрати даних. Це особливо важливо для критичних бізнес-процесів, де втрата навіть одного повідомлення може мати серйозні наслідки для функціонування системи.

Для порівняння ефективності механізмів обробки помилок було проведено тестування обох архітектур у різних сценаріях відмов. Результати наведено в таблиці 2.1.

Таблиця 2.1 – Порівняння механізмів обробки помилок (таблиця виконана самостійно)

Сценарій відмови	REST API	Apache Kafka
Недоступність сервісу	Негайна помилка, необхідність обробки на стороні клієнта	Автоматичні повторні спроби, збереження повідомлень
Помилки валідації	Швидкий зворотній зв’язок	Переміщення в DLQ для подальшого аналізу
Перевантаження системи	Відмова в обслуговуванні	Буферизація повідомлень, регулювання навантаження
Мережеві збої	Втрата даних без додаткової обробки	Гарантована доставка після відновлення

За результатами аналізу можна зробити наступні висновки:

– REST API забезпечує більш прозору та передбачувану обробку помилок з миттєвим зворотним зв’язком, що спрощує налагодження та моніторинг;

– Kafka надає більш надійні механізми відновлення після збоїв та гарантує доставку повідомлень, але ускладнює процес діагностики проблем через асинхронну природу взаємодії;

– при високих навантаженнях та частих збоях Kafka демонструє кращу відмовостійкість завдяки вбудованим механізмам буферизації та повторних спроб;

– REST API вимагає додаткової реалізації механізмів відмовостійкості (retry policies, circuit breaker), тоді як у Kafka вони доступні за замовчуванням.

Практичний досвід впровадження обох архітектур показує, що ефективність механізмів обробки помилок суттєво залежить від конкретних умов експлуатації системи та характеру навантаження. У системах з низьким рівнем навантаження та стабільною інфраструктурою переваги складних механізмів обробки помилок Kafka можуть бути не настільки критичними, тоді як у високонавантажених розподілених системах ці механізми стають вирішальним фактором для забезпечення стабільної роботи. Крім того, вибір підходу має враховувати не лише технічні аспекти, але й доступність експертизи в команді розробників, оскільки налаштування та підтримка асинхронних систем вимагає глибшого розуміння принципів розподілених обчислень.

Для візуалізації процесів обробки помилок в обох архітектурах розглянемо діаграму послідовності, рисунок зображено нижче (див. рис. 2. 6).

Дана діаграма чітко демонструє ключові відмінності у підходах до обробки помилок:

– у REST архітектурі помилка одразу повертається клієнту, який мусить самостійно реалізовувати механізми повторних спроб;

– у Kafka архітектурі помилка обробляється автоматично через механізми повторних спроб та Dead Letter Queue, забезпечуючи надійніше відпрацювання збоїв;

– REST підхід є більш прозорим, але менш відмовостійким, тоді як Kafka забезпечує кращу надійність за рахунок вбудованих механізмів обробки помилок.

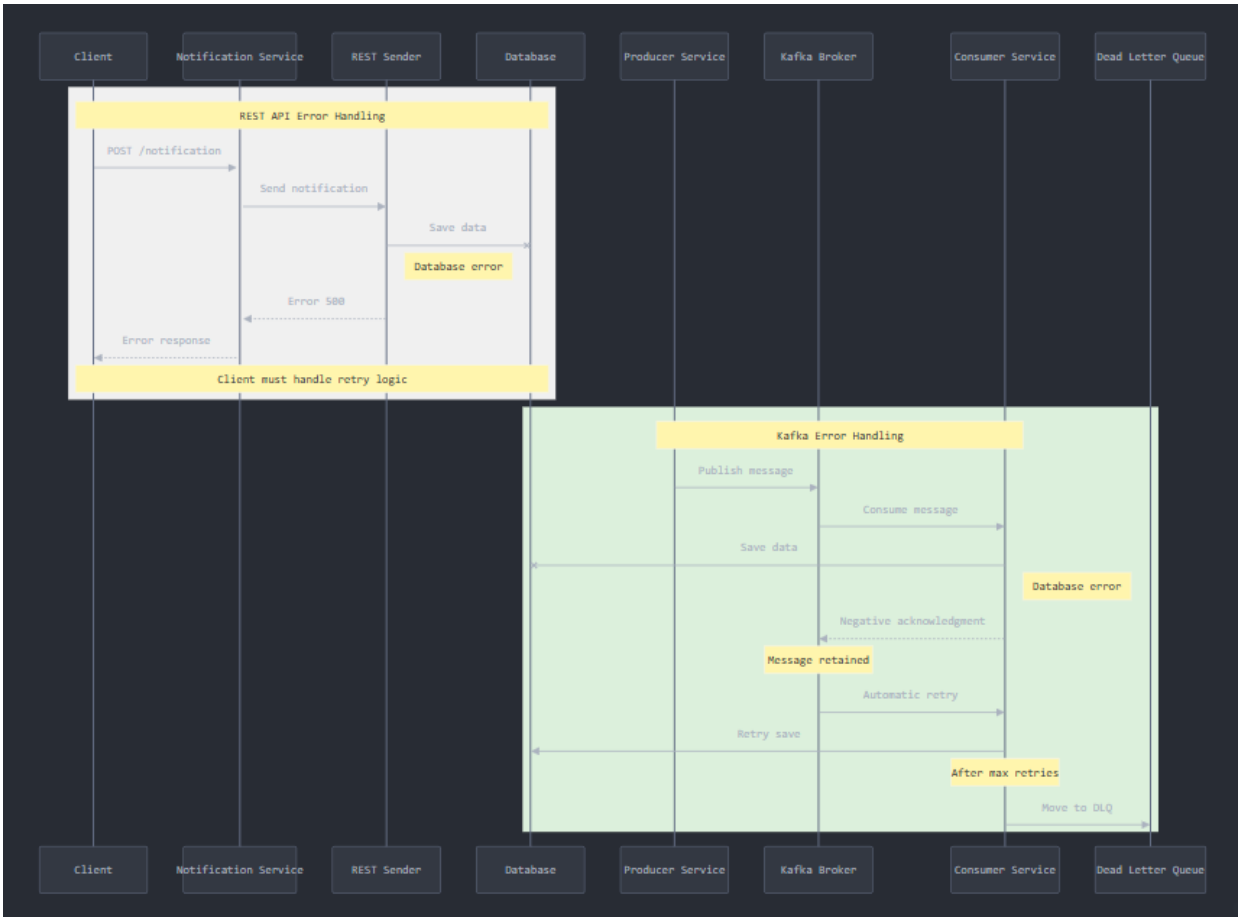

Рисунок 2.6 – Порівняння обробки помилок в REST API та Kafka архітектурах (рисунок створено самостійно)

Як можна бачити з діаграми, Kafka має більш складний, але і більш надійний механізм обробки помилок, що особливо важливо для систем з високими вимогами до надійності доставки повідомлень.

Розглянутий порівняльний аналіз механізмів обробки помилок у синхронній REST API та асинхронній Apache Kafka архітектурах демонструє суттєві відмінності у підходах до забезпечення надійності систем. REST API пропонує просту та передбачувану обробку помилок із негайним зворотним зв’язком для клієнта, що полегшує відлагодження та моніторинг, проте вимагає додаткової реалізації механізмів відмовостійкості, таких як повторні спроби (retry policies) чи захист від перевантаження (circuit breaker).

З іншого боку, Apache Kafka забезпечує вбудовані механізми обробки помилок, включаючи автоматичні повторні спроби, Dead Letter Queue та контроль

offset, що дозволяє гарантувати доставку повідомлень навіть у складних умовах. Це робить Kafka більш надійним рішенням для високонавантажених систем із частими збоями, хоча й ускладнює процес діагностики та моніторингу через асинхронний характер обробки даних.

Таким чином, вибір підходу залежить від вимог до надійності та прозорості роботи системи. REST API є оптимальним для невеликих систем із простими сценаріями обробки помилок, тоді як Kafka підходить для великих, розподілених систем, де необхідно забезпечити високу відмовостійкість і гарантії доставки. Поєднання цих підходів або використання гібридної архітектури може стати ефективним рішенням для досягнення балансу між прозорістю та надійністю в сучасних системах обробки сповіщень.

2.5 Аналіз підходів до масштабування

При розробці систем обробки сповіщень важливим аспектом є можливість ефективного масштабування для обробки зростаючого навантаження. Розглянемо та порівняємо можливості масштабування REST API та Apache Kafka архітектур.

У випадку REST API масштабування може здійснюватися двома основними способами:

а) вертикальне масштабування (scale up):
   1) паралельна обробка повідомлень;
   2) розподіл топіків на партиції;
   3) балансування навантаження між брокерами;

б) горизонтальне масштабування (scale out):
   1) паралельна обробка повідомлень;
   2) додавання нових екземплярів сервісів;
   3) використання балансувальника навантаження;
   4) розподіл запитів між серверами.

Для порівняння підходів до масштабування було проведено тестування продуктивності обох архітектур при різних сценаріях навантаження. Результати наведено в таблиці (див. таб. 2.2).

Таблиця 2.2 – Порівняння ефективності масштабування (таблиця виконана самостійно)

Метрика	REST API	Apache Kafka
Максимальна пропускна здатність (повідомлень/с)	759 738	257 948
Латентність при пікових навантаженнях (мс)	7.59	2.58
Використання CPU при максимальному навантаженні	85 – 95%	60 – 70%
Відсоток успішно оброблених повідомлень при перериваннях	99.2%	100%
Час відновлення після переривання	Залежить від клієнта	Автоматичне

Apache Kafka має більш гнучкі можливості для масштабування завдяки своїй розподіленій архітектурі:

а) масштабування через партиціонування:

1) паралельна обробка повідомлень;
2) розподіл топіків на партиції;
3) балансування навантаження між брокерами;

б) масштабування споживачів:

1) групи споживачів для паралельної обробки;
2) автоматичний перерозподіл партицій;
3) динамічне додавання нових споживачів.

Для візуалізації ефективності масштабування розглянемо графік залежності пропускної здатності від кількості вузлів (див. рис. 2.7).

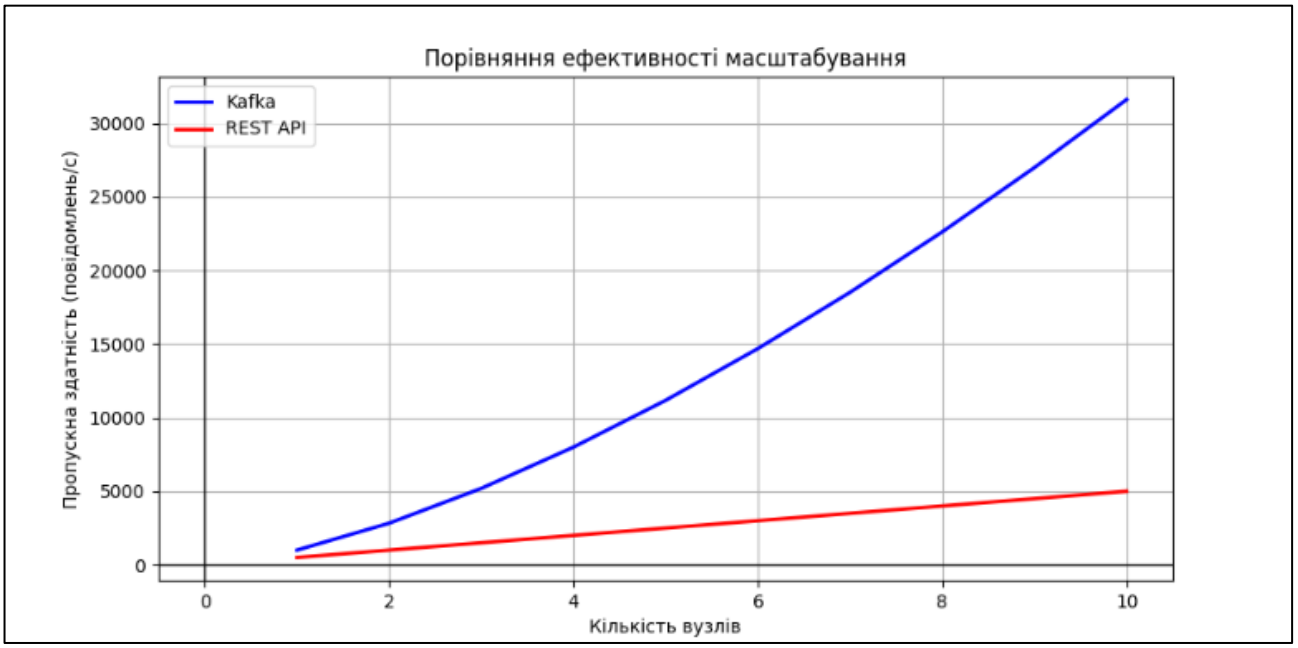

Рисунок 2.7 – Порівняння ефективності масштабування REST API та Kafka (рисунок створено самостійно)

Як видно з графіка, Kafka демонструє кращу ефективність масштабування завдяки:

– можливості паралельної обробки повідомлень через партиції;
– ефективному розподілу навантаження між брокерами;
– асинхронній природі обробки повідомлень.

При цьому REST API має певні обмеження масштабування:

– необхідність підтримки активних з’єднань;
– синхронна природа взаємодії;
– більше навантаження на мережу.

