НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ УКРАЇНИ
«КИЇВСЬКИЙ ПОЛІТЕХНІЧНИЙ ІНСТИТУТ
імені ІГОРЯ СІКОРСЬКОГО»

Навчально-науковий інститут телекомунікаційних систем

Кафедра телекомунікацій

До захисту допущено:

Завідувач кафедри

_____ Сергій КРАВЧУК

«___»_____2025 р.

Дипломна робота

на здобуття ступеня бакалавра

за освітньо-професійною програмою «Інженерія та програмування інфокомунікацій»

спеціальності 172 «Телекомунікації та радіотехніка»

на тему: «Метод підвищення ефективності мережевих комунікацій за допомогою Apache Kafka в розподілених системах»

Виконав:
студент IV курсу, групи ТЗ-11
Гриценко Данил Геннадійович _____

Керівник:
Асистент кафедри ТК НН ІТС,
Сайченко Іван Олегович _____

Консультант з практичного розділу 3 і 4:
Старший викладач кафедри ТК НН ІТС, к.т.н.
Маньківський Володимир Броніславович _____

Рецензент:
Доцент кафедри ІТТ НН ІТС, д.т.н.
Астраханцев Андрій Анатолійович _____

Засвідчую, що у цій дипломній роботі немає запозичень з праць інших авторів без відповідних посилань.
Студент _____

Київ – 2025 року

Національний технічний університет України
«Київський політехнічний інститут імені Ігоря Сікорського»
Навчально-науковий інститут телекомунікаційних систем
Кафедра телекомунікацій

Рівень вищої освіти – перший (бакалаврський)

Спеціальність – 172 «Телекомунікації та радіотехніка»

Освітньо-професійна програма «Інженерія та програмування інфокомунікацій»

ЗАТВЕРДЖУЮ
Завідувач кафедри
_____ Сергій КРАВЧУК
«___»_____2025 р.

ЗАВДАННЯ

на дипломну роботу студенту

Гриценку Данилу Геннадійовичу

1. Тема роботи «Метод підвищення ефективності мережевих комунікацій за допомогою Apache Kafka в розподілених системах», керівник роботи Сайченко Іван Олегович, затверджені наказом по університету від «26» травня 2025 р. № 1755-с.

2. Термін подання студентом роботи 9 червня 2025 р.

3. Вихідні дані до роботи: Apache Kafka; Docker; Spring Boot; Java 17; Apache Maven; Apache ZooKeeper; PostgreSQL; Prometheus; Grafana; Kafka UI; Postman; Kafka Streams API; математичне моделювання черг M/M/c.

4. Зміст роботи: Огляд традиційних підходів до обміну повідомленнями в розподілених системах і недоліків синхронної взаємодії. Обґрунтування вибору Apache Kafka для асинхронної комунікації в мікросервісній архітектурі. Розробка тестового середовища з Docker і Kafka, проведення експериментів для аналізу впливу параметрів Kafka на затримки та пропускну здатність. Теоретичне моделювання за допомогою M/M/c і порівняння результатів із класичними брокерами для формування рекомендацій з оптимізації.

5. Перелік ілюстративного матеріалу (із зазначенням плакатів, презентацій тощо)

Слайд №1 Назва роботи

Слайд №2 Мета, актуальність, об’єкт та завдання дослідження

Слайд №3 Традиційні способи обміну повідомленнями

Слайд №4 Архітектура Apache Kafka та принципи її роботи

Слайд №5 Побудова тестового середовища з Docker та Kafka

Слайд №6 Проведення експериментів та їх результати

Слайд №7 Математичне обґрунтування моделі на основі M/M/c

Слайд №8 Порівняння з традиційними підходами, висновки.

6. Консультанти розділів роботи*

Розділ	Прізвище, ініціали та посада консультанта	Підпис, дата	
		завдання видав	завдання прийняв
РОЗДІЛ 3 РОЗРОБКА МЕТОДУ ПІДВИЩЕННЯ ЕФЕКТИВНОСТІ МЕРЕЖЕВИХ КОМУНІКАЦІЙ	Маньківський Володимир Броніславович	+, 17.05.2025	+, 26.05.2025
РОЗДІЛ 4 ЕКСПЕРИМЕНТАЛЬНЕ ДОСЛІДЖЕННЯ ТА ОЦІНКА ЕФЕКТИВНОСТІ	Маньківський Володимир Броніславович	+, 18.05.2025	+, 27.05.2025

7. Дата видачі завдання 15.09.2024р.

Календарний план

№ з/п	Назва етапів виконання дипломної роботи	Термін виконання етапів роботи	Примітка
1	Ознайомлення з темою, збір та аналіз літератури	15.09.2024-30.09.2024	Виконано
2	Вивчення традиційних методів обміну повідомленнями в розподілених системах	30.09.2024-28.10.2024	Виконано
3	Обґрунтування вибору Apache Kafka та її архітектурних переваг	28.10.2024-30.11.2024	Виконано
4	Розробка тестового середовища на базі Docker і налаштування сервісів Kafka	30.11.2024-15.01.2025	Виконано
5	Реалізація продюсера та консьюмерів, створення тем, підключення бази даних	15.01.2025-25.02.2025	Виконано

* Якщо визначені консультанти. Консультантом не може бути зазначено керівника дипломної роботи.

6	Проведення експериментів із різними конфігураціями Kafka та збір метрик	25.02.2025-10.04.2025	Виконано
7	Теоретичне обґрунтування результатів за допомогою моделі M/M/c	10.04.2025-25.05.2025	Виконано
8	Підготовка та оформлення роботи для захисту	25.05.2025-09.06.2025	Виконано

Студент Данил ГРИЦЕНКО

Керівник Іван САЙЧЕНКО

РЕФЕРАТ

Дипломна робота викладена на 73 сторінках та включає 16 ілюстрацій, 3 таблиці, 12 джерел та 1 додаток.

Мета роботи: дослідити ефективність використання Apache Kafka як інструменту для асинхронної комунікації в розподілених інформаційних системах, а також розробити метод адаптивного налаштування параметрів Kafka для підвищення продуктивності, зниження затримок та уникнення втрат повідомлень.

У роботі реалізовано тестове середовище за допомогою Docker, що включає брокер Kafka, координатор Zookeeper, базу даних PostgreSQL, мікросервіси-продюсери та консьюмери, а також інструменти моніторингу Prometheus і Grafana. Проведено серію експериментів для оцінки впливу кількості партицій, консьюмерів і розміру буфера продюсера на пропускну здатність системи та стабільність доставки повідомлень. Отримані результати проаналізовано за допомогою математичної моделі масового обслуговування M/M/c, що дозволило теоретично підтвердити ефективність запропонованого методу.

Запропонований підхід може бути використаний у реальних програмних системах для підвищення ефективності обробки подій у реальному часі, зокрема в мікросервісних архітектурах, де важлива стабільна передача даних між незалежними компонентами.

Ключові слова: Apache Kafka, асинхронна комунікація, мікросервіси, обробка подій, Docker, буфер, партиція, M/M/c, пропускна здатність, затримка, продюсер, консьюмер.

ABSTRACT

The work is set out on 73 pages and includes 16 illustrations, 3 tables, 12 sources and 1 appendix.

Objective: to explore the efficiency of using Apache Kafka as a tool for asynchronous communication in distributed systems and to develop a method for adaptively configuring Kafka parameters to improve performance, reduce latency, and avoid message loss.

The work includes the implementation of a test environment using Docker, consisting of Kafka broker, Zookeeper coordinator, PostgreSQL database, producer and consumer microservices, and monitoring tools such as Prometheus and Grafana. A series of experiments were conducted to measure the impact of partition count, number of consumers, and producer buffer size on system throughput and message delivery stability. The results were analyzed using the M/M/c queuing model, which confirmed the efficiency of the proposed method from a theoretical perspective.

The proposed solution can be applied in real-world software systems to increase the efficiency of real-time event processing, particularly in microservice architectures where stable data exchange between independent components is crucial.

Keywords: Apache Kafka, asynchronous communication, microservices, event processing, Docker, buffer, partition, M/M/c, throughput, latency, producer, consumer.

ЗМІСТ

ПЕРЕЛІК СКОРОЧЕНЬ

ACK	Acknowledgement (підтвердження доставки)
API	Application Programming Interface (інтерфейс прикладного програмування)
CSV	Comma-Separated Values (формат табличних даних)
DB	Database (База даних)
Grafana	система візуалізації метрик
HTTP	Hypertext Transfer Protocol (протокол передавання гіпертексту)
IoT	Internet of Things (Інтернет речей)
Kafka	Apache Kafka (розподілена платформа обміну повідомленнями)
M/M/c	математична модель масового обслуговування з необмеженою чергою та відсутністю втрат
POM	Project Object Model (структура проєкту у Maven)
PostgreSQL	система керування базами даних
Prometheus	система моніторингу метрик
REST	Representational State Transfer (архітектурний стиль вебсервісів)
RPC	Remote Procedure Call (віддалений виклик процедур)
SDN	Software-Defined Networking (програмно визначені мережі)
TTL	час життя ключів
UI	User Interface (користувацький інтерфейс)
URL	Uniform Resource Locator (уніфікований локатор ресурсу)
VS Code	Visual Studio Code (редактор коду)
ZK	Apache Zookeeper (система координації Kafka-кластерів)

ВСТУП

У сучасних розподілених інформаційних системах критичну роль відіграє ефективна взаємодія між їх компонентами. Зі зростанням обсягів переданих даних, підвищенням вимог до продуктивності та масштабованості, особливо в умовах мікросервісної архітектури, актуальним стає застосування надійних систем обміну повідомленнями.

Традиційні підходи, такі як REST або RPC [5], не завжди забезпечують необхідну гнучкість та стійкість до збоїв, особливо при високому навантаженні. У таких умовах все більшого поширення набувають брокери повідомлень, зокрема Apache Kafka, яка завдяки своїй архітектурі дозволяє обробляти мільйони подій у реальному часі, зберігати історію повідомлень, масштабуватись горизонтально та забезпечувати надійність доставки.

Однак для досягнення максимальної ефективності Kafka потребує коректного налаштування. Продуктивність системи залежить від кількості партицій, консьюмерів, параметрів буфера та інших характеристик. Враховуючи це, у дипломній роботі запропоновано метод підвищення ефективності комунікацій в розподілених системах шляхом поєднання можливостей Kafka та математичного моделювання.

Метою дослідження є розробка методу оптимізації передачі повідомлень у середовищі Apache Kafka, що дозволяє зменшити затримки, уникнути втрат даних та забезпечити стабільну роботу системи при різному навантаженні. Основою теоретичного обґрунтування виступає модель масового обслуговування типу M/M/c (модель Ерланга-С), яка дозволяє розраховувати ймовірність черги, середній час очікування та інші характеристики.

Для досягнення мети було побудовано тестове середовище на основі Docker, розгорнуто брокер Kafka, консьюмери і продюсери [6], а також підключено інструменти моніторингу: Prometheus та Grafana [7]. Проведено

серію експериментів у різних конфігураціях (1-2 партиції, 1-2 консьюмери, зміна розміру буфера), що дозволило емпірично оцінити продуктивність та порівняти її з теоретичними розрахунками.

Актуальність теми зумовлена потребою в гнучких, адаптивних і математично обґрунтованих підходах до побудови мережевих комунікацій у сучасних цифрових системах. Отримані результати можуть бути використані при розробці надійних потокових сервісів, побудові черг обробки подій у реальному часі, а також для забезпечення безперебійної взаємодії між мікросервісами в умовах динамічного навантаження.

РОЗДІЛ 1
ТЕОРЕТИЧНІ ОСНОВИ МЕРЕЖЕВИХ КОМУНІКАЦІЙ У РОЗПОДІЛЕНИХ СИСТЕМАХ

1.1 Основи розподілених систем та їх архітектура

Розподілені системи — це сукупність автономних комп'ютерних компонентів, що пов'язані між собою мережею та координують свої дії з метою досягнення спільної мети. На відміну від централізованих систем, де всі ресурси та обробка даних зосереджені в одному вузлі, розподілені системи використовують множину обчислювальних одиниць, розташованих у різних фізичних або логічних точках. Вони функціонують як єдине ціле, забезпечуючи при цьому високу доступність, гнучкість масштабування та стійкість до відмов [11].