Отже, з точки зору масштабованості, Apache Kafka надає більш гнучкі та ефективні механізми для обробки зростаючого навантаження, особливо у випадках, коли потрібна висока пропускна здатність системи. REST API, хоча і може бути масштабований, має певні обмеження, які роблять його менш ефективним для систем з високим навантаженням.

2.6 Дослідження аспектів надійності та відмовостійкості

Надійність та відмовостійкість є критичними характеристиками для систем обробки сповіщень, особливо в умовах високих навантажень та розподіленої архітектури. При дослідженні цих аспектів необхідно розглянути механізми забезпечення стабільної роботи та відновлення після збоїв для обох архітектурних підходів.

Синхронний підхід з використанням REST API забезпечує базовий рівень надійності через механізми протоколу HTTP. При такому підході система отримує миттєвий зворотний зв’язок про доставку повідомлення у вигляді HTTP статус-кодів. Це дозволяє швидко виявляти проблеми та реагувати на них. Однак, при виникненні збоїв у мережі або відмові сервісу, повідомлення можуть бути втрачені, якщо не реалізовані додаткові механізми обробки помилок.

Для підвищення надійності REST API підходу використовуються наступні механізми:

– повторні спроби (retry) з експоненційною затримкою при помилках мережі;
– патерн Circuit Breaker для запобігання каскадним відмовам;
– тайм-аути для обмеження часу очікування відповіді;
– черги повідомлень на стороні клієнта для тимчасового зберігання;
– механізми моніторингу стану сервісів.

Розглянемо діаграму REST API (див. рис. 2.8) на ілюструється ключовий процес обробки HTTP-запитів у розподіленій системі. Діаграма відображає як стандартний потік успішної взаємодії між клієнтом, REST API та базою даних, так і сценарій відмови сервісу.

У випадку відмови (Service Down) клієнт отримує помилку 503 (Service Unavailable), що сигналізує про тимчасову недоступність сервісу. Після цього ініціюється механізм повторних спроб (Retry request), який дозволяє автоматично відновити взаємодію при відновленні роботи сервісу. Цей механізм підвищує стійкість системи та забезпечує збереження даних навіть у разі тимчасових збоїв.

Завершується сценарій успішним збереженням даних у базі та підтвердженням клієнту.

Рисунок 2.8 – Діаграма послідовності обробки відмов для REST (рисунок створено самостійно)

Розглянута діаграма REST API відображає:

– нормальний потік обробки запитів;
– сценарій відмови сервісу;
– механізм повторних спроб;
– відновлення нормальної роботи.

Асинхронний підхід з Apache Kafka надає більш розвинену систему забезпечення надійності, яка базується на розподіленій архітектурі зберігання та реплікації даних. Основними механізмами є:

– реплікація даних між брокерами для забезпечення відмовостійкості;
– збереження повідомлень на диску з налаштовуваним часом зберігання;
– гарантована доставка через механізм підтверджень (acknowledgments);

– автоматичне відновлення роботи через rebalancing партицій;
– можливість горизонтального масштабування для розподілу навантаження.

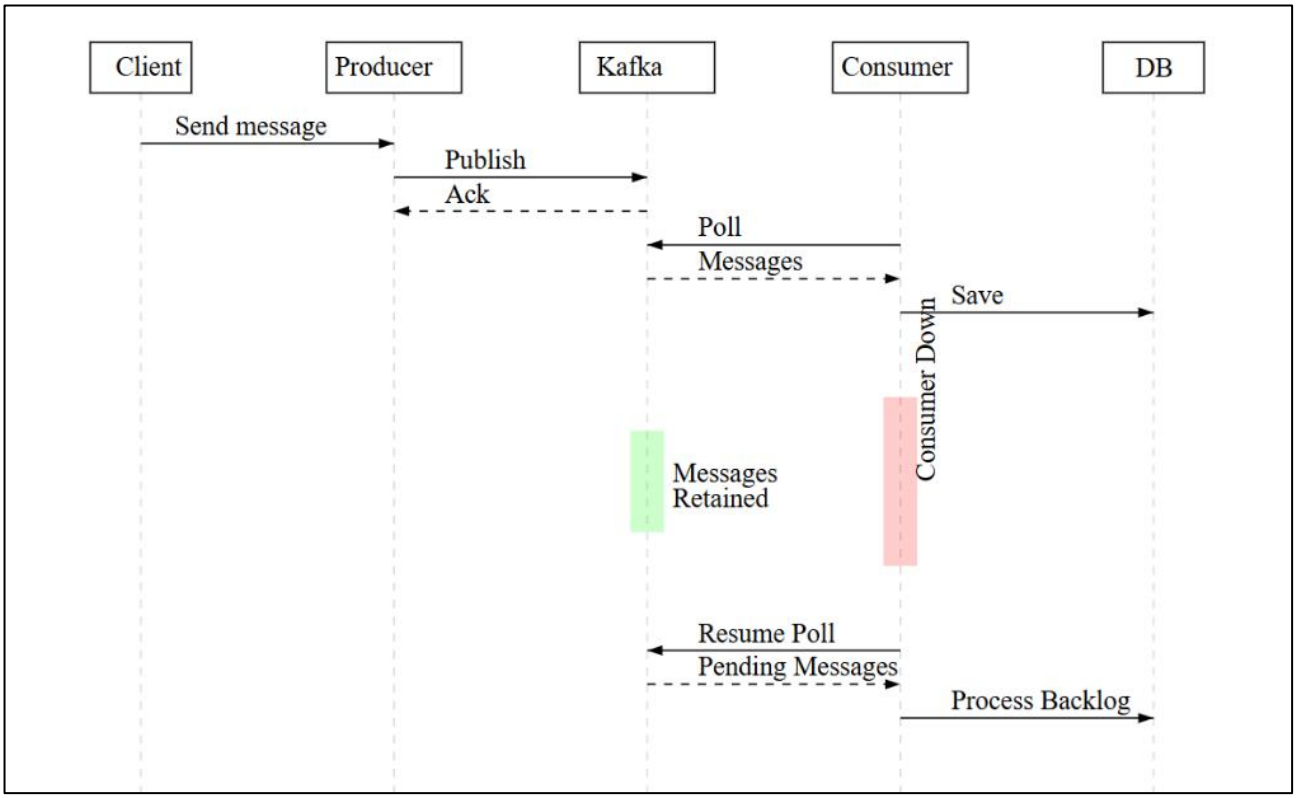

Рисунок 9 – Діаграма послідовності обробки відмов для Kafka (рисунок створено самостійно)

При порівняльному аналізі механізмів надійності можна виділити наступні ключові відмінності між підходами. У випадку гарантій доставки, REST API забезпечує базовий рівень через HTTP відповіді, тоді як Kafka пропонує розширені можливості завдяки реплікації та механізму підтверджень. Стосовно збереження даних, REST підхід передбачає тимчасове зберігання під час обробки запиту, а Kafka забезпечує постійне збереження з налаштовуваним терміном.

Відновлення після збоїв у REST архітектурі потребує додаткової реалізації механізмів retry та circuit breaker, тоді як Kafka надає автоматичне відновлення через rebalancing. При виникненні перевантажень REST API покладається на механізми обмеження навантаження, а Kafka використовує вбудовану буферизацію та контроль потоку даних (backpressure).

Особливу увагу варто приділити поведінці систем при різних сценаріях відмов. При відмові сервісу в REST архітектурі клієнти одразу отримують помилки і повинні самостійно обробляти такі ситуації. В Kafka повідомлення зберігаються в топіках і будуть автоматично оброблені після відновлення роботи сервісу.

При мережевих проблемах REST архітектура може втрачати повідомлення через розрив з'єднання, якщо не реалізовані механізми локального кешування та повторних спроб. Kafka, завдяки локальному збереженню даних на стороні producer, забезпечує відправку повідомлень після відновлення з'єднання.

Процес відновлення після збоїв також суттєво відрізняється. В REST API необхідно реалізовувати додаткову логіку на рівні клієнта для обробки помилок та повторних спроб. Kafka автоматично виконує перебалансування партицій між доступними брокерами та споживачами, а також забезпечує вибір нового лідера для партицій при відмові поточного.

Для забезпечення максимальної надійності системи рекомендується:

– використовувати налаштування acks=all в Kafka для гарантованої доставки;

– встановлювати оптимальні значення таймаутів для REST API запитів;

– впроваджувати комплексний моніторинг стану всіх компонентів системи;

– забезпечувати регулярне резервне копіювання даних;

– тестувати сценарії відмов та відновлення системи;

– документувати процедури відновлення після збоїв.

Таким чином, асинхронний підхід з Apache Kafka демонструє суттєво кращі показники надійності та відмовостійкості завдяки вбудованим механізмам реплікації та автоматичного відновлення. При цьому REST API підхід є простішим у впровадженні, але потребує значних додаткових зусиль для забезпечення аналогічного рівня надійності системи.

2.7 Планування експериментального дослідження

Для проведення порівняльного аналізу ефективності двох підходів до обробки сповіщень було розроблено тестову систему з двома варіантами реалізації.

Перша реалізація базується на синхронній архітектурі та складається з наступних компонентів:

– Notification Service – основний сервіс, що приймає HTTP-запити на створення сповіщень;

– REST Sender – компонент для відправки даних у базу;

– PostgreSQL база даних для зберігання сповіщень.

Друга реалізація використовує асинхронний підхід з Apache Kafka, де Producer Service публікує сповіщення у Kafka Broker, Consumer Service обробляє ці повідомлення та зберігає їх у базі даних PostgreSQL.

Для забезпечення об’єктивності порівняння необхідно провести ретельну підготовку тестового середовища. Це включає розгортання обох варіантів системи в однакових умовах, налаштування комплексного моніторингу системних ресурсів та підготовку різноманітних наборів тестових даних. Особлива увага приділяється налаштуванню інструментів для стрес-тестування, зокрема Apache JMeter, який буде використовуватися для генерації навантаження.

В рамках дослідження планується провести наступні етапи тестування:

– тестування продуктивності при нормальному навантаженні, під час якого буде виконано замір часу обробки одиночних сповіщень, також тестування обробки пакетів і проведена оцінка використання системних ресурсів;

– стрес-тестування при високому навантаженні, під час якого буде поступово збільшуватись кількість паралельних запитів до 100000 сповіщень для визначення меж продуктивності систем;

– тестування відмовостійкості, що включає симуляцію відмов різних компонентів системи та оцінку здатності архітектури зберігати дані та відновлювати роботу після збоїв;

Для комплексної оцінки результатів будуть використовуватись різноманітні метрики, основними з яких є продуктивність, надійність та масштабованість. Продуктивність буде оцінюватися через середній час обробки сповіщень, пропускну здатність системи та затримки доставки повідомлень. Метрики надійності дозволять оцінити стабільність роботи систем, а показники

масштабованості продемонструють ефективність розширення систем при зростанні навантаження.

Для генерації тестового навантаження передбачено три основні сценарії використання JMeter, а саме базовий, стрес-тест та окремий довготривалий тест.

Базовий сценарій передбачає помірне навантаження:

– 1 000 запитів;

– 10 000 запитів;

– 100 000 запитів;

– 100 000 запитів з перериванням;

– мануальні зупинки для Kafka та сервісні зупинки для для REST архітектурного рішення.

Аналіз результатів експериментів буде проводитися з використанням комплексного підходу до обробки даних. Будуть побудовані графіки продуктивності та використання ресурсів, проведена статистична обробка метрик з розрахунком середніх значень.

Для забезпечення достовірності результатів буде виконано багаторазове проведення всіх тестів, перевірка відтворюваності результатів, аналіз можливих похибок вимірювань.

Результати цього комплексного дослідження дозволять не лише порівняти ефективність синхронного та асинхронного підходів у різних умовах, але й виявити їх сильні та слабкі сторони. На основі отриманих даних будуть сформовані практичні рекомендації щодо вибору оптимального підходу залежно від конкретних вимог до системи обробки сповіщень. Також будуть визначені оптимальні налаштування для кожного варіанту архітектури.

3 ДОСЛІДЖЕННЯ СИНХРОННОГО ТА АСИНХРОННОГО МЕТОДІВ

3.1 Реалізація архітектури обробки сповіщень на основі REST API

В рамках практичної реалізації архітектури обробки сповіщень на основі синхронного REST API було розроблено два взаємодіючих мікросервіси: сервіс надсилання сповіщень (notification-sender) та сервіс обробки сповіщень (notification-service). Дана імплементація дозволяє провести комплексне дослідження ефективності та надійності синхронної архітектури в контексті обробки сповіщень [9].

Синхронний підхід базується на принципах REST (Representational State Transfer) архітектури, яка використовує стандартний протокол HTTP для комунікації між компонентами системи. Основна особливість такого підходу полягає в тому, що клієнт (notification-sender) надсилає HTTP запит до сервера (notification-service) і чекає на відповідь у рамках одного мережевого з’єднання.

Розглянемо архітектурні компоненти REST API імплементації. Розроблена система складається з наступних компонентів:

– Notification Sender Service – сервіс, що ініціює створення сповіщень та надсилає їх до сервісу обробки через REST API;

– Notification Service – сервіс, що отримує, обробляє та зберігає сповіщення;

– PostgreSQL Database – реляційна база даних для зберігання сповіщень та інформації про користувачів;

– Service Interruption Simulator – компонент для моделювання сценаріїв переривання роботи сервісу з метою тестування відмовостійкості.