З практичної точки зору, розподілені системи є основою для сучасних хмарних сервісів, обчислень на периферії (edge computing), мікросервісних архітектур, систем з великими даними, телекомунікаційних платформ та IoT-рішень. Такі системи використовуються у фінансовій сфері, в охороні здоров'я, логістиці, онлайн-торгівлі, кібербезпеці та в багатьох інших галузях, де необхідна обробка великих обсягів даних у режимі реального часу.

Однією з ключових переваг розподілених систем є масштабованість. Завдяки можливості додавати нові вузли без зупинки системи, можна легко адаптувати її до зростання навантаження. Іншою важливою характеристикою є відмовостійкість — здатність системи продовжувати роботу навіть у випадку виходу з ладу окремих її компонентів. Завдяки розподіленому зберіганню та дублюванню даних система може автоматично перенаправити запити або відновити інформацію з резервних джерел.

Загальна архітектура розподілених систем, як правило, базується на кількох рівнях. Найпоширенішими є:

- Клієнт-серверна архітектура, де клієнти ініціюють запити, а сервери їх обробляють. У розподіленому варіанті сервери можуть бути дубльовані або географічно розподілені.
- Peer-to-peer (P2P) — система без єдиного центру, у якій кожен вузол може виступати і клієнтом, і сервером. Такі системи більш стійкі до точок відмови, але складніші в управлінні.
- Мікросервісна архітектура — сучасний підхід, у якому кожен окремий сервіс виконує одну функцію, комунікуючи з іншими через мережу. Цей підхід дозволяє легко масштабувати, оновлювати й розгортати окремі частини системи без впливу на решту компонентів.

Особливої уваги в архітектурі розподілених систем заслуговують протоколи взаємодії між вузлами. Основні виклики тут — забезпечення узгодженості станів, безпеки передачі даних, мінімізації затримок і оптимізації пропускної здатності. Для реалізації цих вимог у системах широко використовуються спеціалізовані засоби обміну повідомленнями, що дозволяють синхронізувати дані та події між різними частинами інфраструктури.

Успішне функціонування розподіленої системи залежить від вибору ефективної механіки комунікації, яка дозволяє своєчасно доставляти інформацію між компонентами. Саме в цьому контексті постає потреба у використанні спеціалізованих платформ, здатних забезпечити передачу повідомлень у реальному часі, гарантувати доставку, працювати зі збоєм окремих вузлів та масштабуватись із мінімальними витратами ресурсів. Apache Kafka є прикладом такої технології.

1.2. Проблеми та виклики мережевої взаємодії (затримки, втрата пакетів, навантаження)

Незважаючи на численні переваги розподілених систем, їх ефективне функціонування безпосередньо залежить від стабільності та швидкодії мережевих комунікацій. Саме мережевий рівень взаємодії найчастіше стає критичним у ситуаціях, коли потрібно обробляти великий обсяг даних або забезпечити гарантовану доставку повідомлень між віддаленими компонентами. У таких умовах з’являється низка викликів, які потребують спеціального підходу до проектування, моніторингу та оптимізації системи.

Однією з найважливіших проблем є затримки (latency) у доставці повідомлень. У реальних мережах передача даних між вузлами може супроводжуватися непередбачуваними затримками, спричиненими перевантаженням каналів, маршрутизацією через проміжні вузли, несправністю окремих сегментів мережі або особливостями протоколів передачі. У критичних сценаріях, таких як біржова торгівля, моніторинг виробництва або телемедицина, навіть мілісекундна затримка може мати відчутні наслідки. Тому системи, що використовуються для обміну повідомленнями, повинні забезпечувати не лише низький середній час доставки, але й стабільність цього показника.

Ще одним поширеним викликом є втрата пакетів (packet loss). У нестабільних або перевантажених мережах окремі пакети можуть не доходити до отримувача, що призводить до неповного або викривленого отримання повідомлення. Якщо система обміну повідомленнями не має вбудованих механізмів повторної доставки або валідації, це може призвести до втрати критично важливих даних. У разі використання протоколу UDP, який не гарантує доставку, або за відсутності підтверджень доставки з боку одержувача, ця проблема стає особливо гострою.

Наступним фактором є нерівномірність навантаження (load imbalance), коли окремі компоненти системи отримують значно більше запитів, ніж інші. Це може призводити до локального перевантаження ресурсів, блокувань черг, зростання затримок або навіть до відмови сервісів. У мікросервісних архітектурах така ситуація часто виникає через неправильну маршрутизацію запитів або відсутність механізмів балансування навантаження. Розподілені черги повідомлень, такі як Kafka, можуть частково вирішити цю проблему за рахунок поділу потоків даних на партиції та гнучкого розподілу консьюмерів.

Окремо слід згадати і про проблему узгодженості даних (consistency). У розподілених системах часто виникають ситуації, коли одні й ті самі дані опрацьовуються у різний час різними компонентами, що може призводити до конфліктів або дублювання. Це особливо критично у випадках обробки транзакцій, подій у системах реального часу або паралельного оновлення даних з кількох джерел.

Крім того, не менш актуальним є питання масштабованості. Зі збільшенням навантаження на систему (кількості клієнтів, обсягу повідомлень, частоти подій) необхідно забезпечити таку архітектуру, яка дозволить додавати нові вузли або розширювати ресурси без зупинки системи. Невдалий вибір технології або архітектурного рішення часто призводить до того, що система не здатна справлятися з ростом трафіку, навіть за наявності вільних апаратних ресурсів.

Вирішення вищезазначених проблем вимагає комплексного підходу: від вибору відповідного протоколу обміну повідомленнями та побудови черг, до впровадження розумних стратегій масштабування, кешування та балансування навантаження. У цьому контексті Apache Kafka виступає як платформа, що враховує багато з цих викликів уже на рівні своєї архітектури. Зокрема, вона дозволяє зменшити затримки за рахунок асинхронного обміну, уникнути втрат повідомлень завдяки збереженню в логах, та ефективно розподіляти навантаження між консьюмерами в межах consumer groups.

1.3 Огляд технологій обміну повідомленнями

Надійна та швидка передача даних між компонентами розподілених систем є критично важливою для забезпечення стабільної роботи сучасних цифрових сервісів. Упродовж останніх десятиліть було розроблено чимало технологій та підходів до організації обміну повідомленнями. Кожен з них має свої переваги, обмеження та оптимальні сфери застосування. У цьому підпункті розглянемо три основні класи технологій: традиційні методи комунікації, класичні системи черг повідомлень та сучасні платформи, такі як Apache Kafka.

1.3.1 Традиційні підходи (REST, RPC, WebSockets)

Одним з найбільш поширених способів комунікації між сервісами є використання протоколу HTTP у форматі REST (Representational State Transfer). У цьому підході кожен запит клієнта супроводжується окремим HTTP-зверненням до сервера, яке обробляється в синхронному режимі. REST добре підходить для операцій, що не вимагають обробки в реальному часі, наприклад, реєстрація користувачів, отримання даних з бази або керування станами об’єктів [5].

Однак REST має суттєві обмеження. По-перше, кожен запит створює окреме з’єднання, що призводить до зростання навантаження на сервер і збільшує затримку. По-друге, REST не підтримує двосторонній обмін — клієнт повинен щоразу ініціювати запит. Це ускладнює реалізацію механізмів обробки подій, які надходять незалежно від дій користувача.

Ще одним варіантом є RPC (Remote Procedure Call) — протокол, який дозволяє викликати функції на віддалених машинах так, ніби вони є локальними. RPC передбачає більш щільну зв’язаність компонентів, що зменшує гнучкість, але може бути корисним у високопродуктивних внутрішніх сервісах.

WebSockets — це більш сучасна технологія, яка дозволяє встановити постійне двостороннє з’єднання між клієнтом і сервером. Це з’єднання дає змогу передавати повідомлення в обох напрямках у режимі реального часу без необхідності повторного відкриття каналу зв’язку. WebSockets є зручними для розробки чатів, біржових терміналів або систем моніторингу. Проте, їх реалізація у складних інфраструктурах потребує додаткових механізмів маршрутизації, масштабування та відновлення з’єднань, що ускладнює підтримку великих систем.

1.3.2 Черги повідомлень (RabbitMQ, ActiveMQ, ZeroMQ)

Для забезпечення асинхронного обміну повідомленнями у розподілених системах широко використовуються так звані message brokers — посередники, які отримують повідомлення від відправника (producer) і передають їх до одержувача (consumer). Основна ідея полягає у тому, що компоненти системи можуть працювати незалежно один від одного: якщо отримувач тимчасово недоступний, повідомлення зберігається в черзі та буде доставлене пізніше [3].

Серед найпопулярніших брокерів повідомлень можна виділити:

- RabbitMQ — система з відкритим кодом, що реалізує протокол AMQP. Вона дозволяє створювати складні маршрути доставки, має гнучку систему черг і підтвердження доставки. RabbitMQ добре працює з невеликими навантаженнями і в мікросервісних архітектурах.
- ActiveMQ — ще один популярний брокер від Apache, що підтримує багато протоколів (включаючи STOMP, MQTT, OpenWire). Підходить для інтеграції з Java EE-додатками, підтримує транзакції та повідомлення з пріоритетами.
- ZeroMQ — легковажна бібліотека, яка надає можливість створювати високошвидкісні peer-to-peer системи передачі повідомлень. Вона не є брокером у класичному сенсі, оскільки не має централізованого посередника, але широко використовується там, де потрібна мінімальна затримка.

Недоліком традиційних брокерів є обмежена масштабованість у горизонтальному напрямку та складність у роботі з великим обсягом повідомлень у реальному часі. Саме ці обмеження спонукали до розробки нових платформ, зокрема Apache Kafka.

1.3.3 Роль Apache Kafka у сучасних системах

Apache Kafka — це розподілена платформа обміну повідомленнями, яка була спочатку розроблена в LinkedIn для обробки подій у реальному часі. Її архітектура була створена з урахуванням потреб високонавантажених систем. Kafka підтримує зберігання потоків подій, розподілення навантаження між численними консьюмерами та роботу у кластері з кількох брокерів [1, 2].

Kafka реалізує модель publish-subscribe: продюсери надсилають повідомлення до певних topic, а консьюмери підписуються на них і отримують дані. Ключовою особливістю Kafka є те, що повідомлення зберігаються в партиційованих логах. Кожен споживач самостійно веде облік зчитаних повідомлень за допомогою offset — це дозволяє повертатися до обробки з попереднього місця у разі збою, а також зменшує навантаження на систему доставки.

Kafka відрізняється високою масштабованістю: нові брокери або консьюмери можуть підключатися до системи без зупинки сервісу. Повідомлення в Kafka зберігаються не в оперативній пам’яті, а на диску, що дозволяє працювати з великими обсягами історичних даних без втрати продуктивності [8].

Ще однією перевагою Kafka є гнучка інтеграція з іншими технологіями. Існують модулі Kafka Streams, Kafka Connect, а також готові коннектори для інтеграції з базами даних, файловими системами, аналітичними платформами (Spark, Flink), що дозволяє будувати повноцінні потокові пайплайни обробки даних.

1.3.4 Принципи роботи Apache Kafka (Publish-Subscribe модель, лог сегментації)

Apache Kafka функціонує як розподілена платформа для обміну повідомленнями, у центрі якої лежить модель публікації-підписки (publish-subscribe). Цей принцип дозволяє системі ефективно забезпечувати обмін подіями між численними незалежними компонентами — від мікросервісів до зовнішніх аналітичних систем — без потреби в жорсткому зв'язуванні їх між собою. Таке роз'єднання компонентів підвищує масштабованість, надійність і гнучкість всієї архітектури.

У моделі publish-subscribe є три основні ролі:

- Producer (видавець) — додаток або сервіс, який генерує повідомлення та надсилає їх у певний топік (topic).
- Broker — сервер Kafka, який приймає, зберігає та передає повідомлення. Один кластер Kafka може містити кілька брокерів для горизонтального масштабування.
- Consumer (підписник) — додаток або сервіс, який підписується на певні топіки й отримує повідомлення, обробляючи їх у потрібний спосіб.

Процес комунікації в Kafka виглядає наступним чином: продюсери публікують повідомлення у топіки, брокери зберігають ці повідомлення в логах, а консьюмери, що підписані на відповідні топіки, читають та обробляють повідомлення у своєму темпі. Завдяки цьому система досягає асинхронності, тобто компоненти не зобов'язані працювати одночасно — кожен з них може обробляти дані незалежно у своєму темпі.

Ключовою особливістю Kafka є логічна структура зберігання повідомлень. Кожен топік ділиться на партиції — впорядковані послідовності записів (логи), кожен з яких має унікальний номер — offset. Offset визначає позицію повідомлення в межах конкретної партиції. Консьюмер, зчитуючи повідомлення, самостійно відстежує offset, що дає змогу:

- зберігати прогрес обробки даних (наприклад, після збою повернутись до останнього успішно прочитаного повідомлення);

- реалізовувати паралельну обробку — кілька консьюмерів можуть читати з різних партицій одного топіка;
- уникати дублювання або втрати даних у разі повторного зчитування.

Для забезпечення високої продуктивності Kafka реалізує лог-сегментацію. Повідомлення в партиціях зберігаються у вигляді файлів — логів, які поділяються на сегменти. Кожен сегмент є окремим файлом на диску, який автоматично закривається при досягненні певного розміру або часу. Завдяки такому підходу Kafka:

- забезпечує постійну швидкість запису, оскільки нові повідомлення додаються лише в кінець останнього сегмента;
- може зберігати великі обсяги історичних даних без втрати продуктивності;
- дає можливість швидко видаляти старі повідомлення, не змінюючи структуру логів (наприклад, через політику retention).

Кожен брокер Kafka може обслуговувати кілька партицій та зберігати копії (репліки) партицій з інших брокерів. Таким чином забезпечується відмовостійкість: якщо один брокер виходить з ладу, його партиції можуть бути оброблені іншими брокерами, що мають актуальні репліки.

Ще однією важливою перевагою є гнучка конфігурація доставки повідомлень. Kafka дозволяє налаштовувати:

- acknowledgment (підтвердження) — чи повинен продюсер чекати підтвердження доставки повідомлення на всі репліки;
- delivery semantics — гарантії доставки (at-most-once, at-least-once або exactly-once);
- retention policy — політику зберігання повідомлень (за часом або за розміром).

Завдяки поєднанню моделі publish-subscribe, лог-сегментації та масштабованої партиційної архітектури Apache Kafka стає надзвичайно

ефективним інструментом для побудови систем обробки подій у реальному часі, що працюють із великим навантаженням.

1.3.5 Порівняння Apache Kafka з іншими брокерами повідомлень

На ринку систем обміну повідомленнями існує чимало рішень, кожне з яких орієнтоване на певні типи навантаження, архітектурні підходи та вимоги до надійності. Серед найпоширеніших брокерів повідомлень, крім Apache Kafka, варто згадати RabbitMQ, ActiveMQ та ZeroMQ. Щоб повною мірою зрозуміти переваги та недоліки Kafka, доцільно провести систематизоване порівняння за низкою ключових параметрів.

1. Архітектура та модель взаємодії

Таблиця 1.1

Різниця характеристик різних брокерів повідомлень

Характеристика	**Apache Kafka**	**RabbitMQ/ActiveMQ**	**ZeroMQ**
Модель передачі	Publish-Subscribe (логічна черга)	Черги (queue-based, fanout)	Peer-to-peer
Сховище	Дискове лог-зберігання з сегментацією	У пам’яті з опційним дисковим записом	Без збереження (in-memory)
Затримка обробки	Низька (для великих обсягів)	Низька–середня	Мінімальна (але нестійка)
Підтримка історії	Повна, до заданого періоду	Часткова	Відсутня
Масштабування	Горизонтальне (через партиції)	Вертикальне / обмежене горизонтальне	Вручну через логіку програми

2. Продуктивність і навантаження

Kafka створена для високонавантажених середовищ, де обсяг переданих повідомлень може сягати мільйонів на секунду. Її архітектура оптимізована

для великого throughput (пропускної здатності) при низьких затратах на апаратні ресурси завдяки запису даних у журнали та зчитуванню по offset-ах.

RabbitMQ і ActiveMQ добре працюють у невеликих або середніх системах, особливо там, де важлива гнучкість маршрутизації повідомлень. Проте при зростанні обсягу даних вони швидко вичерпують свої ресурси й потребують складних підходів до масштабування.

ZeroMQ вирізняється максимальною швидкістю, оскільки не має брокера як такого. Але ця перевага нівелюється тим, що відсутнє зберігання повідомлень і будь-який збій у мережі веде до втрат. ZeroMQ потребує значно більших зусиль для створення стійкої інфраструктури.

3. Надійність і гарантії доставки

Kafka має гнучкі механізми гарантування доставки повідомлень: можливість вибору між доставкою "принаймні один раз" (at-least-once), "не більше одного разу" (at-most-once) або "рівно один раз" (exactly-once). Це дозволяє адаптувати систему до потреб конкретного бізнес-процесу.

RabbitMQ теж підтримує підтвердження доставки, проте залежить від використаної черги та конфігурацій. Активне використання транзакцій і підтверджень може знижувати продуктивність. ActiveMQ подібний до RabbitMQ, але орієнтований переважно на Java-середовище.

ZeroMQ практично не має гарантій доставки, покладаючись на логіку в коді користувача. Це робить його дуже швидким, але водночас — ризикованим для критичних систем.

4. Гнучкість інтеграції та екосистема

Kafka має потужну екосистему: Kafka Streams для обробки потоку даних, Kafka Connect для підключення джерел/сховищ, численні клієнтські бібліотеки для Java, Python, Go, Node.js тощо. Її активно інтегрують з Apache Flink, Spark, Hadoop, що робить Kafka основою для аналітичних і ETL-систем.

RabbitMQ має широке API і численні плагіни, але поступається Kafka в продуктивності та масштабованості. ActiveMQ добре інтегрується з Java EE, але менш активний у підтримці сучасних стеків.

ZeroMQ — це радше транспортна бібліотека, ніж повноцінний брокер, тому інтеграція повністю залежить від розробника і не має централізованої підтримки або інструментів моніторингу.

1.3.6 Загальні висновки порівняння

Apache Kafka вирізняється високою масштабованістю, стійкістю до збоїв та здатністю працювати з великими потоками повідомлень у реальному часі. Вона особливо ефективна для задач, де важливо зберігати історію подій, працювати з великим навантаженням і забезпечувати точне відтворення черги у разі збоїв.

У свою чергу, RabbitMQ або ActiveMQ підходять для систем, де потрібна гнучка маршрутизація повідомлень, але навантаження не надто високе. ZeroMQ — це вузькоспеціалізований інструмент для ситуацій, де швидкість важливіша за гарантії.

Таким чином, вибір між Kafka та іншими брокерами повідомлень залежить від цілей системи: для обробки подій у реальному часі, побудови потокових архітектур, аналітики — Kafka є безумовно кращим рішенням.

Висновки

У першому розділі було проведено огляд основних понять і викликів, пов’язаних із мережевими комунікаціями в розподілених системах. Проаналізовано традиційні підходи до передачі повідомлень (REST, RPC, WebSockets) [5] та їх обмеження при високих навантаженнях. Також було

розглянуто типові черги повідомлень, зокрема RabbitMQ, ActiveMQ і ZeroMQ, [3, 11] та виявлено їх недоліки у масштабованості й збереженні історії подій.

Особливу увагу приділено Apache Kafka [9] як сучасному брокеру повідомлень, орієнтованому на обробку потоків даних у реальному часі. Було розкрито її архітектуру, механізм роботи партицій, лог-сегментації та управління offset-ами. Порівняльний аналіз Kafka з іншими брокерами показав її переваги в продуктивності, надійності та можливостях горизонтального масштабування.

Загалом, розділ сформував теоретичну базу для подальшого дослідження — підтверджуючи, що Apache Kafka є оптимальним інструментом для реалізації асинхронної комунікації в розподілених системах з високим навантаженням.

РОЗДІЛ 2
АНАЛІЗ ЕФЕКТИВНОСТІ APACHE KAFKA У РОЗПОДІЛЕНИХ СИСТЕМАХ

2.1. Архітектурні особливості Kafka у високонавантажених системах

Apache Kafka була створена з прицілом на роботу у високонавантажених середовищах, де обробка мільйонів подій на секунду є звичайним завданням. Її архітектура ґрунтується на принципах горизонтального масштабування, стійкості до відмов і ефективної обробки потоків даних. Такі особливості дають змогу Kafka залишатись однією з найпродуктивніших і надійніших систем для потокової передачі повідомлень.

У центрі архітектури Kafka лежить кластер брокерів — це сукупність серверів, які виконують роль зберігання та пересилання повідомлень. Кожен брокер може працювати автономно, але у кластері брокери координуються для розподілу даних, балансування навантаження та забезпечення відмовостійкості.

Ключовими особливостями Kafka в умовах високого навантаження є:

1. Партиційна модель даних

Кожен топік Kafka (логічний канал, у який надсилаються повідомлення) поділяється на партиції — незалежні впорядковані лінійні журнали. Партиції розподіляються між брокерами, що дає змогу:

- Паралелізувати обробку повідомлень — кожна партиція може обслуговуватись окремим консьюмером;
- Горизонтально масштабувати систему — із зростанням навантаження можна додавати нові партиції та брокери;
- Зменшити конфлікти доступу до даних — партиції працюють незалежно одна від одної.

Кількість партицій визначає рівень паралелізму, тому у високонавантажених системах цей параметр ретельно підбирається залежно від кількості консьюмерів і характеру навантаження.

2. Реплікація та стійкість до збоїв

Kafka реалізує реплікацію партицій — кожна партиція має одну головну (лідер) і одну або кілька копій (фолловери), які зберігаються на інших брокерах [4]. Це дозволяє:

- Продовжити роботу у разі відмови лідера — фолловер автоматично переходить у стан лідера;
- Забезпечити високу доступність даних — навіть у випадку збою частини кластера.

Механізм вибору лідера забезпечується через Zookeeper або, починаючи з Kafka 2.8, через внутрішній контролер (KRaft mode), що зменшує залежність від зовнішніх систем.

3. Зберігання даних у логах на диску

На відміну від багатьох черг повідомлень, Kafka не покладається на оперативну пам’ять. Всі повідомлення зберігаються на диску у вигляді лог-файлів, що розбиті на сегменти. Завдяки цьому:

- Kafka здатна працювати з терабайтами даних без деградації продуктивності;
- Дані можна повторно читати, що корисно для аналітики, відлагодження або відновлення після збою;
- Система може бути і чергою, і сховищем подій (event store).

Kafka оптимізована для послідовного запису на диск, який значно швидший за довільний доступ, тому навіть при роботі з HDD система показує високу продуктивність.

4. Власна система збереження offset-ів

Консьюмери у Kafka самостійно зберігають інформацію про те, які повідомлення були прочитані (offset). Це робиться або в Kafka (у внутрішньому топіку __consumer_offsets), або у зовнішньому сховищі, наприклад у Zookeeper чи базі даних.

Переваги такого підходу:

- Гнучкість: кожен консьюмер контролює, звідки починати обробку — з початку, з певної точки, або лише нові повідомлення;
- Відновлення після збоїв: у разі аварії консьюмер може відновити обробку з останнього offset-а;
- Контроль обробки: можна побудувати системи, де події обробляються з затримкою або повторно, для перевірки або аналітики.

5. Асинхронна взаємодія та високий throughput

Kafka працює в асинхронному режимі, що дозволяє відокремити продюсерів і консьюмерів — вони не зобов’язані бути онлайн одночасно. Це дозволяє:

- Обробляти пікові навантаження — повідомлення накопичуються в Kafka і читаються, коли з’явиться вільний ресурс;
- Підвищити стабільність системи — відмова одного компонента не призводить до втрати даних або зупинки всієї системи.

Усе це забезпечує високу пропускну здатність Kafka, яка може обробляти мільйони повідомлень на секунду навіть у кластері середнього розміру.

Таким чином, архітектура Apache Kafka є результатом глибокого інженерного опрацювання проблем, що виникають у масштабованих та навантажених середовищах. Її дизайн дозволяє досягти балансу між швидкодією, гнучкістю і надійністю, що робить її ідеальним вибором для побудови систем обміну повідомленнями нового покоління.