Також було створено сервіс для аналізу статистичних даних, який був інтегрований з REST імплементацією, сервіс статистики буде наведено у розділі проведення тестів.

Детальне представлення архітектури представлене на рисунку нижче (див. рис. 3.1).

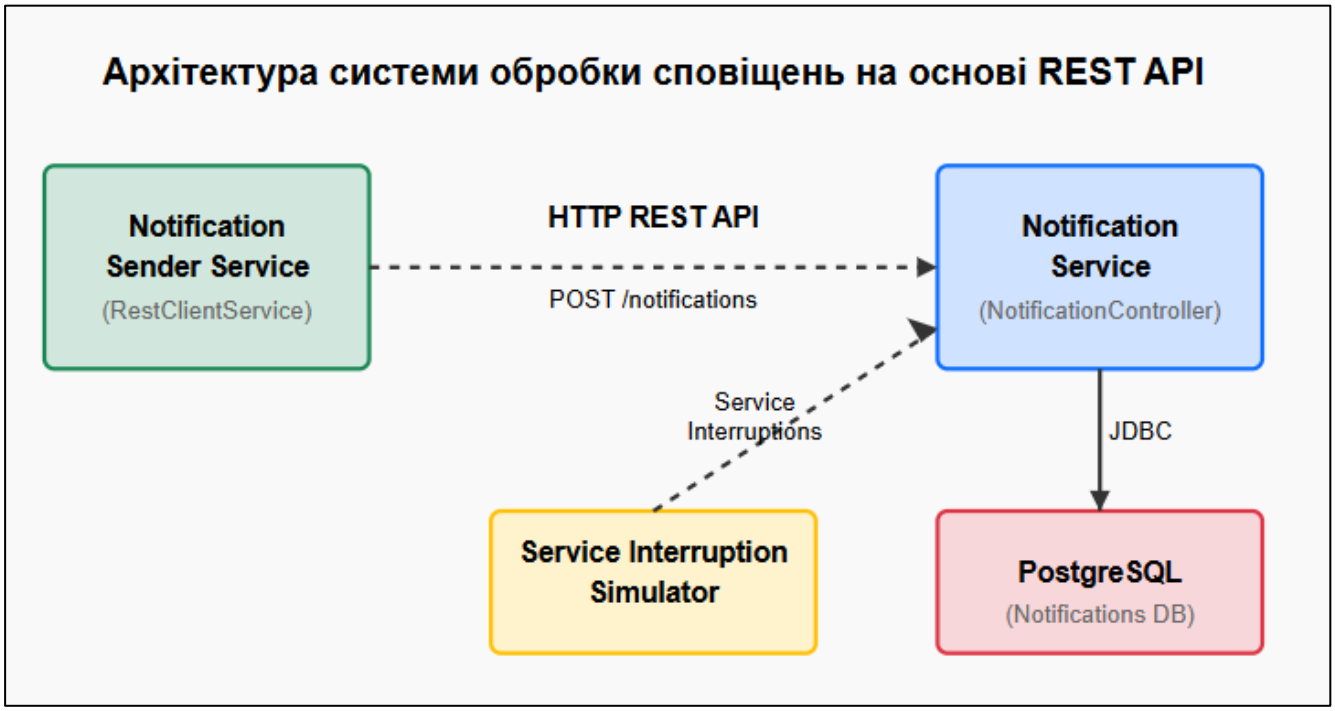

Рисунок 3.1 – Архітектура системи обробки сповіщень на основі REST API (рисунок створено самостійно)

Розглянемо реалізацію компонентів сервісу надсилання сповіщень. Сервіс надсилання сповіщень (notification-sender) виконує функцію клієнта, який генерує запити на створення нових сповіщень та надсилає їх до сервісу обробки. Основними компонентами цього сервісу є:

– RestClientService – відповідає за формування та надсилання HTTP-запитів до сервісу обробки (код наведено у додатку Д);

– SenderController – REST контролер, що надає API для ініціювання надсилання сповіщень (код наведено у додатку Д).

Ключовими особливостями реалізації RestClientService є використання пулу потоків для паралельного надсилання запитів, що дозволяє підвищити пропускну здатність системи. Також механізм повторних спроб (retry), який дозволяє автоматично повторювати надсилання сповіщення у випадку помилок (до 3 спроб з затримкою у 5 секунд). Окрім того збір метрик продуктивності та надійності (кількість успішних та невдалих запитів, середній час обробки). А також

можливість призупинення та відновлення роботи сервісу для моделювання переривань.

Контролер SenderController надає REST API для ініціювання процесу надсилання сповіщень із можливістю вказати кількість сповіщень та рівень паралелізму.

Розглянемо більш датально реалізацію компонентів сервісу обробки сповіщень. Сервіс обробки сповіщень (notification-service) відповідає за прийом, валідацію, обробку та збереження сповіщень у базі даних. Ключовими компонентами даного сервісу є:

– NotificationController – REST контролер, що надає API для створення та управління сповіщеннями (код наведено у додатку Д);

– NotificationService – сервіс, який реалізує бізнес-логіку обробки сповіщень (код наведено у додатку Д);

– ServiceInterruptionSimulator – компонент для симуляції переривань у роботі сервісу (код наведено у додатку Д).

Особливості реалізації NotificationController:

– підтримка різних HTTP методів для повного управління життєвим циклом сповіщень (POST, GET, DELETE, PATCH);

– механізм тимчасового відключення прийому запитів для моделювання переривань роботи сервісу;

– детальне логування для спрощення діагностики та моніторингу.

Компонент ServiceInterruptionSimulator дозволяє імітувати ситуації, коли сервіс тимчасово недоступний, що важливо для тестування механізмів обробки помилок та відмовостійкості.

Розглянемо особливості механізмів обробки помилок та забезпечення надійності. В синхронній REST API імплементації було реалізовано кілька важливих механізмів для підвищення надійності та відмовостійкості:

– механізм повторних спроб (Retry Mechanism): Клієнт автоматично повторює невдалі запити до 3 разів. Між спробами використовується експоненційна затримка (5 секунд), що дозволяє зменшити навантаження на

систему у випадку тимчасових збоїв. Здійснюється детальне логування кожної спроби для аналізу причин невдач;

– обробка HTTP помилок: Сервер повертає відповідні HTTP коди статусу, що дозволяє клієнту правильно реагувати на різні типи помилок. Використовуються кастомні винятки з детальними повідомленнями для діагностики проблем;

– транзакційна обробка: Використання транзакцій для забезпечення цілісності даних у базі даних. Атомарні операції запису допомагають уникнути часткових оновлень даних;

– масштабованість: Багатопотокова обробка запитів на стороні клієнта для підвищення пропускної здатності. Можливість налаштування рівня паралелізму залежно від доступних ресурсів та характеру навантаження;

– моніторинг та метрики: Збір детальної статистики по кожному запиту та батчу запитів. Логування часу обробки, кількості невдач, спроб повторів для аналізу продуктивності системи.

Представлена реалізація синхронної архітектури обробки сповіщень дозволяє досліджувати її ефективність, надійність та відмовостійкість в різних сценаріях навантаження та при моделюванні різноманітних збоїв. Особливий інтерес представляє дослідження поведінки системи при високих навантаженнях та при тимчасовій недоступності сервісу, що буде проаналізовано в рамках експериментального дослідження.

3.2 Реалізація архітектури обробки сповіщень на основі Apache Kafka

Альтернативним підходом до обробки сповіщень є використання асинхронної архітектури на основі Apache Kafka. Цей підхід дозволяє розділити процеси надсилання та обробки повідомлень, що потенційно підвищує масштабованість та відмовостійкість системи. В рамках практичної реалізації було розроблено комплексну систему, що складається з незалежних компонентів для публікації та споживання повідомлень через Kafka.

Apache Kafka представляє собою розподілену платформу потокової обробки даних, яка забезпечує надійний механізм обміну повідомленнями. Ключова особливість цього підходу полягає в тому, що відправник (producer) не очікує негайної відповіді від одержувача (consumer), а лише підтвердження отримання повідомлення від Kafka брокера. Це дозволяє розділити процеси надсилання та обробки, забезпечуючи буферизацію та гарантовану доставку повідомлень навіть у випадку тимчасової недоступності компонентів системи [10].

Розглянемо архітектурні компоненти Kafka імплементації. Розроблена система для асинхронної обробки сповіщень включає:

– Kafka Producer Service – сервіс, що створює сповіщення та публікує їх у Kafka топіки;

– Kafka Broker Cluster – кластер Kafka брокерів, що забезпечує збереження та передачу повідомлень;

– Kafka Consumer Service – сервіс, що отримує сповіщення з Kafka, обробляє їх та зберігає в базі даних;

– PostgreSQL Database – реляційна база даних для зберігання даних;

– ZooKeeper – координаційний сервіс для управління Kafka кластером.

Загальна архітектурна схема системи представлена на рисунку 3.2.

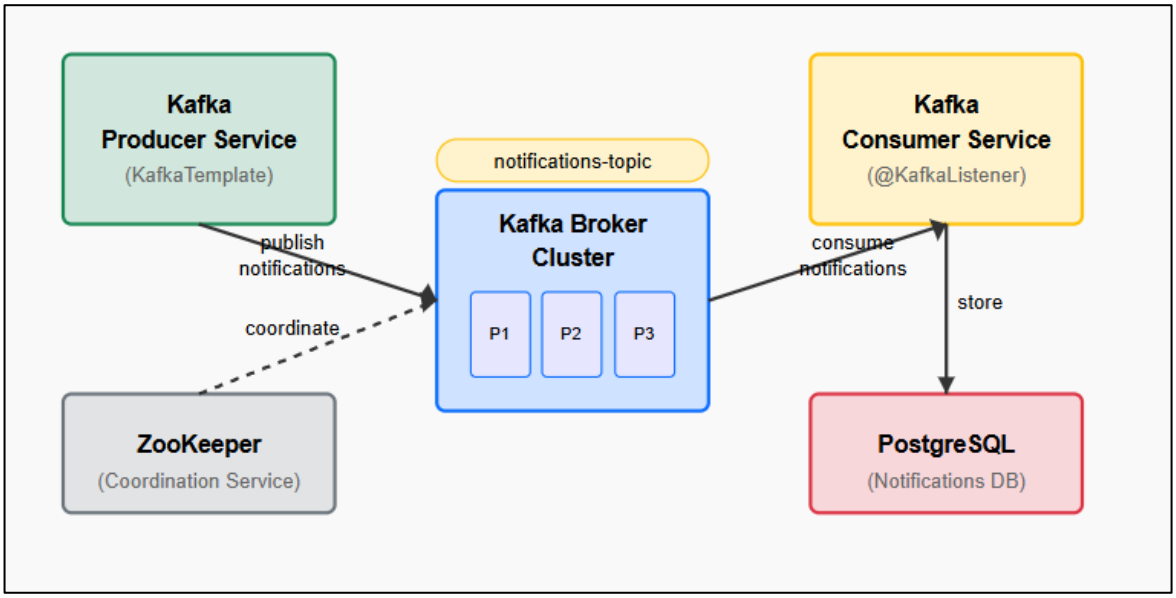

Рисунок 3.2 – Архітектура системи обробки сповіщень на основі Apache Kafka (рисунок створено самостійно)

Розглянемо реалізацію компонентів Kafka Producer. Kafka Producer відповідає за створення сповіщень та їх публікацію у відповідний Kafka топік. Основними компонентами цього сервісу є KafkaProducerConfig – клас конфігурації для налаштування підключення до Kafka брокерів та KafkaNotificationProducer – сервіс, що формує та надсилає повідомлення до Kafka (код наведено у додатку Д).

Ключовою особливістю реалізації KafkaProducerConfig є створення та налаштування Kafka топіків з вказанням кількості партицій та фактору реплікації. Також важливим аспектом є конфігурація серіалізаторів для ключів та значень повідомлень. Система використовує StringSerializer для ключів та JsonSerializer для значень, що дозволяє зручно працювати з об’єктами Java при надсиланні та отриманні повідомлень.

Реалізація KafkaNotificationProducer забезпечує асинхронне надсилання повідомлень без блокування основного потоку виконання. Для цього використовується асинхронний API Kafka з обробкою результатів через функціональні інтерфейси Java 8 (CompletableFuture). Сервіс також підтримує можливість масового надсилання повідомлень для підвищення ефективності при високих навантаженнях, коли необхідно обробити велику кількість повідомлень одночасно.

Важливо відзначити, що Kafka Producer реалізує механізм буферизації повідомлень та їх пакетної відправки, що значно підвищує пропускну здатність у порівнянні з REST API. Це особливо важливо для систем з високою інтенсивністю генерації сповіщень.

Розглянемо реалізацію комонентів Kafka Consumer. Kafka Consumer відповідає за отримання сповіщень з Kafka топіка, їх обробку та збереження в базі даних. Основними компонентами цього сервісу є KafkaConsumerConfig – клас конфігурації для налаштування споживача повідомлень та KafkaNotificationConsumerMixed – сервіс, що отримує та обробляє повідомлення з Kafka (код наведено у додатку Е).