2.2 Масштабованість та відмовостійкість Apache Kafka

Однією з ключових переваг Apache Kafka, що відрізняє її від традиційних систем обміну повідомленнями, є здатність ефективно масштабуватись та зберігати стабільність роботи навіть в умовах часткових збоїв компонентів системи. Ці властивості роблять Kafka ідеальним інструментом для критично важливих розподілених систем, де недоступність або втрата повідомлень є неприпустимою.

2.2.1 Масштабованість Kafka

Масштабованість — це здатність системи зберігати продуктивність або навіть покращувати її при зростанні навантаження шляхом додавання обчислювальних ресурсів [8]. У Kafka ця властивість досягається завдяки кільком взаємодоповнюючим механізмам.

1. Партиціювання топіків

Кожен топік Kafka ділиться на партиції, які можуть бути розподілені між різними брокерами. Додавання нових партицій дозволяє збільшити кількість паралельно оброблюваних потоків даних. У такому випадку:

- Продюсери можуть писати у декілька партицій одночасно;
- Кожен консьюмер у межах групи може обслуговувати одну або кілька партицій;
- Зростає загальна пропускна здатність системи.

Цей підхід забезпечує лінійне масштабування: що більше партицій — то більше вузлів можуть одночасно писати або читати дані, не блокуючи один одного.

2. Горизонтальне масштабування брокерів

Kafka дозволяє горизонтально масштабувати кластер брокерів — тобто додавати нові сервери без зупинки системи. Після додавання нового брокера партиції можуть бути перенесені на нього або створені нові, що зменшить навантаження на вже існуючі вузли.

- Це забезпечується автоматизованими процесами rebalance і partition reassignment;
- Завдяки внутрішній координації (через Zookeeper або Kafka Controller), Kafka може перенаправляти потоки даних без потреби втручання користувача.

3. Масштабування на рівні споживачів

Consumer group — це група консьюмерів, які разом читають повідомлення з топіка. Kafka автоматично розподіляє партиції між учасниками групи. Додавання нового консьюмера до групи збільшує обробну здатність системи без дублювання обробки повідомлень.

Завдяки цьому Kafka дуже ефективна у мікросервісних архітектурах, де кожен сервіс може масштабуватись незалежно, просто створюючи новий екземпляр, який приєднується до відповідної consumer group.

2.2.2 Відмовостійкість Kafka

Відмовостійкість — це здатність системи зберігати працездатність або самовідновлюватись після збоїв апаратного чи програмного характеру. Kafka має вбудовані механізми для забезпечення високого рівня надійності:

1. Реплікація партицій

Кожна партиція у Kafka має репліки, які зберігаються на різних брокерах [4]. Одну з них призначено лідером, інші — фолловерами. Всі записи здійснюються спочатку у лідер-партицію, а потім копіюються до фолловерів.

У разі виходу лідера з ладу:

- Один із фолловерів призначається новим лідером;
- Kafka продовжує обробку подій без втрати даних.

Цей механізм гарантує високу доступність навіть при відмові окремих вузлів.

2. Підтвердження доставки (ACK)

Продюсери можуть налаштовувати рівень підтвердження доставки:

- acks=0 — повідомлення вважається доставленим одразу після надсилання (швидко, але ненадійно);
- acks=1 — підтвердження лише від лідера;
- acks=all — підтвердження після запису на всі репліки (максимальна надійність).

Це дозволяє балансувати між швидкістю і надійністю залежно від сценарію використання.

3. Збереження offset-ів і відновлення після збоїв

Kafka дозволяє консьюмерам самостійно відстежувати прочитані повідомлення. Якщо обробка переривається, споживач може повторно прочитати повідомлення, починаючи з останнього збереженого offset-а. Завдяки цьому забезпечується:

- Точність обробки (наприклад, при фінансових транзакціях);
- Можливість повторної обробки (при зміні логіки або помилках).

4. Самообслуговування кластера

Kafka здатна автоматично балансувати навантаження, обирати нових лідерів, переспрямовувати потоки даних без зупинки системи. Це зменшує потребу в ручному адмініструванні та знижує ризик людських помилок.

Таким чином, Apache Kafka надає потужні засоби як для масштабування вшир, так і для забезпечення високої надійності. Її архітектурна гнучкість

дозволяє адаптувати платформу під різноманітні сценарії — від локальних систем моніторингу до глобальних аналітичних платформ, що обробляють мільйони подій щосекунди.

2.3. Конфігураційні параметри, що впливають на продуктивність Kafka

Apache Kafka надає широкий набір конфігураційних параметрів, які дозволяють точно налаштовувати поведінку системи відповідно до вимог конкретного середовища. Успішне масштабування та забезпечення високої продуктивності Kafka неможливе без глибокого розуміння цих параметрів. Від них залежить швидкість доставки повідомлень, стійкість до збоїв, затримки, використання ресурсів та здатність до обробки великих обсягів подій у реальному часі.

1. Кількість партицій (num.partitions)

Цей параметр визначає кількість партицій, на які розбивається кожен топік. Партиції є основною одиницею паралелізму в Kafka. Чим більше партицій:

- тим більше консьюмерів можуть одночасно обробляти дані;
- тим вища загальна пропускна здатність системи;
- проте зростає навантаження на контролюючі компоненти та потреба у пам’яті.

Оптимальна кількість партицій залежить від кількості консьюмерів у системі, типу навантаження та доступних апаратних ресурсів. Занадто мала кількість обмежує масштабування, а надто велика — може погіршити стабільність системи.

2. Розмір буфера (batch.size, buffer.memory)

batch.size (у байтах) визначає обсяг даних, який продюсер відправляє у Kafka за один раз. Чим більший batch, тим менше мережевих звернень, що підвищує ефективність. Але при надто великому значенні зростає затримка відправлення.

buffer.memory — загальний розмір буфера в пам’яті, який продюсер використовує для зберігання записів перед відправкою. Якщо буфер заповнено, відправлення буде заблоковано. Недостатній об’єм буфера може призводити до втрати повідомлень при піковому навантаженні.

3. Затримка надсилання (linger.ms)

Цей параметр визначає, наскільки довго продюсер чекатиме, перш ніж відправити batch, навіть якщо він не заповнений. Наприклад, linger.ms = 10 означає, що продюсер зачекає до 10 мс, перш ніж відправити накопичені повідомлення. Це дозволяє зібрати більші пакети та зменшити кількість мережевих запитів, що корисно для зменшення навантаження на брокери.

Занадто велике значення може призвести до затримок доставки, тому параметр потрібно обирати з урахуванням компромісу між продуктивністю та латентністю.

4. Параметри підтвердження (acks, retries)

acks визначає, скільки брокерів повинні підтвердити отримання повідомлення, перш ніж продюсер вважатиме його доставленим:

- acks=0 — найшвидше, але ненадійно (без підтверджень);
- acks=1 — підтвердження тільки від лідера (середній рівень надійності);
- acks=all — підтвердження від усіх реплік (максимальна надійність, але повільніше).

retries задає кількість спроб повторної відправки повідомлення у разі помилки. Це особливо важливо для гарантії доставки, але потребує контролю, щоб уникнути дублювання повідомлень або перевантаження мережі.

5. Розмір сегмента логів (segment.bytes, segment.ms)

Kafka зберігає повідомлення у вигляді логів, які поділені на сегменти. Параметри segment.bytes та segment.ms керують:

- максимальним розміром сегменту у байтах;
- максимальним часом зберігання повідомлень у одному файлі-сегменті.

Ці налаштування важливі для контролю використання дискового простору, швидкості читання/видалення старих даних та ефективної роботи з історичними подіями.

6. Політика збереження повідомлень (retention.ms, retention.bytes)

Kafka дозволяє налаштувати тривалість зберігання повідомлень у логах за допомогою параметрів:

- retention.ms — максимальний час зберігання;
- retention.bytes — максимальний обсяг, що може бути збережений на диск.

Якщо перевищено один із параметрів, старіші повідомлення автоматично видаляються. Це дає змогу обмежити використання диску, але потребує уважного контролю, щоб не втратити важливі дані.

7. Максимальна кількість коннекцій (num.network.threads, num.io.threads)

Ці параметри визначають кількість потоків у Kafka-брокерах, які обслуговують мережеву активність (network.threads) та вхід/вихід (io.threads). Недостатня кількість потоків призводить до затримок, але надлишок — до перевитрати ресурсів.

У середовищах з високим навантаженням ці параметри є критично важливими для забезпечення рівномірного розподілу навантаження та уникнення "вузьких місць" у системі.

8. Вплив комбінацій параметрів

Варто зазначити, що продуктивність Kafka залежить не від одного параметра, а від комбінації налаштувань. Наприклад:

- Великий batch.size без належного linger.ms може не дати очікуваного ефекту;
- Високе значення acks=all з малим retries не забезпечить гарантовану доставку;
- Мала кількість партицій при великій кількості консьюмерів обмежить масштабованість.

Тому важливо здійснювати ретельний підбір конфігурацій з урахуванням специфіки задачі, а також проводити тестування з реальними сценаріями навантаження.

У результаті можна зробити висновок: грамотна конфігурація Kafka — ключ до її ефективної роботи в реальних умовах. Від розміру batch-ів до політики зберігання — кожен параметр впливає на стабільність, швидкість та надійність системи. Лише через експериментальну перевірку, спираючись на математичні моделі й моніторинг метрик, можна досягти оптимального результату.

2.3. Аналіз механізмів гарантії доставки повідомлень

Забезпечення гарантованої доставки повідомлень є критичним аспектом у побудові надійних розподілених систем. У випадках, коли передані дані мають важливе значення — наприклад, у фінансових транзакціях, логах безпеки чи системах моніторингу, — навіть одноразова втрата повідомлення може призвести до серйозних наслідків. Apache Kafka, як розподілена платформа потокової обробки подій, передбачає кілька рівнів гарантії доставки повідомлень, що дозволяє гнучко адаптувати її під конкретні вимоги до надійності та швидкодії.

Однією з переваг Kafka є можливість вибору між різними моделями доставки повідомлень: at-most-once, at-least-once і exactly-once [11]. Кожна з них орієнтована на різні сценарії використання і має відповідні компроміси між продуктивністю та надійністю.

У режимі at-most-once повідомлення передається з мінімальними затримками, однак у разі помилки в процесі надсилання або збою в системі повідомлення може бути втрачене. Цей варіант застосовується в умовах, коли втрата окремих подій є прийнятною і критичне значення має саме швидкодія. У свою чергу, at-least-once гарантує доставку кожного повідомлення, навіть якщо це призводить до можливого дублювання. Такий режим є типовим для аналітичних систем, логування або ситуацій, де втрати неприпустимі, але дублікати можна обробити на рівні споживача. Найнадійнішим є режим exactly-once, який забезпечує унікальну й однократну доставку, але потребує більш складної транзакційної взаємодії між продюсером, брокером і консьюмером, що підвищує загальну затримку.

Реалізація гарантій доставки в Kafka спирається на кілька важливих компонентів. По-перше, це параметр acks, який визначає, скільки брокерів мають підтвердити отримання повідомлення до того, як продюсер вважатиме його успішно доставленим. Значення acks=0 означає, що повідомлення відправляється без очікування підтвердження, що забезпечує максимальну швидкість, але мінімальну надійність. Значення acks=1 передбачає підтвердження від лідера партиції, а acks=all гарантує підтвердження від усіх синхронних реплік, забезпечуючи найвищу надійність.

Ще одним важливим параметром є retries, який задає кількість повторних спроб відправлення повідомлення у випадку невдачі. Це дозволяє підвищити ймовірність доставки при короткочасних збоях мережі або тимчасовій недоступності брокера. Також параметр enable.idempotence, який увімкнений за замовчуванням у сучасних версіях Kafka, дозволяє уникнути

дублювання повідомлень у разі повторної відправки, тим самим забезпечуючи консистентність навіть у режимі at-least-once.

З боку консьюмерів важливу роль відіграє контроль offset-ів — індексів, які вказують на позицію останнього обробленого повідомлення. Kafka дозволяє автоматичне або ручне збереження offset-ів. У менш критичних сценаріях зручно використовувати автоматичний режим, який фіксує offset-и після певного інтервалу. Проте для високонадійних систем доцільно застосовувати ручне збереження offset-ів лише після успішного завершення обробки повідомлення. Це дозволяє уникнути ситуацій, коли повідомлення вважається прочитаним, але фактично не було оброблене через помилку або збій на стороні споживача.

Крім того, Apache Kafka підтримує транзакційну модель обробки повідомлень, яка дозволяє об’єднати серію записів у топік і комітити їх як єдину транзакцію. Це гарантує, що або всі повідомлення будуть доставлені, або жодне — що критично для сценаріїв, де потрібна атомарність операцій. У поєднанні з транзакційною обробкою Kafka забезпечує реалізацію моделі exactly-once навіть у випадку обробки повідомлень між кількома темами або з інтеграцією в зовнішні системи через Kafka Connect.

Загалом, механізми гарантії доставки повідомлень у Kafka надають широкі можливості для балансування між швидкодією, ресурсними витратами та рівнем надійності. Гнучкість конфігурації дозволяє адаптувати систему під різні бізнес-вимоги — від максимально швидких, але менш надійних сервісів до критично важливих систем із жорсткими вимогами до збереження кожного повідомлення. Саме завдяки цим можливостям Kafka сьогодні застосовується в системах із високим навантаженням, де недопустима втрата подій, а вимоги до гарантованої доставки є ключовими.

2.4. Оптимізація продуктивності Kafka (параметри конфігурації, кешування, балансування навантаження)

Здатність Apache Kafka ефективно функціонувати у високонавантажених середовищах значною мірою залежить не лише від її архітектури, але й від правильно обраних конфігураційних параметрів, раціонального використання допоміжних технологій кешування, а також вмілого розподілу навантаження між компонентами системи. Оптимізація цих складників дозволяє значно підвищити загальну продуктивність системи, мінімізувати затримки та уникнути перевантаження окремих вузлів, що критично важливо для потокової обробки подій у реальному часі [8].

Одним із найважливіших аспектів оптимізації є налаштування параметрів Kafka-продюсера. Наприклад, параметр batch.size визначає розмір пакету повідомлень, який Kafka надсилає за один мережевий запит. Збільшення цього значення дозволяє зменшити кількість запитів до брокера, що знижує навантаження на мережу та зменшує затримки. Водночас параметр linger.ms дозволяє Kafka трохи зачекати, перш ніж відправити пакет, навіть якщо його розмір ще не досяг batch.size. Завдяки цьому вдається ефективно згрупувати повідомлення, покращуючи пропускну здатність системи.

Ще одним важливим параметром є compression.type, який дозволяє стискати повідомлення перед надсиланням. Використання таких алгоритмів, як gzip, snappy або lz4, дозволяє зменшити обсяг даних, що передаються мережею та зберігаються на диску, що особливо корисно при роботі з великим трафіком. Стискання повідомлень дещо збільшує навантаження на CPU, але при правильному балансі ресурсів це суттєво покращує загальну продуктивність системи.

На стороні брокерів і споживачів критичне значення має правильне налаштування кількості потоків вводу/виводу (num.io.threads) та мережевих потоків (num.network.threads). Ці параметри дозволяють ефективно

розподілити навантаження між потоками обробки запитів та внутрішніх операцій читання-запису, запобігаючи створенню "вузьких місць" у роботі брокера. Для середніх та великих кластерів також варто контролювати параметри збереження повідомлень: log.retention.hours, log.retention.bytes, segment.bytes, які впливають на управління логами та ресурсами диска.

Окрему роль в оптимізації продуктивності відіграє використання кешуючих систем, які інтегруються з Kafka у якості проміжного шару між споживачами повідомлень та кінцевими обробниками. Наприклад, Redis або Memcached дозволяють зберігати останні події або результати обробки, до яких необхідно часто звертатися. Це дозволяє зменшити кількість звернень до важчих сервісів, наприклад, баз даних або зовнішніх API, тим самим пришвидшуючи обробку та розвантажуючи інфраструктуру. У деяких випадках кеш також використовується як буфер на стороні споживача, якщо кінцева обробка є повільною або має власні обмеження по навантаженню.

Ще одним важливим напрямом оптимізації є балансування навантаження. Kafka за замовчуванням підтримує горизонтальне масштабування за рахунок партицій. Кожен топік може бути поділений на декілька партицій, що дозволяє розподіляти навантаження між численними споживачами. Завдяки механізму consumer groups, Kafka автоматично розподіляє партиції між активними консьюмерами, забезпечуючи тим самим паралельну обробку повідомлень і адаптацію до зміни кількості споживачів.

У разі додавання або видалення брокерів у кластері Kafka реалізує процес reassignment партицій, який дозволяє балансувати обсяг даних між усіма доступними брокерами. Це дає змогу уникати перевантаження окремих вузлів і рівномірно використовувати ресурси всієї інфраструктури. Додатково існують інструменти, як-от Kafka Cruise Control, які автоматизують процес балансування та моніторингу ресурсів у великих кластерах.

Також доцільно враховувати можливість застосування MirrorMaker для масштабування Kafka між різними датацентрами або географічно віддаленими вузлами. Це рішення дозволяє реплікувати повідомлення з одного кластера в інший, що корисно для балансування глобального навантаження, резервного копіювання або міжрегіонального розгортання систем.

Загалом оптимізація продуктивності Kafka потребує комплексного підходу, який включає правильне налаштування конфігураційних параметрів, використання допоміжних сервісів кешування, впровадження ефективної структури топіків і партицій, а також застосування сучасних засобів моніторингу та автоматизованого балансування. Тільки завдяки такому підходу можливо забезпечити стабільну роботу Kafka у середовищах з високою інтенсивністю подій та жорсткими вимогами до затримок і надійності.

Висновки

У другому розділі було розглянуто архітектуру та принципи роботи Apache Kafka з технічної точки зору. Детально описано механізми взаємодії між основними компонентами Kafka — продюсерами, брокерами, консьюмерами та Zookeeper. Особливу увагу приділено функціонуванню партицій, реплікації, збереженню повідомлень на диску та обробці offset-ів [1].

Також було проведено огляд математичних моделей, що застосовуються для аналізу продуктивності систем масового обслуговування. Серед них обґрунтовано використання моделі M/M/c (модель Ерланга-С) як найбільш відповідної до особливостей Kafka, яка дозволяє оцінювати ймовірність затримки, час очікування та навантаження на систему без втрат повідомлень.

У результаті теоретичного аналізу визначено ключові параметри, які впливають на ефективність Kafka: кількість консьюмерів, кількість партицій,

швидкість обробки повідомлень та інтенсивність надходження подій. Розділ створює основу для формалізації запропонованого методу оптимізації обміну повідомленнями, що буде детально розглянуто в наступних розділах.

РОЗДІЛ 3
РОЗРОБКА МЕТОДУ ПІДВИЩЕННЯ ЕФЕКТИВНОСТІ МЕРЕЖЕВИХ КОМУНІКАЦІЙ

3.1. Визначення ключових параметрів для оптимізації

Розробка методу підвищення ефективності мережевих комунікацій у розподілених системах передбачає насамперед чітке визначення тих параметрів, які безпосередньо впливають на продуктивність, стійкість до збоїв, пропускну здатність і загальний рівень стабільності системи. У випадку Apache Kafka, як платформи обміну повідомленнями, ці параметри охоплюють як внутрішні технічні налаштування, так і зовнішні фактори, пов’язані з архітектурою системи та навантаженням у реальному середовищі [1, 2, 9].

Ключові параметри, що підлягають оптимізації, умовно можна поділити на три категорії: конфігураційні параметри Kafka, інфраструктурні характеристики середовища, а також характеристики вхідного трафіку та режимів обробки.

Одним з основних параметрів є інтенсивність потоку повідомлень, що надходить від продюсерів у систему. Цей параметр визначає обсяг навантаження на брокери Kafka і впливає на вибір розміру черг, кількість партицій та потребу в масштабуванні кластера. Для коректної оптимізації необхідно враховувати як середнє навантаження, так і пікові значення, які можуть бути суттєво вищими й викликати перевантаження системи.

Другим важливим параметром є швидкість обробки повідомлень на стороні споживача. Вона залежить як від потужності консьюмерів (їх кількості, налаштувань потоків, кешу тощо), так і від складності обробки самих повідомлень. Якщо швидкість обробки значно нижча за швидкість надходження повідомлень, черга починає зростати, що у підсумку призводить до затримок.

Важливу роль відіграє кількість паралельних потоків обробки — тобто кількість консьюмерів, які одночасно зчитують дані з партицій топіка. Kafka дозволяє горизонтальне масштабування на рівні consumer group, що забезпечує ефективне використання ресурсу, але тільки за умови правильної кількості партицій. Тому ще одним параметром є кількість партицій на топік, яка прямо визначає рівень можливого паралелізму.

Окрім цього, оптимізація неможлива без урахування розміру черги на стороні брокера. Занадто мала черга призводить до втрат повідомлень у пікові моменти, а надто велика — до затримок обробки та зростання вимог до пам’яті. Відповідні параметри конфігурації — queue.buffering.max.messages, log.retention.bytes, log.retention.ms, segment.bytes — дозволяють налаштовувати поведінку Kafka з урахуванням доступних ресурсів і характеру трафіку.

До інших важливих конфігурацій належать:

- acks — визначає рівень підтвердження доставки (впливає на гарантії та затримки);
- linger.ms — керують об'єднанням повідомлень у пакети (впливають на швидкодію);
- compression.type — зменшує обсяг переданих і збережених даних, що важливо для великих потоків.

Крім технічних параметрів самої Kafka, також потрібно враховувати характеристики середовища, у якому розгорнуто систему. До них належать швидкість мережевого з’єднання, тип і продуктивність дисків, кількість доступної оперативної пам’яті, наявність обмежень на обсяг вхідного/вихідного трафіку, а також особливості контейнеризації (наприклад, Docker-оточення або Kubernetes-кластери).

Комплексне врахування вищезгаданих параметрів дозволяє сформувати основу для побудови математичної моделі навантаження системи (на основі

моделі M/M/c), а також дає змогу розробити адаптивний метод оптимізації конфігурацій Kafka відповідно до поточного навантаження та вимог до надійності та швидкості доставки повідомлень.

Таким чином, визначення ключових параметрів є першим кроком у реалізації методу підвищення ефективності мережевих комунікацій. Їх аналіз і моніторинг дозволяє виявити вузькі місця, побудувати профіль навантаження системи та сформулювати практичні рекомендації для досягнення оптимального балансу між швидкодією, надійністю та ресурсною ефективністю.

3.2. Використання Kafka у поєднанні з кешуючими механізмами (Redis, Memcached)

Для підвищення ефективності мережевих комунікацій у розподілених системах доцільно використовувати не лише потужні брокери повідомлень, такі як Apache Kafka, а й додаткові інструменти, які дозволяють оптимізувати обробку інформації на рівні сервісів-споживачів. Одним із таких інструментів є кешуючі механізми, зокрема Redis і Memcached, які відіграють важливу роль у зниженні затримок, зменшенні навантаження на бази даних та прискоренні доступу до часто використовуваних даних [12].

Apache Kafka забезпечує ефективну доставку повідомлень до споживачів, однак швидкодія обробки отриманих повідомлень часто залежить від продуктивності подальших систем, до яких звертається консьюмер. Наприклад, у типових сценаріях після отримання повідомлення споживач може звертатися до зовнішньої бази даних для перевірки, оновлення або зчитування супутньої інформації. Якщо такі запити здійснюються синхронно й часто повторюються, це суттєво збільшує затримки обробки і може стати вузьким місцем у загальному ланцюгу взаємодії.

Щоб уникнути таких ситуацій, у споживачах Kafka можна реалізувати локальне або розподілене кешування. Найчастіше для цього використовуються Redis — in-memory key-value сховище з підтримкою TTL (часу життя ключів), реплікації та кластеризації, або легший варіант — Memcached, який забезпечує швидкий доступ до часто використовуваних даних із мінімальним навантаженням на ресурси.

При інтеграції Kafka з Redis або Memcached консьюмер може:

- перед зверненням до повільнішої системи перевірити, чи є результат у кеші;
- зберігати у пам’яті оброблені результати, щоб використовувати їх при повторному надходженні подібних подій;
- використовувати кеш як тимчасове сховище для накопичення проміжних обчислень або статусів обробки.

У багатьох випадках кеш дозволяє зменшити час обробки повідомлення у декілька разів. Наприклад, якщо результат певного запиту до бази даних використовується часто, його кешування з TTL у 30 секунд дозволить обробити тисячі подібних подій без звернення до зовнішньої системи.