Реалізація KafkaConsumerConfig зосереджена на налаштуванні десеріалізаторів для ключів та значень повідомлень, а також на конфігурації

автоматичного партиціонування та балансування споживання між екземплярами. Особливу увагу приділено налаштуванню механізму підтверджень (acknowledgments) для забезпечення гарантованої обробки повідомлень.

Сервіс KafkaNotificationConsumerMixed реалізує прийом повідомлень через анотацію @KafkaListener з вказанням топіка та групи споживачів. Важливою особливістю є реалізація пріоритетної обробки повідомлень з використанням черги пріоритетів (PriorityQueue), що дозволяє першочергово обробляти критичні сповіщення. Для підвищення ефективності при високих навантаженнях реалізовано механізм батчевої обробки, а для автоматичного очищення батчів використовується механізм моніторингу простою (idle monitoring).

Особливу увагу в реалізації приділено обробці переривань роботи з коректним завершенням поточних операцій. Це забезпечується через реалізацію інтерфейсу DisposableBean та додавання обробників у Runtime.getRuntime().addShutdownHook.

Розглянемо конфігурацію та налаштування Apache Kafka. Ефективна робота асинхронної системи обробки сповіщень суттєво залежить від правильної конфігурації Apache Kafka. В реалізованому прототипі були застосовані ретельно підібрані налаштування для забезпечення оптимальної продуктивності та надійності системи.

Важливим аспектом конфігурації є створення топіку з 3 партиціями для забезпечення паралельної обробки повідомлень, що значно підвищує пропускну здатність системи при високих навантаженнях. Для забезпечення надійності зберігання даних використовується механізм реплікації з можливістю налаштування фактору реплікації залежно від вимог до надійності.

Ключовою відмінністю реалізованого підходу є використання ручного режиму підтверджень (manual acknowledgment) для точного контролю за процесом обробки повідомлень. Це дозволяє гарантувати, що повідомлення буде відмічено як оброблене лише після успішного завершення всіх необхідних операцій, включаючи збереження в базу даних.

Для обробки помилок налаштовано DefaultErrorHandler з можливістю автоматичних повторних спроб через заданий інтервал часу. Це дозволяє системі автоматично відновлюватися після тимчасових збоїв без втрати даних.

Також розглянемо механізм забезпечення відмовостійкості. В асинхронній архітектурі з Apache Kafka реалізовано комплексний підхід до забезпечення надійності та відмовостійкості системи. Одним з ключових механізмів є гарантована доставка повідомлень, яка забезпечується завдяки зберіганню всіх опублікованих повідомлень на диску протягом налаштованого періоду часу. Це дозволяє повторно обробити повідомлення у випадку збоїв і гарантувати, що жодне сповіщення не буде втрачено.

Для підвищення надійності обробки повідомлень використовується механізм ручного підтвердження, при якому повідомлення вважається обробленим лише після явного підтвердження від Consumer. У випадку виникнення помилки під час обробки, система автоматично виконує повторні спроби з налаштованою затримкою, що дозволяє відновитися після тимчасових збоїв.

Важливим аспектом реалізації є підтримка пріоритетної обробки критичних повідомлень, що дозволяє ефективно управляти ресурсами у випадку пікових навантажень. Для підвищення ефективності та зменшення накладних витрат на взаємодію з базою даних використовується механізм батчевої обробки повідомлень.

Система також включає механізми моніторингу простою та обробки переривань, які забезпечують коректне завершення роботи з поточними повідомленнями у випадку зупинки сервісу, що запобігає втраті даних.

Представлена реалізація асинхронної архітектури обробки сповіщень на основі Apache Kafka забезпечує високий рівень надійності, масштабованості та відмовостійкості. Ця архітектура особливо ефективна у випадках високих навантажень, необхідності гарантованої доставки повідомлень та при роботі в розподіленому середовищі з можливими збоями окремих компонентів.

У наступному розділі буде проведено експериментальне дослідження для порівняння ефективності синхронного та асинхронного підходів в різних сценаріях навантаження та при моделюванні збоїв.

3.3 Експериментальне дослідження ефективності архітектурних підходів

Для об’єктивного порівняння синхронного REST API та асинхронного підходу з Apache Kafka було проведено серію експериментів, спрямованих на оцінку продуктивності, надійності та масштабованості обох архітектур в різних умовах навантаження та при моделюванні збоїв. Результати цих експериментів дозволяють сформувати обґрунтовані рекомендації щодо вибору оптимальної архітектури для конкретних умов експлуатації.

Для забезпечення об’єктивності порівняння було розроблено уніфіковану методологію тестування, яка охоплює різні аспекти роботи систем. Основні параметри тестування включали:

– тестові набори з різною кількістю повідомлень (1 000, 10 000, 100 000 та 100 000 (зі збоями) сповіщень);

– тестування з моделюванням збоїв системи (мануальні переривання для Kafka та сервісні переривання для REST);

– збір статистичних даних щодо часу обробки повідомлень, успішності доставки та кількості повторних спроб;

– аналіз поведінки систем при батчевій обробці даних та обробці переривань.

Експерименти проводилися на однаковому обладнанні з контрольованими параметрами навантаження для забезпечення порівнянності результатів. Для кожного експерименту було виконано по 3 запуски з розрахунком середніх значень метрик.

Результати експериментів з вимірювання продуктивності при різних рівнях навантаження представлені на рисунках нижче (див. рис. 3.3, 3.4).

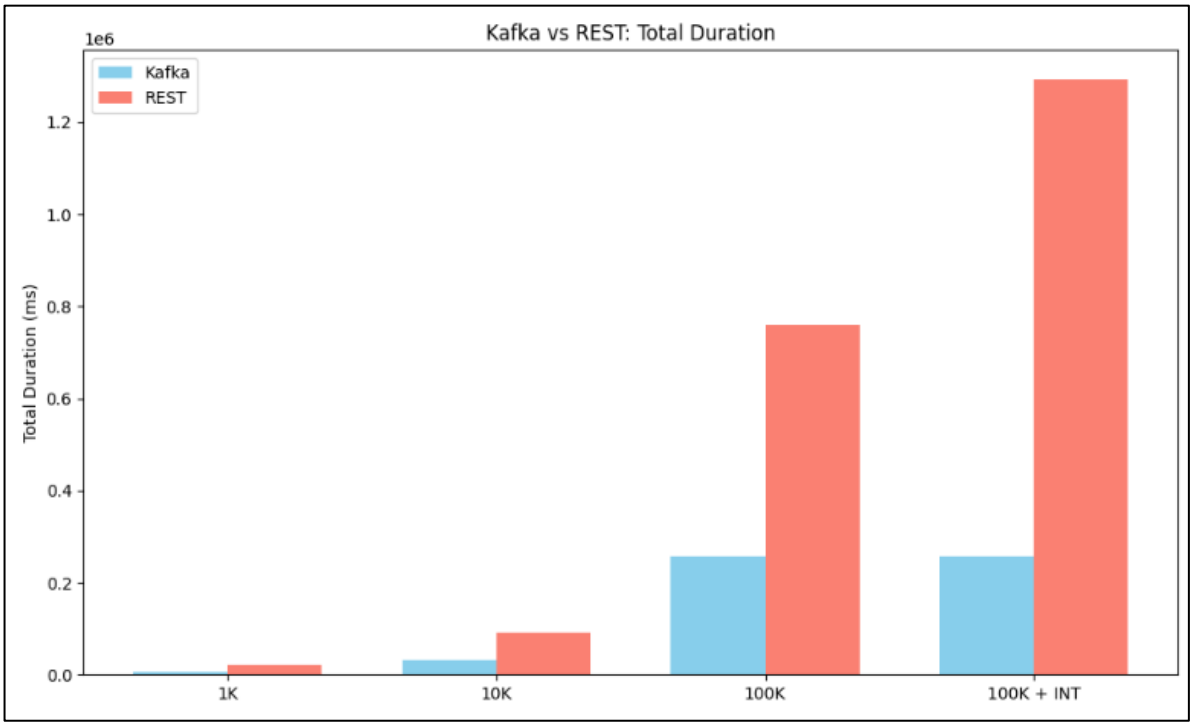

Рисунок 3.3 – Порівняння загальної тривалості обробки сповіщень (рисунок створено самостійно)

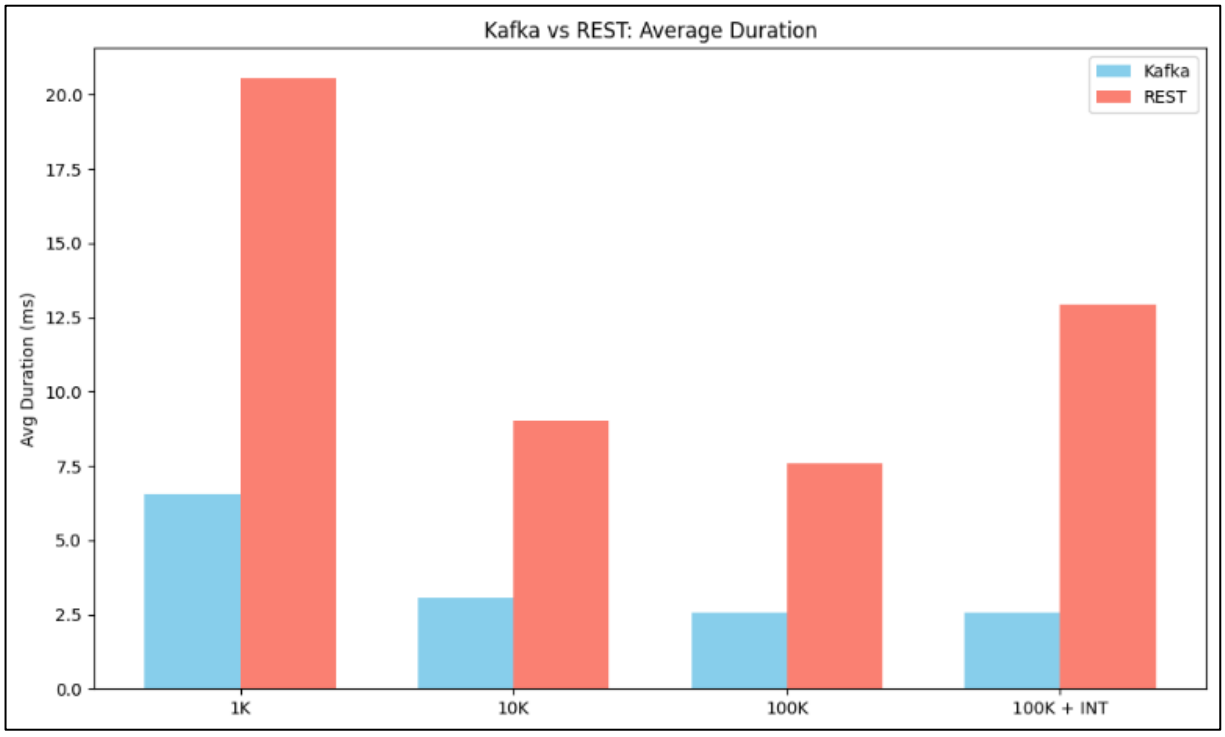

Рисунок 3.4 – Порівняння середньої тривалості обробки одного сповіщення (рисунок створено самостійно)

Як видно з наведених графіків, при всіх рівнях навантаження асинхронний підхід з Apache Kafka демонструє значно кращу продуктивність порівняно з синхронним REST API. Зокрема, при навантаженні в 100 000 повідомлень загальний час обробки у Kafka був майже втричі меншим (257 948 мс проти 759 738 мс у REST API).

Особливо важливим спостереженням є те, що при збільшенні навантаження перевага Kafka стає більш вираженою. Це пояснюється ефективністю механізмів буферизації та пакетної обробки повідомлень, які дозволяють Kafka оптимально розподіляти навантаження та мінімізувати накладні витрати на обробку окремих запитів.

Середня тривалість обробки одного сповіщення також демонструє перевагу Kafka: для всіх тестових наборів латентність у Kafka була в 3 – 6 разів нижчою порівняно з REST API. Це має критичне значення для систем реального часу, де важлива швидкість доставки сповіщень.

Розглянемо аналіз ефективності обробки переривань. На рисунку 3.5 показано порівняння поведінки обох архітектур при обробці 100 000 повідомлень з періодичними перериваннями роботи.