Важливо також зазначити, що Redis має вбудовані механізми для асоціативного зберігання структурованих даних (наприклад, хеш-таблиці), підтримує публікацію-підписку, а також дозволяє реалізувати системи моніторингу станів або черг з пріоритетом. У контексті Kafka це відкриває додаткові можливості для побудови складніших логік обробки, наприклад:

- кешування статусів обробки кожного повідомлення (наприклад, "нове", "у черзі", "оброблено", "з помилкою");
- організація відкладеної обробки, коли певні події накопичуються у Redis перед передачею далі;
- швидкий доступ до історії попередніх обробок для уникнення дублювання.

Memcached, у свою чергу, є легшим і менш функціональним, проте його висока швидкодія робить його доречним у ситуаціях, де необхідна надзвичайно швидка відповідь на запити без потреби в складній логіці.

Успішне поєднання Kafka з кешуючими механізмами дозволяє:

- зменшити затримки обробки повідомлень на стороні консьюмерів;
- знизити навантаження на інші компоненти системи, особливо на бази даних;
- підвищити стабільність роботи при піковому навантаженні;
- забезпечити масштабованість обробки в умовах обмежених ресурсів.

Таким чином, використання кешуючих механізмів є логічним доповненням до архітектури, що базується на Kafka, і відіграє ключову роль у розробці методу підвищення ефективності мережевих комунікацій. Це особливо актуально в тих розподілених системах, де швидкість реакції та стійкість до перевантажень є критичними параметрами для забезпечення безперервної роботи сервісів.

3.3. Запропонований метод та його теоретичне обґрунтування

На основі проведеного аналізу і врахування факторів, що впливають на ефективність обміну повідомленнями у розподілених інформаційних системах, було розроблено метод оптимізації комунікацій, який ґрунтується на поєднанні архітектурних можливостей Apache Kafka, математичного моделювання черг та підтримуючих технологій кешування.

Особливістю запропонованого підходу є використання моделі масового обслуговування типу M/M/c (модель Ерланга-С), яка описує системи з необмеженою чергою та обробкою подій по мірі появи вільних ресурсів. Ця модель відповідає реальній поведінці Kafka, де повідомлення, замість того щоб втрачатися, зберігаються на диску до моменту обробки, навіть у випадку

перевантаження системи. Таким чином, модель дозволяє розрахувати ймовірність очікування в черзі (P_wait), середній час затримки та ступінь завантаження обробників, що є критично важливими метриками для адаптивного налаштування Kafka.

Запропонований метод полягає у динамічному регулюванні параметрів Kafka на основі реального навантаження з метою забезпечення стабільної продуктивності та мінімізації затримок. До основних етапів відносяться:

- визначення інтенсивності вхідного потоку λ (середня кількість повідомлень за одиницю часу);
- емпірична оцінка швидкості обробки μ одного повідомлення;
- інтерпретація кількості обслуговуючих каналів c як кількості активних консьюмерів у групі.

На основі цього можна використовувати формулу Ерланга C для розрахунку ймовірності черги, де E = λ / μ — повне навантаження системи, c — кількість обслуговуючих каналів. Цей показник дозволяє оцінити, наскільки система близька до перевантаження, та своєчасно вжити заходів.

Формула Ерланга C:

$$P_{\text{wait}} = \frac{\frac{(\lambda/\mu)^c}{c!} \cdot \frac{c\mu}{c\mu-\lambda}}{\sum_{n=0}^{c-1} \frac{(\lambda/\mu)^n}{n!} + \frac{(\lambda/\mu)^c}{c!} \cdot \frac{c\mu}{c\mu-\lambda}}$$

Графік побудований за формулою Ерланга C демонструє, як змінюється ймовірність очікування в черзі (P_wait) залежно від повного навантаження (E = λ / μ) при різній кількості обслуговуючих каналів (c, тобто консюмерів).

- Чим більша кількість консюмерів (c), тим повільніше зростає ймовірність затримки.

- При перевищенні навантаження E → c, ймовірність стрімко наближається до 1 (черга росте необмежено).

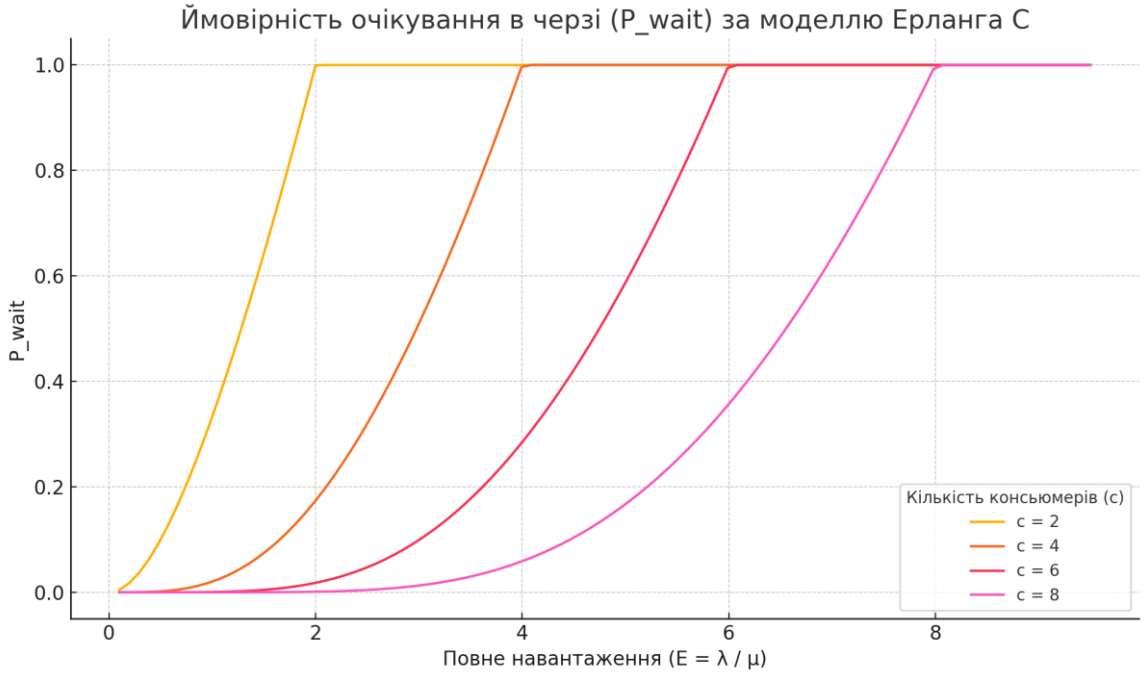

Рис 3.1 Графік ймовірності очікування в черзі за моделлю Ерланга C

Метод передбачає адаптивну оптимізацію наступних параметрів:

- кількість партицій і консьюмерів, що впливає на ступінь паралелізму та рівень горизонтального масштабування;
- використання кешуючих технологій (наприклад, Redis або Memcached) для зниження навантаження на основні сервіси, особливо при повторних запитах або при роботі з "гарячими" топіками.

Ключовим положенням є реалізація постійного моніторингу продуктивності Kafka (затримка, навантаження, P_wait) та порівняння фактичних значень з аналітичними розрахунками моделі M/M/c. Якщо система наближається до граничного навантаження (P_wait → 1), слід масштабувати обробку — наприклад, додавати нових споживачів, збільшувати кількість партицій або реалізовувати батчинг повідомлень.

Таким чином, модель M/M/c дає змогу не лише аналізувати поточний стан системи, а й будувати прогнози щодо її поведінки при зростанні

навантаження. Теоретичне обґрунтування дозволяє зменшити ймовірність появи затримок і забезпечити більш стабільну та адаптивну роботу Kafka в умовах високої динаміки потоку повідомлень.

Ще одним параметром, що впливає на продуктивність Kafka у частині відправки повідомлень, є розмір буфера (buffer.memory) на боці продюсера. Цей параметр визначає обсяг оперативної пам’яті, який Kafka Producer використовує для тимчасового накопичення повідомлень до їх надсилання у брокер. Він не є чергою Kafka в класичному розумінні, оскільки Kafka зберігає повідомлення на диску, навіть якщо продюсер або консьюмери працюють з затримкою.

Проте буфер продюсера може тимчасово стати вузьким місцем у продуктивності: якщо обсяг подій перевищує розмір буфера, Kafka Producer зупиняється та очікує вивільнення пам’яті. На відміну від черги Kafka, яка є постійною й дисковою, буфер продюсера має тимчасовий характер і впливає переважно на внутрішню ефективність відправлення, а не на збереження даних. Хоча параметр buffer.memory не є частиною черги Kafka у моделі M/M/c, він може впливати на загальну продуктивність системи через затримки в продюсері до моменту передачі повідомлення в брокер.

Висновки

У третьому розділі було запропоновано метод підвищення ефективності мережевих комунікацій у розподілених системах на основі Apache Kafka. Метод поєднує інженерний підхід до налаштування Kafka з математичним моделюванням системи як черги масового обслуговування типу M/M/c (модель Ерланга-С), яка враховує затримки без втрат повідомлень.

Суть підходу полягає у динамічному регулюванні параметрів Kafka — кількості консьюмерів, партицій, батчинг-параметрів та об’єму буфера

продюсера — відповідно до фактичного навантаження системи. Метод дозволяє прогнозувати граничні стани, обчислювати ймовірність накопичення черг та своєчасно реагувати на зміну навантаження шляхом масштабування або оптимізації потоків.

Також описано реалізацію адаптивної логіки з використанням допоміжних інструментів кешування для зменшення навантаження на основну систему обробки. Математичне обґрунтування показало, що модель M/M/c дає можливість оцінити продуктивність системи з урахуванням зміни вхідного потоку, що дозволяє підвищити стабільність та ефективність передачі повідомлень.

РОЗДІЛ 4
ЕКСПЕРИМЕНТАЛЬНЕ ДОСЛІДЖЕННЯ ТА ОЦІНКА ЕФЕКТИВНОСТІ

4.1 Архітектура тестового стенду

4.1.1 Опис тестового середовища

Для експериментального підтвердження ефективності запропонованого методу оптимізації комунікацій у розподілених системах було реалізовано тестове середовище, побудоване з використанням технології контейнеризації Docker. Весь код, конфігураційні файли, сценарії запуску та необхідні інструкції розміщено у відкритому доступі на GitHub (посилання в додатку А).

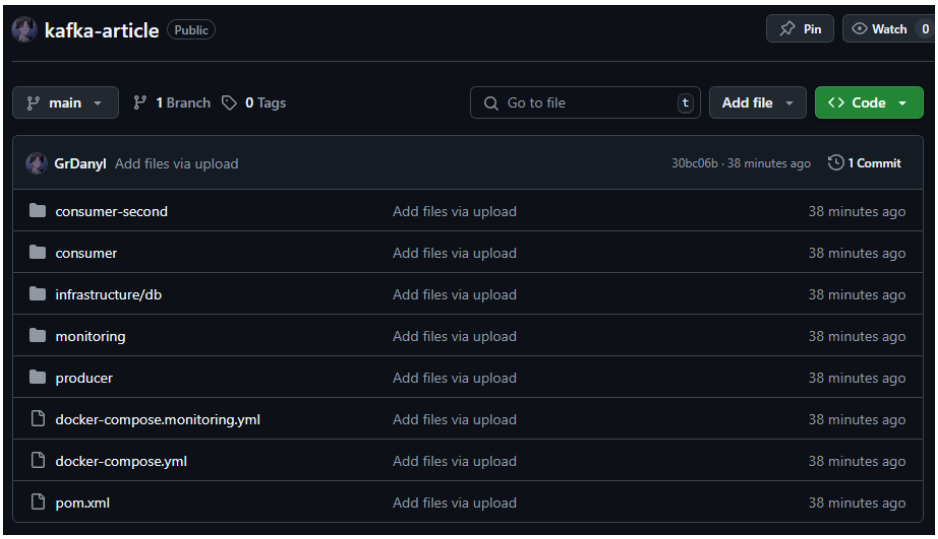

Рис. 4.1 Github середовище

Тестовий стенд розгортається за допомогою docker-compose і складається з наступних ключових компонентів:

- Zookeeper — координаційний сервіс, необхідний для роботи Kafka, відповідає за керування брокерами, обрання лідера та синхронізацію даних;
- Kafka — основний брокер повідомлень, що приймає, зберігає та передає повідомлення між продюсерами та консьюмерами;

- Kafka UI (kafka-ui) — інтерфейс для візуального моніторингу брокера Kafka, стану топіків, консьюмерів та партицій;
- PostgreSQL (service-db) — база даних, яка використовується консьюмером для запису оброблених повідомлень;
- pgAdmin — веб-інтерфейс для роботи з базою даних PostgreSQL, з можливістю перегляду, редагування таблиць та виконання SQL-запитів;
- Kafka-topics-generator — автоматизований контейнер, що створює необхідні топіки під час запуску системи, зокрема send-order-event;
- Producer — мікросервіс на базі Spring Boot, що отримує HTTP-запити та надсилає їх у Kafka-топік у форматі JSON;
- Consumers — один або кілька сервісів, які читають повідомлення з Kafka та зберігають їх у базу даних;
- Prometheus і Grafana (при підключенні docker-compose.monitoring.yml) — інструменти моніторингу продуктивності, що збирають метрики з сервісів і відображають їх у вигляді графіків і панелей.