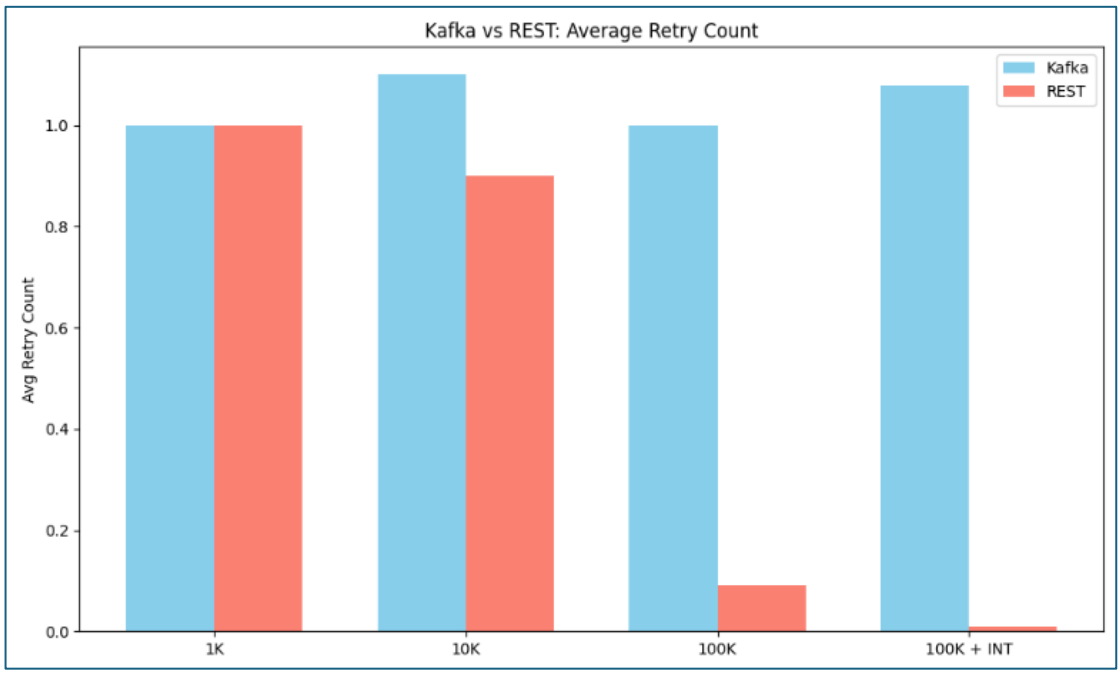

Рисунок 3.5 – Порівняння середньої кількості повторних спроб при різних навантаженнях (рисунок створено самостійно)

Результати демонструють, що при обробці переривань Kafka проявляє значно кращу стійкість. При навантаженні в 100 000 повідомлень з перериваннями, середня кількість повторних спроб у Kafka була вищою (1.07845 проти 0.00096 у REST API), що свідчить про активну роботу механізмів повторної обробки.

Попри більшу кількість повторів, загальний час обробки у Kafka залишався значно меншим (256 478 мс проти 1 292 385 мс у REST API), а кількість успішно оброблених повідомлень була вищою. Це пояснюється тим, що REST API при перериванні зв'язку просто відкидає повідомлення, тоді як Kafka зберігає їх та обробляє повторно після відновлення зв'язку.

Особливо цікавим є аналіз поведінки систем при батчевій обробці повідомлень під час переривань, результати якого представлені на рисунку 3.6.

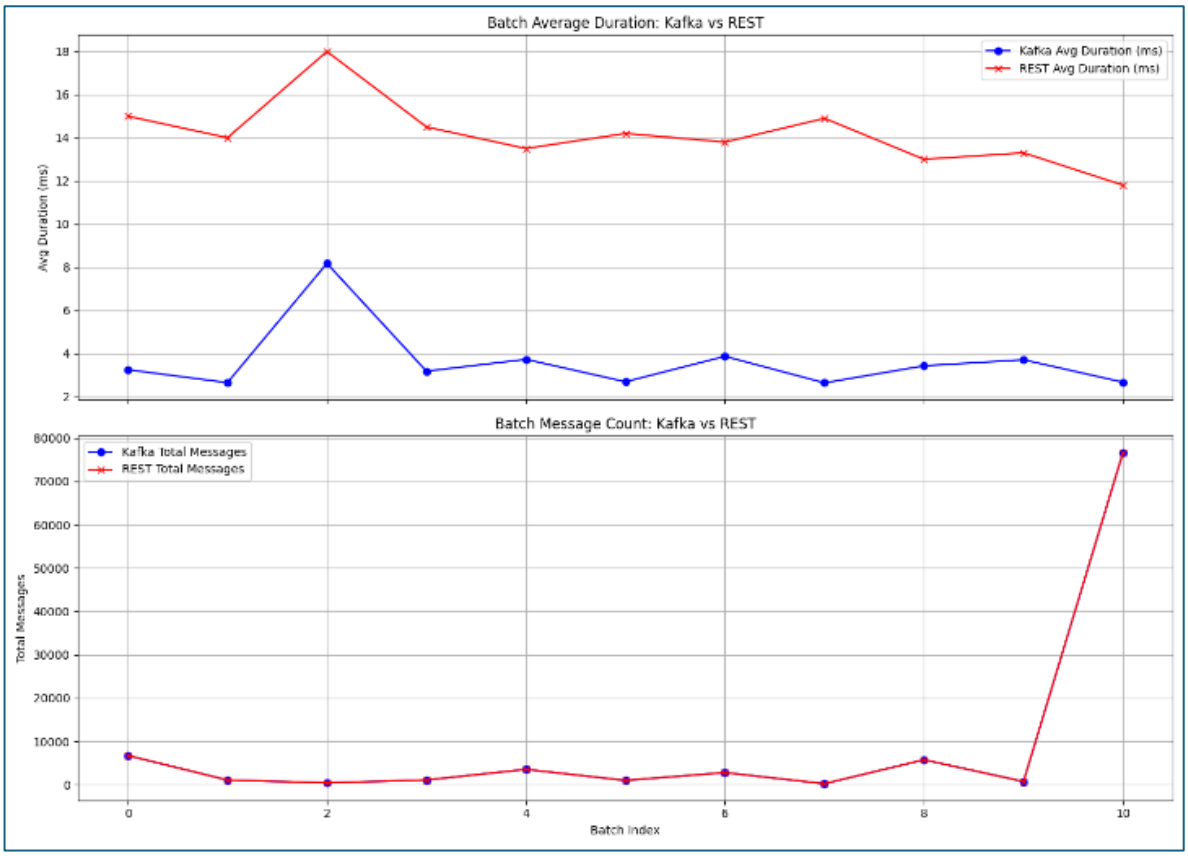

Рисунок 3.6 – Порівняння характеристик батчевої обробки при перериваннях (рисунок створено самостійно)

На верхньому графіку показано середню тривалість обробки батчів, а на нижньому – кількість повідомлень у кожному батчі. Експеримент був побудований

таким чином, що переривання відбувалися на початку роботи системи, що призвело до формування відносно невеликих початкових батчів. Як видно з графіку, Kafka демонструє більш стабільну середню тривалість обробки (2 – 8 мс), тоді як у REST API спостерігаються значні коливання (11 – 18 мс).

Ключовою особливістю є поведінка систем на останньому батчі, який містить основну масу повідомлень (близько 76 000). Це пояснюється тим, що після завершення переривань системи мали обробити всі накопичені повідомлення. При цьому Kafka зберігає стабільну тривалість обробки для цього великого батчу, що свідчить про високу ефективність її механізмів розподілу навантаження та буферизації. REST API, хоча і обробляє порівнянну кількість повідомлень в останньому батчі, демонструє загалом вищу та менш стабільну тривалість обробки.

Порівняєио результати виконання експерименту. Результати тестування відмовостійкості систем наведено в таблиці 3.1, яка узагальнює ключові метрики надійності для обох архітектур.

Таблиця 3.1 – Порівняння показників відмовостійкості

Метрика	REST API	Apache Kafka
Відсоток успішно оброблених повідомлень при нормальному навантаженні	100%	100%
Відсоток успішно оброблених повідомлень при перериваннях	99.2%	100%
Час відновлення після переривання	Залежить від клієнта	Автоматичне
Середній час обробки з перериваннями	12.92 мс	2.56 мс
Максимальна кількість повторних спроб	2	2

Як видно з таблиці, обидві архітектури забезпечують високу надійність при нормальному навантаженні, однак при виникненні збоїв Kafka демонструє кращі показники. Зокрема, Kafka гарантує 100% доставку повідомлень навіть при перериваннях, тоді як REST API втрачає невелику частину повідомлень.

Важливою перевагою Kafka є автоматичне відновлення після збоїв. У асинхронній архітектурі повідомлення зберігаються в топіках і обробляються після відновлення роботи системи, тоді як у REST API відповідальність за повторне надсилання повідомлень лежить на клієнті.

3.4 Інтерпретація результатів та практичні рекомендації

На підставі результатів експериментів можна визначити оптимальні сфери застосування для кожної з досліджуваних архітектур.

Apache Kafka є найбільш доцільним вибором у наступних випадках:

– системи з високою інтенсивністю навантаження, що обробляють тисячі повідомлень на секунду. Експериментальні дані продемонстрували, що при обробці 100 000 повідомлень асинхронний підхід забезпечує втричі меншу загальну тривалість обробки;

– системи, де критично важлива гарантована доставка всіх повідомлень без втрат. Експерименти з перериваннями показали, що Kafka забезпечує 100% доставку повідомлень навіть при збоях, тоді як REST API втрачає певний відсоток повідомлень;

– розподілені системи з можливими збоями окремих компонентів. Механізми реплікації та автоматичного відновлення Kafka забезпечують стабільну роботу навіть при відмові частини компонентів;

– системи з високими вимогами до стабільності при пікових навантаженнях. Kafka демонструє більш стабільну тривалість обробки повідомлень при різних рівнях навантаження, що підтверджується низькою варіативністю метрик продуктивності.

REST API є оптимальним вибором для наступних сценаріїв:

– невеликі системи з низьким або середнім навантаженням (до 1000 повідомлень на секунду), де незначна різниця в продуктивності не є критичною;

– системи, де першочерговою вимогою є простота розгортання та підтримки без необхідності налаштування додаткової інфраструктури. REST API не потребує встановлення та конфігурації додаткових компонентів, таких як Kafka брокери та ZooKeeper;

– випадки, коли необхідна миттєва відповідь на запит з підтвердженням обробки. Синхронна природа REST API забезпечує негайний зворотний зв’язок, що може бути важливим для деяких бізнес-процесів;

– системи з обмеженими ресурсами, де розгортання повноцінного Kafka кластера є економічно недоцільним, особливо для стартапів та малих проектів.

Важливим висновком дослідження є розуміння, що жоден з підходів не є універсально кращим у всіх сценаріях. У багатьох випадках оптимальним рішенням може бути комбінація обох архітектур у рамках однієї системи.

Гібридний підхід може бути реалізований наступним чином:

– критичні операції, що потребують миттєвої відповіді та синхронної верифікації (наприклад, підтвердження важливих транзакцій, авторизація користувачів), можуть виконуватися через REST API;

– масова обробка даних, фонові процеси та некритичні сповіщення можуть бути делеговані асинхронній системі на базі Kafka, що забезпечить високу пропускну здатність та надійність;

– при необхідності можлива організація складної взаємодії, коли ініціювання процесу відбувається через REST API, а подальша обробка та сповіщення про результати – через Kafka.

Таке поєднання дозволяє використати переваги обох підходів: оперативність та простоту REST API для критичних операцій і високу продуктивність та надійність Kafka для масової обробки даних.

Результати дослідження показують, що архітектура на основі Apache Kafka має значно кращі характеристики масштабованості, що робить її привабливим

вибором для систем з потенціалом зростання. При проектуванні системи доцільно заздалегідь враховувати можливість масштабування у майбутньому, навіть якщо початкові обсяги навантаження не є критичними.

Водночас, для систем, де передбачається стабільне навантаження без значного зростання в майбутньому, використання REST API може бути достатнім та більш економічно обґрунтованим рішенням через простоту впровадження та підтримки.

Отже, при виборі архітектури необхідно враховувати не лише поточні вимоги, але й передбачуваний розвиток системи в майбутньому, щоб забезпечити оптимальне співвідношення між витратами на впровадження та здатністю системи задовольняти зростаючі потреби бізнесу.

4 РОЗРОБКА ГІБРИДНОЇ АРХІТЕКТУРИ ОБРОБКИ СПОВІЩЕНЬ

4.1 Теоретичні основи гібридних архітектурних рішень

У сучасних розподілених системах все частіше виникає потреба у поєднанні різних архітектурних стилів для забезпечення оптимальної продуктивності, масштабованості та надійності. Гібридні архітектурні рішення, що поєднують різні підходи до комунікації між компонентами, дозволяють використовувати переваги кожного з них, одночасно мінімізуючи їхні недоліки. Особливо актуальним є комбінування синхронних та асинхронних патернів взаємодії, що дозволяє створювати системи, які ефективно працюють у різних умовах навантаження та мають високу відмовостійкість [8].

Гібридна архітектура у контексті обробки сповіщень – це підхід, який передбачає одночасне використання різних механізмів передачі та обробки повідомлень залежно від їхніх характеристик та поточного стану системи. На відміну від традиційних підходів, які покладаються виключно на синхронну взаємодію (як REST API) або асинхронну (як Apache Kafka), гібридний підхід вводить адаптивний шар, який приймає рішення щодо оптимального способу обробки кожного сповіщення.

Концептуально гібридна архітектура складається з наступних компонентів:

– єдиний вхідний шлюз (Gateway), який приймає всі запити;

– система прийняття рішень (Decision Engine), яка визначає оптимальний маршрут;

– синхронний обробник (REST Processor) для запитів, що потребують миттєвої відповіді;

– асинхронний обробник (Kafka Pipeline) для запитів, які можуть бути оброблені з затримкою.

Основний принцип роботи гібридної архітектури полягає у застосуванні правильного інструменту для кожної конкретної задачі, базуючись на чітко визначених критеріях.

Розглянемо побудову ефективної гібридної архітектури. Вона базується на кількох фундаментальних теоретичних принципах:

– принцип розділення відповідальності (Separation of Concerns), який передбачає чіткий розподіл функцій між компонентами системи. У гібридній архітектурі це означає, що кожен з каналів комунікації (синхронний та асинхронний) відповідає за обробку тих типів сповіщень, для яких він найкраще підходить;

– адаптивність (Adaptivity), що дозволяє системі динамічно вибирати оптимальний канал комунікації на основі контексту. Цей принцип реалізується через механізм прийняття рішень, який аналізує характеристики кожного запиту та поточний стан системи;

– стійкість до відмов (Resilience), який забезпечує надійність системи навіть при збоях окремих компонентів. Гібридна архітектура дозволяє перенаправляти трафік між каналами при виникненні проблем, що забезпечує безперервність обслуговування;

– оптимальне використання ресурсів (Resource Optimization), який забезпечує ефективний розподіл навантаження між компонентами системи та мінімізує витрати на інфраструктуру.

Розглянемо переваги та виклики гібридного підходу. З теоретичної точки зору, гібридний підхід до обробки сповіщень надає ряд суттєвих переваг порівняно з використанням лише одного типу архітектури:

– оптимальне співвідношення між продуктивністю та надійністю. Синхронний підхід забезпечує низьку латентність для критичних операцій, тоді як асинхронний – високу пропускну здатність для масових операцій та гарантовану доставку під час пікових навантажень;

– підвищена відмовостійкість. Навіть при відмові одного з каналів комунікації система може продовжувати функціонування через інший канал, забезпечуючи безперервність бізнес-процесів;

– гнучкість у масштабуванні. Гібридна архітектура дозволяє масштабувати кожен з каналів незалежно, відповідно до потреб системи, що оптимізує витрати на інфраструктуру;

– можливість диференційованої обробки різних типів сповіщень. Критичні сповіщення можуть оброблятися через синхронний канал для забезпечення миттєвої відповіді, тоді як масові нетермінові сповіщення можуть бути направлені через асинхронний канал.

Незважаючи на значні переваги, гібридна архітектура також створює певні теоретичні виклики, які необхідно враховувати при проектуванні системи:

– основним викликом є збільшення складності системи. Впровадження двох різних механізмів комунікації вимагає розробки додаткових компонентів для маршрутизації та прийняття рішень, що ускладнює архітектуру та підвищує ризик помилок;

– забезпечення узгодженості даних. При використанні різних каналів комунікації можуть виникати проблеми з узгодженістю даних, особливо у випадках, коли система має обробляти взаємопов’язані повідомлення;

– складність моніторингу та діагностики. Гібридна архітектура вимагає комплексного підходу до моніторингу, який охоплює обидва канали комунікації та дозволяє виявляти проблеми на ранніх стадіях;

– оптимізація алгоритму прийняття рішень. Ефективність гібридної архітектури значною мірою залежить від якості алгоритму, який визначає оптимальний канал для кожного запиту. Розробка та налаштування такого алгоритму вимагає глибокого розуміння характеристик системи та навантаження.

Теоретичні основи гібридних архітектурних рішень для обробки сповіщень демонструють значний потенціал для підвищення загальної ефективності системи. Комбінування синхронного підходу (REST-API) з асинхронним (Apache Kafka) дозволяє створити рішення, яке поєднує переваги обох підходів та мінімізує їхні недоліки.

Ключовим елементом такої архітектури є механізм прийняття рішень, який визначає оптимальний канал комунікації для кожного запиту на основі його

характеристик та поточного стану системи. Розробка ефективного алгоритму прийняття рішень є однією з найважливіших задач при проектуванні гібридної архітектури.

Хоча гібридний підхід підвищує складність системи та створює додаткові виклики для розробників, теоретичні переваги, які він надає у контексті продуктивності, надійності та масштабованості, роблять його привабливим рішенням для сучасних високонавантажених систем обробки сповіщень.

4.2 Проєктування гібридної архітектури з використанням REST API та Apache Kafka

На основі теоретичних принципів гібридних архітектурних рішень, розглянутих у попередньому розділі, пропонується проєктування гібридної системи обробки сповіщень, що поєднує синхронний REST API та асинхронний Apache Kafka. Ця архітектура спрямована на максимальне використання переваг обох підходів при мінімізації їхніх недоліків.

Порівняльний аналіз синхронного REST API та асинхронного підходу з Apache Kafka був представлений на XXIX Міжнародному молодіжному форумі «Радіоелектроніка та молодь у XXI столітті» [15]. У роботі «Порівняльний аналіз архітектури обробки сповіщень на стороні back-end: синхронного REST API та асинхронного підходу з Apache Kafka» були викладені результати дослідження обох архітектурних підходів, їх переваг та недоліків в різних умовах навантаження. На основі цих досліджень в даній кваліфікаційній роботі розроблено концепцію гібридної архітектури, що дозволяє ефективно поєднувати переваги обох підходів.

Запропонована гібридна архітектура складається з кількох ключових компонентів, які взаємодіють між собою для забезпечення ефективної обробки сповіщень. Центральними компонентами системи є Notification Gateway, який виступає єдиною точкою входу для всіх запитів на обробку сповіщень та надає уніфікований API для клієнтів, та Notification Router – ключовий компонент, який аналізує вхідні запити та приймає рішення щодо оптимального каналу обробки для кожного сповіщення. Router використовує спеціальний алгоритм для визначення,

чи слід обробляти запит через синхронний REST API або асинхронний Kafka канал. REST Processor відповідає за синхронну обробку сповіщень, які потребують миттєвої відповіді, та напряму взаємодіє з базою даних. Kafka Producer забезпечує публікацію сповіщень у відповідні Kafka топіки для асинхронної обробки. Kafka Cluster виступає розподіленою системою обміну повідомленнями, що забезпечує надійне зберігання та доставку сповіщень. Kafka Consumer споживає повідомлення з Kafka топіків, обробляє їх та зберігає результати у базі даних. База даних (Database) використовується як сховище для зберігання інформації про сповіщення та метаданих. Така структура забезпечує чіткий розподіл відповідальності між компонентами та гнучкість у виборі оптимального шляху обробки для різних типів сповіщень.

На рисунку 4.1 представлена високорівнева діаграма запропонованої гібридної архітектури.

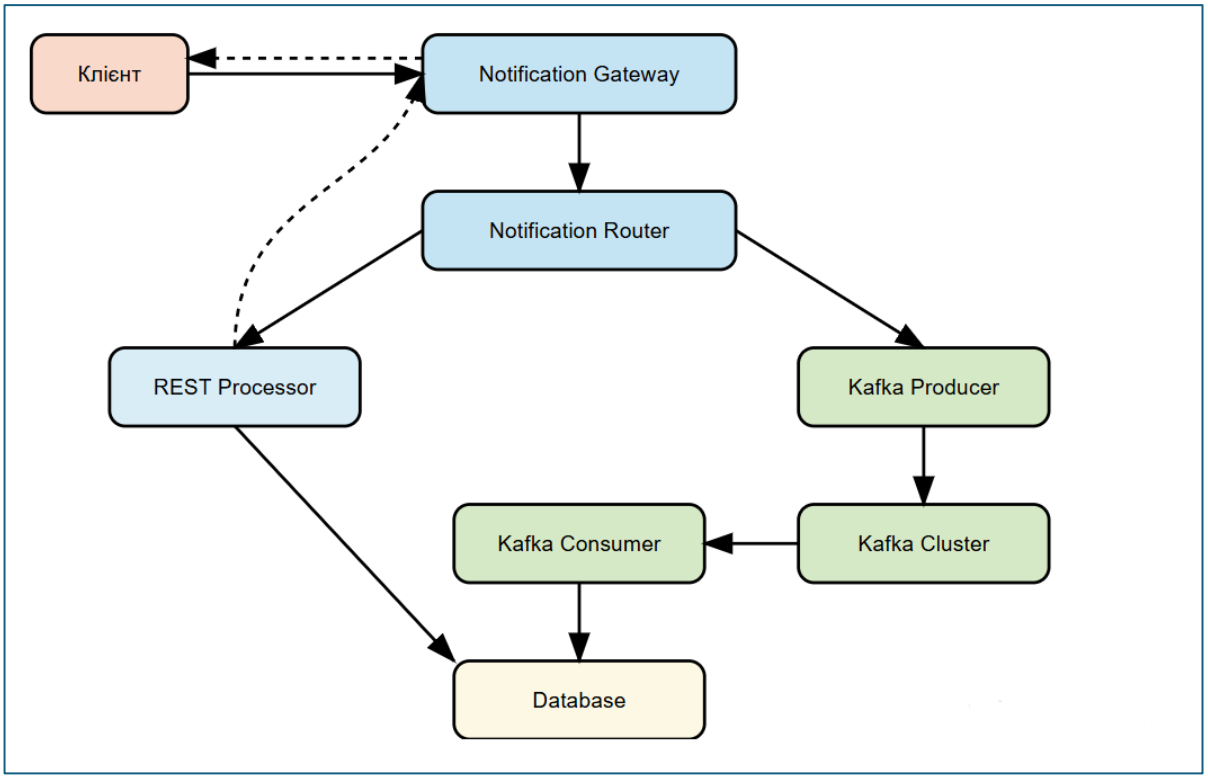

Рисунок 4.1 – Гібридна архітектура системи обробки сповіщень (рисунок створено самостійно)

Взаємодія між компонентами гібридної архітектури базується на чітко визначених принципах, які забезпечують ефективну роботу системи в різних умовах. Єдиний вхідний інтерфейс гарантує, що всі запити надходять через єдиний Gateway, що спрощує інтеграцію з зовнішніми системами. Незалежна маршрутизація дозволяє Notification Router приймати рішення індивідуально для кожного запиту, базуючись на його характеристиках та поточному стані системи. Ізоляція каналів забезпечує, що REST Processor та Kafka Pipeline працюють незалежно один від одного, що дозволяє уникнути каскадних відмов та забезпечує стійкість системи. Автоматичне переключення активується у випадку недоступності одного з каналів, коли система автоматично переключається на інший канал, що гарантує безперервність обслуговування. Паралельна обробка дозволяє Kafka Consumer обробляти повідомлення одночасно, що підвищує загальну пропускну здатність системи.

Процес обробки сповіщень у гібридній архітектурі включає кілька етапів. Спочатку Gateway отримує запит на обробку сповіщення від клієнта та виконує первинну валідацію даних. Далі Notification Router аналізує характеристики запиту (тип сповіщення, розмір батчу, необхідність підтвердження) та визначає оптимальний канал для його обробки. Запит направляється або до REST Processor для синхронної обробки, або до Kafka Producer для асинхронної обробки, залежно від рішення Router. Якщо запит направлено через REST канал, REST Processor обробляє його, зберігає дані в базі та негайно повертає результат клієнту. Якщо ж запит направлено через Kafka канал, Kafka Producer публікує повідомлення в Kafka Cluster, звідки воно споживається Kafka Consumer для подальшої обробки та збереження в базі даних. Такий процес забезпечує гнучкість у виборі оптимального шляху обробки для різних типів сповіщень та максимальну ефективність системи в різних умовах навантаження.

Гібридна архітектура включає кілька механізмів, які забезпечують надійність та відмовостійкість системи обробки сповіщень. Для REST каналу реалізується механізм повторних спроб з експоненційною затримкою, який автоматично повторює запити при тимчасових проблемах. Kafka забезпечує гарантовану

доставку повідомлень та їх обробку навіть при тимчасовій недоступності Consumer. Реплікація даних між брокерами в Kafka забезпечує збереження повідомлень при відмові окремих вузлів. Kafka Producer може буферизувати повідомлення при тимчасовій недоступності кластера, забезпечуючи їх збереження та подальшу обробку. У випадку недоступності одного з каналів система автоматично переключається на інший канал. При надмірному навантаженні на REST канал запити можуть автоматично перенаправлятися на Kafka для рівномірного розподілу навантаження. Ці механізми забезпечують високу надійність системи навіть при відмовах окремих компонентів та пікових навантаженнях.

Гібридна архітектура надає різноманітні можливості для масштабування системи обробки сповіщень відповідно до зростаючих потреб. Для REST каналу можливе горизонтальне масштабування через додавання нових екземплярів REST Processor та використання балансувальника навантаження. Для Kafka каналу масштабування відбувається через збільшення кількості партицій у топіках та додавання нових брокерів до кластера. Це забезпечує рівномірний розподіл навантаження та високу пропускну здатність системи. Для забезпечення оптимального масштабування необхідно також враховувати вертикальне масштабування окремих компонентів (збільшення обчислювальних ресурсів) та масштабування бази даних для підтримки зростаючого обсягу даних. Гібридна архітектура дозволяє масштабувати кожен канал незалежно, що забезпечує оптимальне використання ресурсів та гнучкість у відповіді на змінні вимоги до системи.

Проєктування гібридної архітектури обробки сповіщень, що поєднує синхронний REST API та асинхронний Apache Kafka, дозволяє створити систему, яка ефективно працює в різних умовах навантаження та має високу надійність. Ключовим елементом такої архітектури є механізм прийняття рішень, який динамічно визначає оптимальний канал для обробки кожного запиту. Запропонована архітектура включає механізми забезпечення надійності, масштабованості, які дозволяють системі адаптуватися до змінних вимог та

забезпечувати безперервність бізнес-процесів. Хоча такий підхід підвищує складність системи, теоретичні переваги, які він надає у контексті продуктивності та надійності, роблять його привабливим рішенням для сучасних високонавантажених систем обробки сповіщень.