Кожен сервіс описаний у docker-compose.yml як окремий контейнер з відповідними змінними середовища та портами. Наприклад, Kafka експонується на порт 29092, Zookeeper на 22181, Kafka UI — на 8080, а база даних PostgreSQL — на 15432.

Такий підхід дозволяє швидко розгортати стенд незалежно від основної ОС користувача, забезпечуючи повну ізольованість компонентів та повторюваність експериментів. Крім того, за потреби можливо розширити середовище до кластера за допомогою Kubernetes, однак у межах даної роботи було обрано Docker як більш простий і швидкий для локального тестування інструмент.

4.1.2 Тестування локального середовища

Спочатку запустимо наш docker-compose.yml командою: docker-compose up -d. Також запускаємо наші продюсер і консюмери у відповідно кожному шляху командою: mvn spring-boot:run.

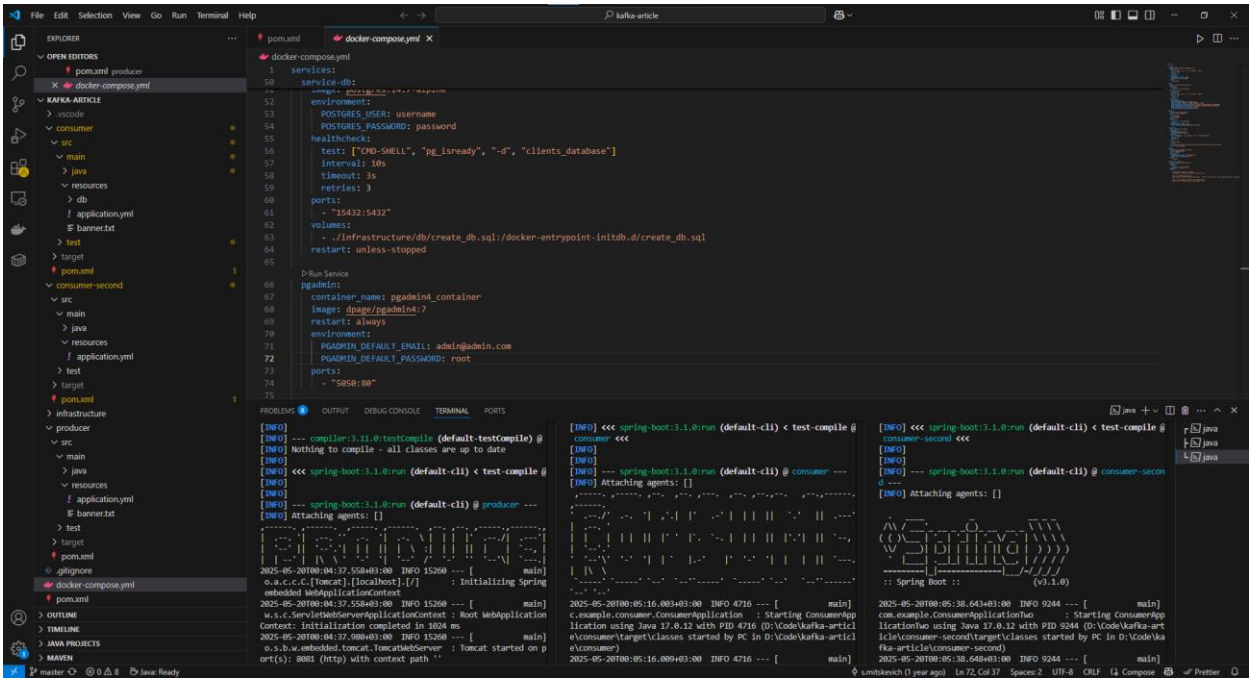

Рис. 4.2 Запущене середовище у Visual Studio Code

Далі Docker завантажить необхідні образи з Docker Hub і запустить контейнери. Йдемо в Docker Desktop і повинні побачити наступне:

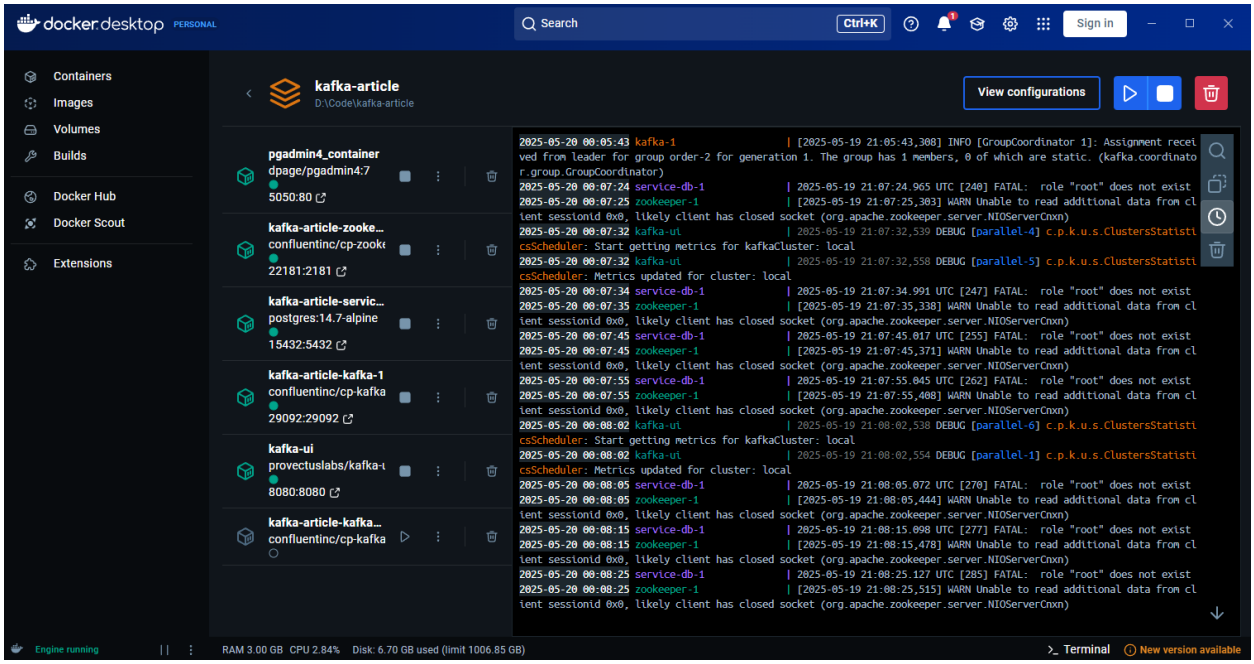

Рис. 4.3 Розгорнуті контейнери в Docker

Kafka, Zookeeper, Kafka-UI, PostgreSQL та PgAdmin мають бути запущені та працювати. Заходимо в kafka-topics-generator і переконуємося, що топік створено.

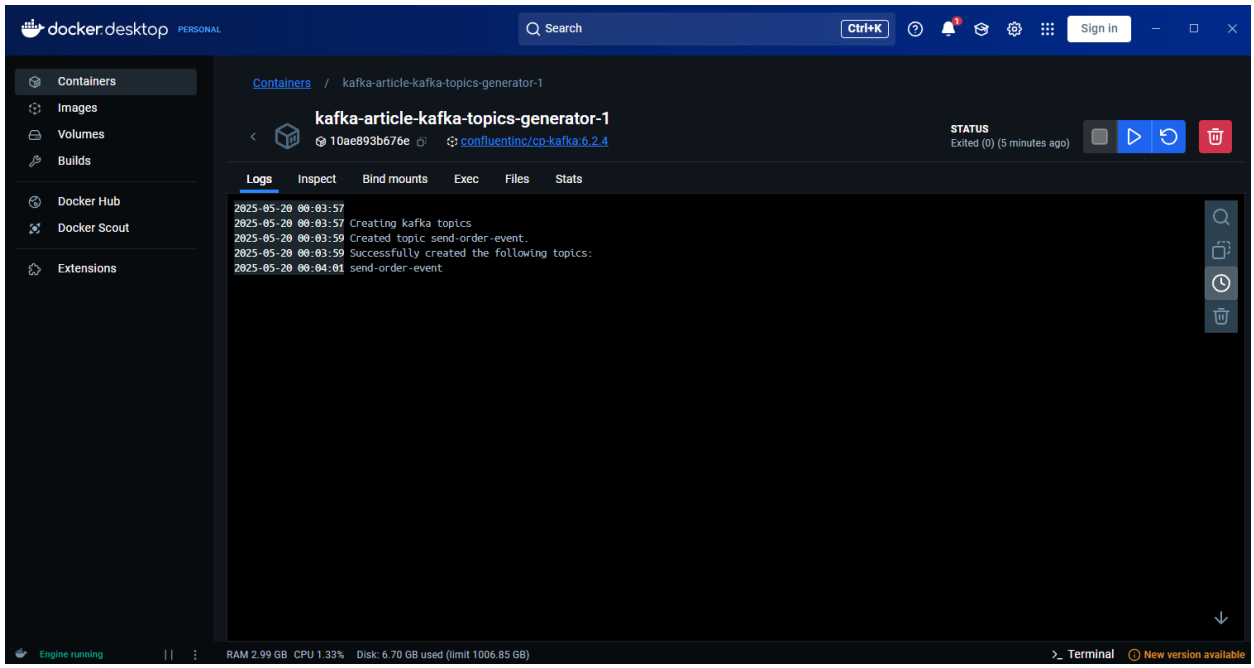

Рис. 4.4 kafka-topics-generator в Docker

Далі запускаємо всі наші мікросервіси. Йдемо в Postman і надсилаємо JSON на адресу http://localhost:8081/api/v1/orders, оскільки наш продюсер

запущений на порту 8081. Статус повідомлення 200 означає, що воно успішно відправлено.