4.3 Алгоритм прийняття рішень щодо вибору каналу передачі сповіщень

Центральним елементом запропонованої гібридної архітектури є алгоритм прийняття рішень, який реалізується в компоненті Notification Router. Цей алгоритм визначає оптимальний канал для обробки кожного сповіщення, базуючись на його характеристиках та поточних умовах системи. Ефективність гібридної архітектури напряму залежить від якості цього алгоритму, тому його розробці необхідно приділити особливу увагу.

В основі запропонованого алгоритму лежить оцінка кількох ключових параметрів сповіщення, які найбільше впливають на вибір оптимального каналу обробки. На основі аналізу характеристик систем обробки сповіщень та особливостей їх застосування було виділено три основні фактори, які є найбільш значущими для прийняття рішення:

– тип сповіщення (T) – характеризує семантичну природу повідомлення та його критичність для бізнес-процесів. Різні типи сповіщень мають різні вимоги до часу обробки та гарантій доставки;

– розмір батчу (B) – кількість повідомлень, які необхідно обробити одночасно. Цей параметр суттєво впливає на ефективність обробки та навантаження на систему;

– необхідність підтвердження доставки (C) – визначає, чи потребує клієнт миттєвого підтвердження отримання та обробки сповіщення.

Для формалізації процесу прийняття рішень пропонується математична модель, яка дозволяє оцінити «схильність» до використання того чи іншого каналу обробки на основі значень вхідних параметрів. Ця модель представлена формулою:

$$R = w_1 \times T + w_2 \times (1 - min(B, Bmax)/Bmax) + w_3 \times C \qquad (1)$$

де R – результуючий показник, який визначає схильність до використання REST API (вищі значення) або Kafka (нижчі значення),

T – числове значення, яке відповідає типу сповіщення (наприклад, TRANSACTION = 3, SYSTEM = 2, USER = 1),

B – розмір батчу (кількість повідомлень),

$Bmax$ – максимальний розмір батчу, що розглядається (наприклад, 1000 повідомлень),

C – необхідність підтвердження доставки (REQUIRED = 1, OPTIONAL = 0),

w_1, w_2, w_3 – вагові коефіцієнти, які визначають важливість відповідних параметрів.

Для прийняття рішення щодо вибору каналу використовується порогове значення θ:

– якщо R > θ і REST-канал доступний: використовуємо REST API;

– інакше, якщо Kafka-канал доступний: використовуємо Kafka;

– інакше: використовуємо резервний механізм (наприклад, локальне збереження з відкладеною обробкою).

Тип сповіщення (T) є найважливішим фактором при виборі каналу обробки. Різні типи сповіщень мають різні вимоги до швидкості обробки та гарантій доставки. Транзакційні сповіщення ($T = 3$), такі як підтвердження платежів або зміни статусу замовлення, вимагають швидкої обробки та миттєвого зворотного зв'язку, тому для них більш підходить REST API. Системні сповіщення ($T = 2$), такі як оновлення статусу системи або сповіщення про технічні роботи, мають середній пріоритет. Користувацькі сповіщення ($T = 1$), такі як оповіщення про нові можливості або маркетингові повідомлення, зазвичай не є критичними за часом і можуть бути ефективно оброблені через Kafka.

Розмір батчу (B) також суттєво впливає на вибір каналу обробки. Для невеликих батчів REST API забезпечує низьку латентність та простоту відстеження статусу обробки. Однак при збільшенні розміру батчу ефективність REST API знижується через необхідність підтримувати велику кількість HTTP-з'єднань та

очікувати відповіді для кожного запиту. Для великих батчів (сотні або тисячі повідомлень) Kafka забезпечує кращу пропускну здатність та ефективність обробки. Компонент $(1 - min(B, Bmax)/Bmax)$ в формулі дає значення, близьке до 1 для малих батчів (що схиляє до вибору REST API) і близьке до 0 для великих батчів (що схиляє до вибору Kafka).

Необхідність підтвердження доставки (C) визначає, чи потребує клієнт миттєвого підтвердження обробки сповіщення. Якщо підтвердження обов'язкове $(C = 1)$, це схиляє до вибору REST API, який забезпечує негайну відповідь. Якщо підтвердження опціональне $(C = 0)$, це схиляє до вибору Kafka, який забезпечує вищу пропускну здатність та надійність, але не надає миттєвого підтвердження.

Для практичного використання запропонованої моделі необхідно визначити оптимальні значення вагових коефіцієнтів та порогового значення. На основі експертної оцінки та аналізу характеристик обох каналів обробки пропонуються такі початкові значення:

- w_1 = 0.4 (важливість типу сповіщення);
- w_2 = 0.3 (важливість розміру батчу);
- w_3 = 0.3 (важливість необхідності підтвердження);
- θ = 0.6 (поріг вибору REST API).

Ці значення можуть бути адаптовані відповідно до специфіки конкретної системи та характеристик навантаження. Для наочного представлення процесу прийняття рішень можна побудувати матрицю рішень, яка показує вибір каналу для різних комбінацій параметрів T, B та C. Така матриця дозволяє візуалізувати логіку алгоритму та проаналізувати його поведінку в різних умовах.

Практична реалізація алгоритму передбачає створення окремого компонента Decision Engine у складі Notification Router, який буде виконувати обчислення за наведеною формулою та приймати рішення про маршрутизацію кожного сповіщення в режимі реального часу. Важливим аспектом є також впровадження системи моніторингу ефективності прийнятих рішень, яка дозволить відстежувати відсоток правильних маршрутизацій та динамічно коригувати вагові коефіцієнти на основі аналізу фактичної продуктивності обох каналів обробки. Алгоритм має

також передбачати механізм fallback для ситуацій, коли обидва канали тимчасово недоступні, забезпечуючи збереження сповіщень у локальній черзі з подальшою обробкою після відновлення роботи системи. Додатково рекомендується впровадити систему A/B тестування для порівняння ефективності різних конфігурацій алгоритму в умовах реального навантаження.

Псевдокод алгоритму прийняття рішень зображено на рисунку нижче.

```
1  функція route_decision(notification_type, batch_size,
   confirmation_required, system_state):
2  T = get_type_value(notification_type)
3  B = batch_size
4  C = 1 if confirmation_required else 0
5  Hrest = is_rest_available(system_state)
6  Hkafka = is_kafka_available(system_state)
7
8  R = w₁ * T + w₂ * (1 - min(B, Bmax)/Bmax) + w₃ * C
9
10 if R > θ and Hrest:
11 return "REST_API"
12 elif Hkafka:
13 return "KAFKA"
14 else:
15 return "FALLBACK_MECHANISM"
```

Рисунок 4.2 – Псевдокод алгоритму прийняття рішень (рисунок створено самостійно)

Цей алгоритм може бути реалізований в компоненті Notification Router як ключовий механізм маршрутизації сповіщень. Завдяки своїй простоті та ефективності він забезпечує швидке прийняття рішень без значного навантаження на систему.

Запропонований алгоритм прийняття рішень має кілька важливих переваг:

– простота реалізації – алгоритм базується на простих математичних операціях та не вимагає складних обчислень, що забезпечує його ефективність в умовах високого навантаження;

– гнучкість – через налаштування вагових коефіцієнтів та порогового значення алгоритм може бути адаптований до різних вимог та умов роботи системи;

– розширюваність – за необхідності алгоритм може бути доповнений додатковими параметрами або модифікований для врахування специфічних вимог конкретної системи;

– прозорість – логіка прийняття рішень є ясною та детермінованою, що спрощує аналіз та налагодження системи.

Таким чином, запропонований алгоритм прийняття рішень є ефективним інструментом для оптимізації обробки сповіщень у гібридній архітектурі, що поєднує REST API та Apache Kafka. Він дозволяє максимально використати переваги обох підходів, забезпечуючи оптимальну продуктивність, надійність та масштабованість системи в різних умовах навантаження.

4.4 Практичні рекомендації щодо вибору та впровадження архітектур обробки сповіщень

На основі проведеного порівняльного аналізу синхронного REST API та асинхронного підходу з Apache Kafka, а також теоретичного обґрунтування можливості їх комбінування у гібридній архітектурі, можна сформулювати ряд практичних рекомендацій, які допоможуть архітекторам програмного забезпечення приймати обґрунтовані рішення при проєктуванні систем обробки сповіщень.

У роботі, представленій на XXIX Міжнародному молодіжному форумі «Радіоелектроніка та молодь у XXI столітті» [15], було проведено порівняльний аналіз синхронного REST API та асинхронного підходу з Apache Kafka, який став основою для формулювання практичних рекомендацій, представлених у даній кваліфікаційній роботі. Розроблені рекомендації щодо вибору оптимальної архітектури базуються на результатах експериментальних досліджень та теоретичному аналізі, що забезпечує їх практичну цінність для архітекторів програмного забезпечення.

Вибір оптимальної архітектури обробки сповіщень має ґрунтуватися на аналізі конкретних вимог та обмежень проєкту. Не існує універсального рішення, яке було б оптимальним для всіх можливих сценаріїв використання. Натомість, необхідно враховувати специфіку проєкту та обирати підхід, який найкраще відповідає його вимогам.

Синхронний підхід з використанням REST API рекомендується застосовувати у наступних випадках:

– для систем з невеликим або середнім навантаженням (до 1000 запитів на секунду), де питання продуктивності не є критичними. Як показало дослідження, при невеликих навантаженнях різниця у продуктивності між REST API та Kafka є незначною, тому доцільно використовувати простіший у реалізації та підтримці підхід.

– для систем, де критично важлива миттєва відповідь на запит. Синхронний підхід забезпечує негайне підтвердження обробки сповіщення, що може бути важливо для деяких бізнес-процесів;

– для систем з обмеженими ресурсами, де розгортання та підтримка додаткової інфраструктури (такої як Kafka кластер) може бути економічно недоцільним. REST API вимагає меншої кількості компонентів та простішої інфраструктури;

Асинхронний підхід з використанням Apache Kafka є оптимальним вибором у наступних сценаріях:

– для високонавантажених систем, що обробляють тисячі або десятки тисяч повідомлень на секунду. Проведене дослідження показало, що при навантаженні в 100 000 повідомлень Kafka демонструє втричі кращу продуктивність порівняно з REST API;

– для систем з високими вимогами до надійності та гарантованої доставки повідомлень. Kafka забезпечує збереження повідомлень на диску та їх реплікацію між брокерами, що мінімізує ризик втрати даних навіть при відмові окремих компонентів системи;

– для систем, де важлива масштабованість та здатність ефективно обробляти пікові навантаження. Архітектура Kafka з партиціонуванням топіків та розподілом навантаження між споживачами дозволяє легко масштабувати систему для обробки зростаючого обсягу даних;

– для систем з розподіленою архітектурою, де компоненти можуть бути розгорнуті в різних географічних локаціях. Kafka забезпечує надійну комунікацію між розподіленими компонентами та мінімізує проблеми, пов’язані з мережевими затримками та нестабільністю з’єднання.

У деяких випадках найбільш ефективним рішенням може бути гібридний підхід, який поєднує синхронну та асинхронну обробку сповіщень. Цей підхід рекомендується розглянути у наступних ситуаціях:

Для систем з різними типами сповіщень, частина з яких вимагає миттєвої обробки та підтвердження, а інша частина може бути оброблена з затримкою. Гібридний підхід дозволяє використовувати оптимальний канал для кожного типу сповіщень.

Для систем з нерівномірним навантаженням, де в періоди пікового навантаження важлива висока пропускна здатність, а в періоди низького навантаження – простота та зручність супроводу. Гібридний підхід дозволяє адаптивно вибирати оптимальний канал обробки залежно від поточного стану системи.

Для систем з високими вимогами до відмовостійкості, де важливо забезпечити безперервну роботу навіть при відмові окремих компонентів. Гібридний підхід з механізмом автоматичного переключення між каналами підвищує загальну надійність системи.

При впровадженні системи обробки сповіщень незалежно від обраної архітектури рекомендується дотримуватися наступних принципів:

– модульна структура системи з чітким розділенням відповідальності між компонентами спрощує розробку, тестування та подальший супровід. Кожен компонент має відповідати за одну конкретну функцію та мати чітко визначені інтерфейси взаємодії з іншими компонентами;

– моніторинг та метрики є критично важливими для забезпечення стабільної роботи системи. Необхідно впровадити комплексний моніторинг, який охоплює всі компоненти системи та дозволяє своєчасно виявляти проблеми та аномалії.

Для забезпечення ефективного масштабування системи обробки сповіщень при зростанні навантаження рекомендується:

– для REST API реалізувати горизонтальне масштабування через додавання нових екземплярів сервісів та використання балансувальника навантаження. При цьому важливо забезпечити синхронізацію даних між екземплярами для підтримки консистентності системи;

– для Kafka оптимізувати конфігурацію через визначення оптимальної кількості партицій для кожного топіка та їх розподілу між брокерами. Кількість партицій має відповідати очікуваному максимальному числу паралельних споживачів, а розподіл – забезпечувати рівномірне навантаження на брокери;

– для гібридної архітектури забезпечити незалежне масштабування кожного з каналів відповідно до характеристик навантаження та вимог системи. При цьому алгоритм прийняття рішень має враховувати поточний стан та навантаження кожного каналу.

Для підвищення надійності системи обробки сповіщень та мінімізації ризику втрати даних рекомендується:

– для REST API реалізувати механізм повторних спроб з експоненційною затримкою на стороні клієнта та механізм відкладеної обробки для запитів, які не вдалося обробити відразу. Також важливо впровадити патерн Circuit Breaker для запобігання каскадним відмовам при проблемах з окремими компонентами системи;

– для Kafka налаштувати оптимальний рівень підтверджень (acks) та фактор реплікації для забезпечення надійного збереження даних навіть при відмові окремих брокерів. Також рекомендується використовувати ручне підтвердження обробки повідомлень (manual acknowledgment) для гарантії, що повідомлення не буде втрачено до його успішної обробки;

– для гібридної архітектури реалізувати механізм автоматичного переключення між каналами при виникненні проблем та систему моніторингу стану каналів для своєчасного виявлення та реагування на збої.

Оптимальне налаштування системи обробки сповіщень вимагає розуміння конкретних характеристик навантаження та особливостей інфраструктури. Рекомендується проводити тестування з різними конфігураціями для визначення оптимальних значень параметрів, таких як розмір батчу, частота вивільнення, таймаути та інші налаштування, які впливають на продуктивність та надійність системи.

Представлені рекомендації ґрунтуються на результатах проведеного дослідження та спрямовані на допомогу архітекторам програмного забезпечення у виборі та впровадженні оптимальної архітектури обробки сповіщень. Вибір конкретного підходу має ґрунтуватися на аналізі вимог та обмежень конкретного проєкту, а також на розумінні сильних та слабких сторін кожного з розглянутих архітектурних рішень.

Підсумовуючи, можна зазначити, що синхронний REST API та асинхронний Apache Kafka є потужними інструментами для побудови систем обробки сповіщень, кожен з яких має свої переваги та обмеження. Їх ефективне використання, або в деяких випадках комбінування у гібридну архітектуру, дозволяє створювати надійні, масштабовані та продуктивні системи, які відповідають сучасним вимогам до обробки сповіщень.

ВИСНОВКИ

В результаті проведеного дослідження архітектурних підходів до обробки сповіщень на стороні back-end, зокрема порівняльного аналізу синхронного REST API та асинхронного підходу з Apache Kafka, можна зробити наступні висновки.

Проведений аналіз предметної галузі показав актуальність проблеми вибору оптимальної архітектури обробки сповіщень у сучасних мікросервісних системах. Зростання обсягів оброблюваних сповіщень у корпоративних системах на 40% щорічно створює нові виклики для архітекторів програмного забезпечення та підкреслює важливість дослідження ефективних архітектурних рішень.

Для забезпечення об’єктивного порівняння архітектурних підходів було розроблено комплексну систему метрик, яка охоплює ключові аспекти продуктивності, надійності та експлуатаційних характеристик систем. Це дозволило всебічно оцінити кожен із підходів та виявити їх сильні та слабкі сторони.

Дослідження синхронної архітектури на основі REST API показало, що цей підхід забезпечує простоту реалізації, миттєвий зворотній зв’язок та легку інтеграцію з існуючими системами. Водночас, він має обмеження щодо продуктивності при високих навантаженнях та масштабованості системи. Синхронний підхід є ефективним вибором для систем з невеликим або середнім навантаженням (до 1000 запитів на секунду), де важлива простота розробки та швидкість впровадження.

Аналіз асинхронної архітектури з Apache Kafka продемонстрував її переваги у контексті високої пропускної здатності, надійності зберігання даних та масштабованості системи. Kafka забезпечує гарантовану доставку повідомлень, стійкість до відмов компонентів та ефективну обробку пікових навантажень. Цей підхід є оптимальним для високонавантажених систем, що обробляють тисячі або десятки тисяч повідомлень на секунду, та систем з високими вимогами до надійності.

Порівняльний аналіз механізмів обробки помилок показав суттєві відмінності між підходами. REST API забезпечує просту та прозору обробку помилок з миттєвим зворотним зв’язком, але вимагає додаткової реалізації механізмів відмовостійкості. Kafka надає вбудовані механізми автоматичних повторних спроб, Dead Letter Queue та контролю offset, що забезпечує високу надійність системи, але ускладнює процес діагностики через асинхронну природу взаємодії.

Дослідження підходів до масштабування виявило, що Kafka демонструє кращу ефективність масштабування завдяки можливості паралельної обробки повідомлень через партиції, ефективному розподілу навантаження між брокерами та асинхронній природі обробки повідомлень. REST API має певні обмеження масштабування через необхідність підтримки активних з’єднань та синхронну природу взаємодії.

Експериментальне дослідження ефективності архітектурних підходів підтвердило теоретичні висновки та надало кількісні оцінки їх продуктивності та надійності. При навантаженні в 100 000 повідомлень асинхронний підхід з Apache Kafka продемонстрував втричі кращу продуктивність порівняно з REST API. Особливо значні відмінності спостерігалися при моделюванні сценаріїв переривань роботи системи, де Kafka забезпечила 100% доставку повідомлень без втрат.

На основі отриманих результатів була розроблена концепція гібридної архітектури обробки сповіщень, яка поєднує переваги синхронного та асинхронного підходів. Ключовим елементом такої архітектури є механізм прийняття рішень щодо вибору оптимального каналу обробки для кожного сповіщення, який базується на аналізі типу сповіщення, розміру батчу та необхідності підтвердження доставки.

Запропоновано математичну модель та алгоритм прийняття рішень, який дозволяє динамічно вибирати оптимальний канал обробки для кожного сповіщення з урахуванням його характеристик та поточного стану системи. Цей алгоритм враховує такі параметри як тип сповіщення, розмір батчу та необхідність

підтвердження доставки, що забезпечує оптимальне використання ресурсів системи.

Сформульовано практичні рекомендації щодо вибору та впровадження архітектури обробки сповіщень залежно від конкретних вимог та обмежень проєкту. Ці рекомендації охоплюють аспекти продуктивності, надійності, масштабованості та експлуатаційних характеристик систем, що дозволяє архітекторам програмного забезпечення приймати обґрунтовані рішення при проєктуванні систем обробки сповіщень.

У рамках дослідження було створено працюючі прототипи систем обробки сповіщень на основі обох архітектурних підходів, що дозволило провести детальне експериментальне порівняння їх ефективності в різних умовах навантаження та при моделюванні збоїв. Результати експериментів підтвердили теоретичні висновки та стали основою для формулювання практичних рекомендацій.

Основні результати дослідження, зокрема порівняльний аналіз синхронного REST API та асинхронного підходу з Apache Kafka, були представлені на XXIX Міжнародному молодіжному форумі «Радіоелектроніка та молодь у XXI столітті», де отримали позитивні відгуки від наукової спільноти. Розроблена концепція гібридної архітектури та алгоритм прийняття рішень щодо вибору оптимального каналу обробки можуть бути використані при проєктуванні та розробці реальних систем обробки сповіщень, що підтверджує практичну цінність отриманих результатів.

Таким чином, проведене дослідження надає комплексне розуміння особливостей синхронного REST API та асинхронного підходу з Apache Kafka для обробки сповіщень на стороні back-end, а також пропонує нові підходи до їх ефективного поєднання в рамках гібридної архітектури. Результати дослідження мають практичну цінність для архітекторів програмного забезпечення та можуть бути використані при проєктуванні високонавантажених, надійних та масштабованих систем обробки сповіщень в різних галузях програмної інженерії.

ПЕРЕЛІК ДЖЕРЕЛ ПОСИЛАННЯ

1. Newman S. Building Microservices: Designing Fine-Grained Systems. 2nd ed. O'Reilly Media, 2021. 280 p. URL: https://www.oreilly.com/library/view/building-microservices-2nd/9781492034018/ (дата звернення: 28.04.2025).

2. Richardson C. Microservices Patterns: With Examples in Java. Manning Publications, 2018. 520 p. URL: https://microservices.io/patterns/index.html (дата звернення: 02.05.2025).

3. Fielding R. T. Architectural Styles and the Design of Network-based Software Architectures: dissertation. University of California, Irvine, 2000. 180 p. URL: https://www.ics.uci.edu/~fielding/pubs/dissertation/top.htm (дата звернення: 05.05.2025).

4. Narkhede N., Shapira G., Palino T. Kafka: The Definitive Guide: Real-Time Data and Stream Processing at Scale. O'Reilly Media, 2017. 322 p. URL: https://learning.oreilly.com/library/view/kafka-the-definitive/9781491936153/ (дата звернення: 08.05.2025).

5. Reinsel D., Gantz J., Rydning J. The Digitization of the World From Edge to Core. IDC White Paper, 2018. URL: https://www.seagate.com/files/www-content/our-story/trends/files/idc-seagate-dataage-whitepaper.pdf (дата звернення: 12.04.2025).

6. Soldani J., Tamburri D. A., Van Den Heuvel W. J. The pains and gains of microservices: A Systematic grey literature review. Journal of Systems and Software. 2018. Vol. 146. P. 215-232. DOI: 10.1016/j.jss.2018.09.082 (дата звернення: 15.05.2025).

7. Kleppmann M. Designing Data-Intensive Applications: The Big Ideas Behind Reliable, Scalable, and Maintainable Systems. O'Reilly Media, 2017. 616 p. URL: https://learning.oreilly.com/library/view/designing-data-intensive-applications/9781491903063/ (дата звернення: 20.05.2025).

8. Vernon V. Implementing Domain-Driven Design. Addison-Wesley Professional, 2013. 656 p. URL: https://www.domainlanguage.com/ddd/reference/ (дата звернення: 25.05.2025).

9. Palumbo F., Zamboni G. Kafka in Action. Manning Publications, 2021. 368 p. URL: https://kafka.apache.org/quickstart (дата звернення: 03.04.2025).

10. Gough J. Building High-Performance RESTful Web Services. Packt Publishing, 2017. 392 p. URL: https://restfulapi.net/rest-architectural-constraints/ (дата звернення: 28.05.2025).

11. Armbrust M., Fox A., Griffith R., Joseph A. D. A view of cloud computing. Communications of the ACM. 2010. Vol. 53, № 4. P. 50-58. DOI: 10.1016/j.jss.2018.09.082 (дата звернення: 28.05.2025).

12. Garg N. Apache Kafka. Packt Publishing, 2013. 68 p. URL: https://kafka.apache.org/28/documentation/streams/ (дата звернення: 17.04.2025).

13. Zolotariov D. The platform for creation of event-driven applications based on Wolfram Mathematica and Apache Kafka. Інноваційні технології та наукові рішення для промисловості. 2021. № 2 (16). С. 12-18. URL: https://journals.uran.ua/itssi/article/view/236656 (дата звернення: 01.06.2025).

14. Dashkevych O., Shubin I. Analysis of the Apache Kafka capabilities as part of providing Big Data Streaming. Інформаційні системи та технології ІСТ-2018. Харків: ХНУРЕ, 2018. С. 443-445. DOI: 10.30837/ITSSI.2021.16.012 (дата звернення: 01.06.2025).

15. Старіченко В. С., Лановий О. Ф. Порівняльний аналіз архітектури обробки сповіщень на стороні back-end: синхронного REST API та асинхронного підходу з Apache Kafka. Радіоелектроніка та молодь у XXI столітті: Матеріали XXIX Міжнародного молодіжного форуму (16-19 квітня 2025 р.). Т. 6: Інформаційні інтелектуальні системи. Харків: ХНУРЕ, 2025. С. 324-326. URL: https://nure.ua/konferencii-ta-workshops/mizhnarodnij-molodizhnij-forum-radioelektronika-i-molod-u-hhi-stolitti/xxix-mizhnarodnyj-molodizhnyj-forum-radioelektronika-i-molod-u-khkhi-stolitti (дата звернення: 28.05.2025).

ПЕРЕЛІК ДЖЕРЕЛ ПОСИЛАННЯ ЗА НАУКОВИМИ НАПРЯМАМИ КЕРІВНИКА ТА НАУКОВЦІВ КАФЕДРИ ПРОГРАМНОЇ ІНЖЕНЕРІЇ

14. Dashkevych O., Shubin I. Analysis of the Apache Kafka capabilities as part of providing Big Data Streaming. Інформаційні системи та технології ІСТ-2018. Харків: ХНУРЕ, 2018. С. 443-445. DOI: https://doi.org/10.30837/ITSSI.2021.16.012 (дата звернення: 01.06.2025).

15. Старіченко В. С., Лановий О. Ф. Порівняльний аналіз архітектури обробки сповіщень на стороні back-end: синхронного REST API та асинхронного підходу з Apache Kafka. Радіоелектроніка та молодь у XXI столітті: Матеріали XXIX Міжнародного молодіжного форуму (16-19 квітня 2025 р.). Т. 6: Інформаційні інтелектуальні системи. Харків: ХНУРЕ, 2025. С. 324-326. URL: https://nure.ua/konferencii-ta-workshops/mizhnarodnij-molodizhnij-forum-radioelektronika-i-molod-u-hhi-stolitti/xxix-mizhnarodnyj-molodizhnyj-forum-radioelektronika-i-molod-u-khkhi-stolitti (дата звернення: 28.05.2025).