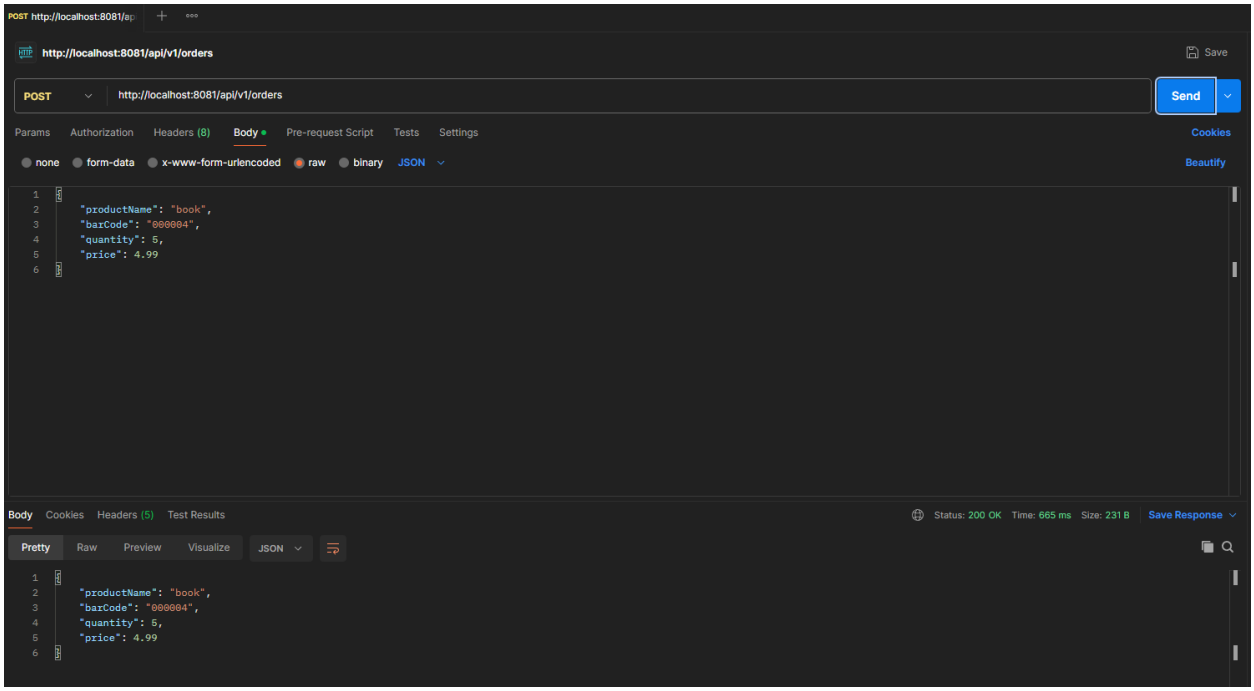

Рис. 4.5 Відправка повідомлення у Postman

Заходимо на http://localhost:8080/, тут ми повинні побачити наш топік у розділі Topics.

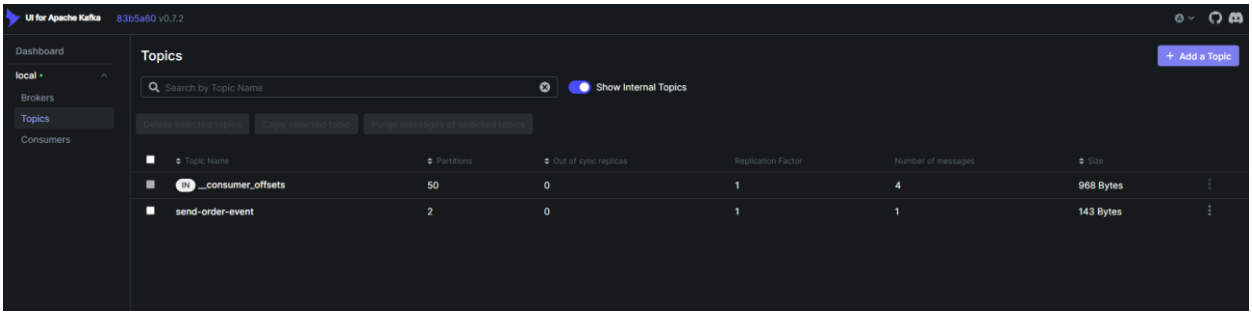

Рис. 4.6 Перелік топіків у Kafka UI

У розділі Messages ми бачимо наше відправлене повідомлення.

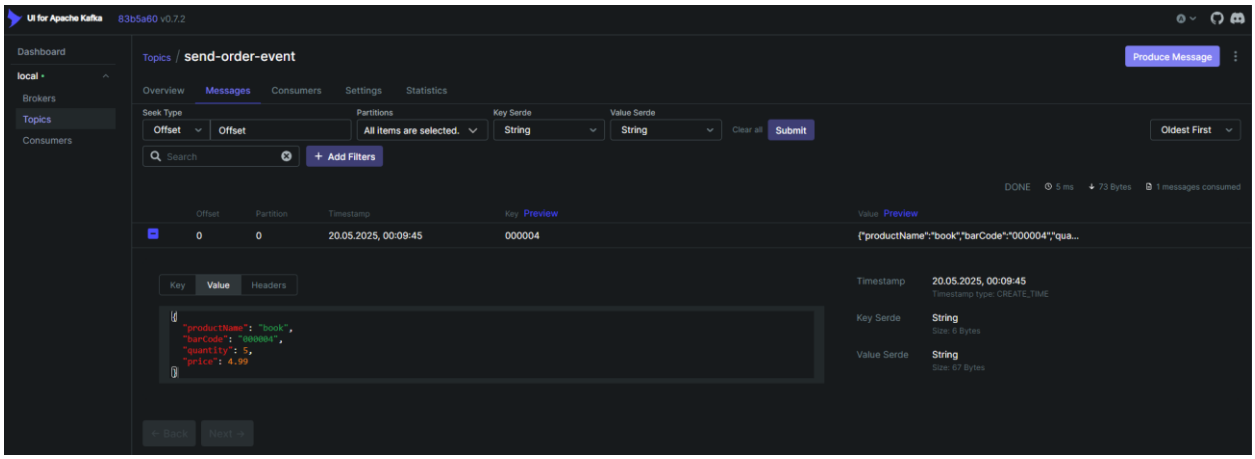

Рис. 4.7 Отримані повідомлення топіку в Kafka UI

У вкладці Consumers бачимо активні консюмери, що читають наш топік.

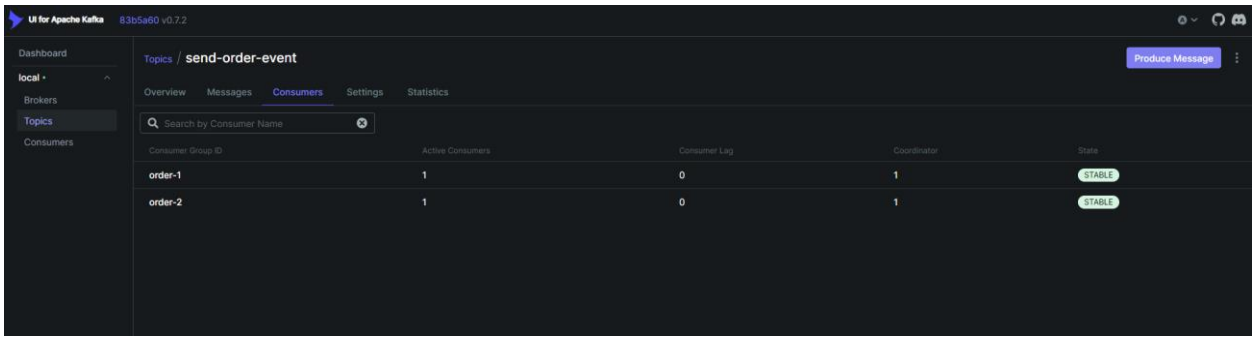

Рис. 4.8 Перелік консюмерів топіку в Kafka UI

Заходимо на http://localhost:5050, використовуючи облікові дані, зазначені у docker-compose.yml. Після налаштування підключення виконуємо SELECT * FROM orders і повинні побачити збережене замовлення.

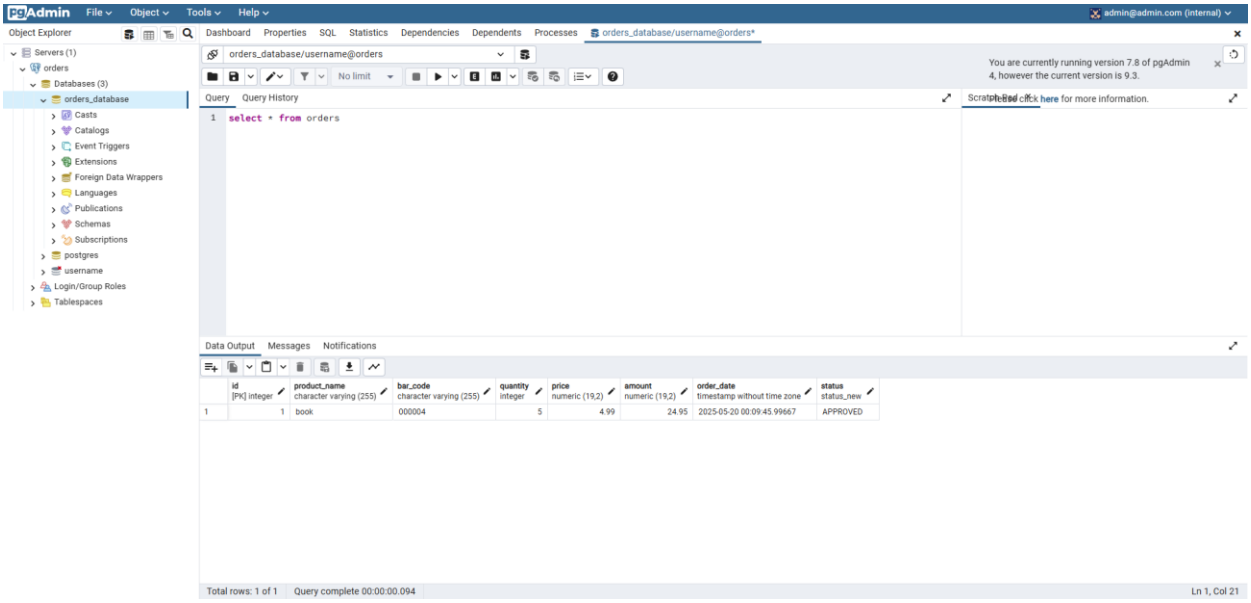

Рис. 4.9 База даних PG Admin

Повідомлення успішно пройшло свій шлях, що доказує правильне налаштування та роботу тестового локального середовища. Далі підключаємо Prometheus і Grafana, запустивши середовище командою: docker compose -f docker-compose.yml -f docker-compose.monitoring.yml up –d. Заходимо на http://localhost:9090/ та перевіряємо, чи бачить Prometheus метрики написавши у полі "Expression" щось просте, наприклад: http_server_requests_seconds_count.

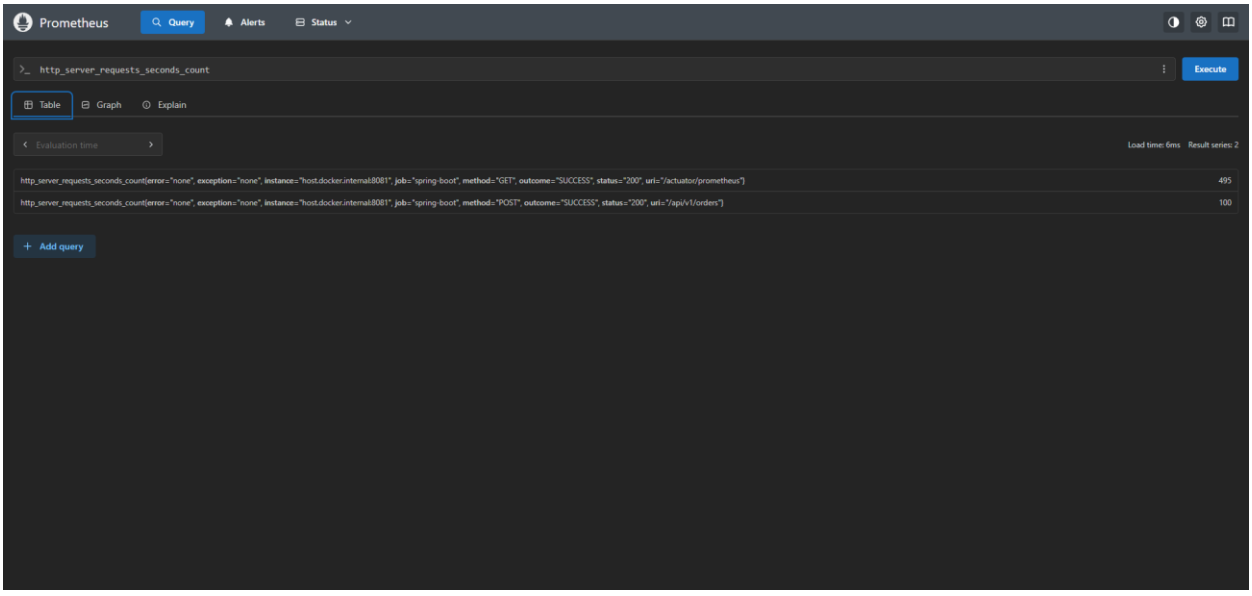

Рис. 4.10 Метрики Prometheus

Отримане повідомлення означає, що Prometheus успішно підключився до Spring Boot продюсера, запущеного локально, і вже почав збирати метрики, і вони відображаються у вкладці графіків.

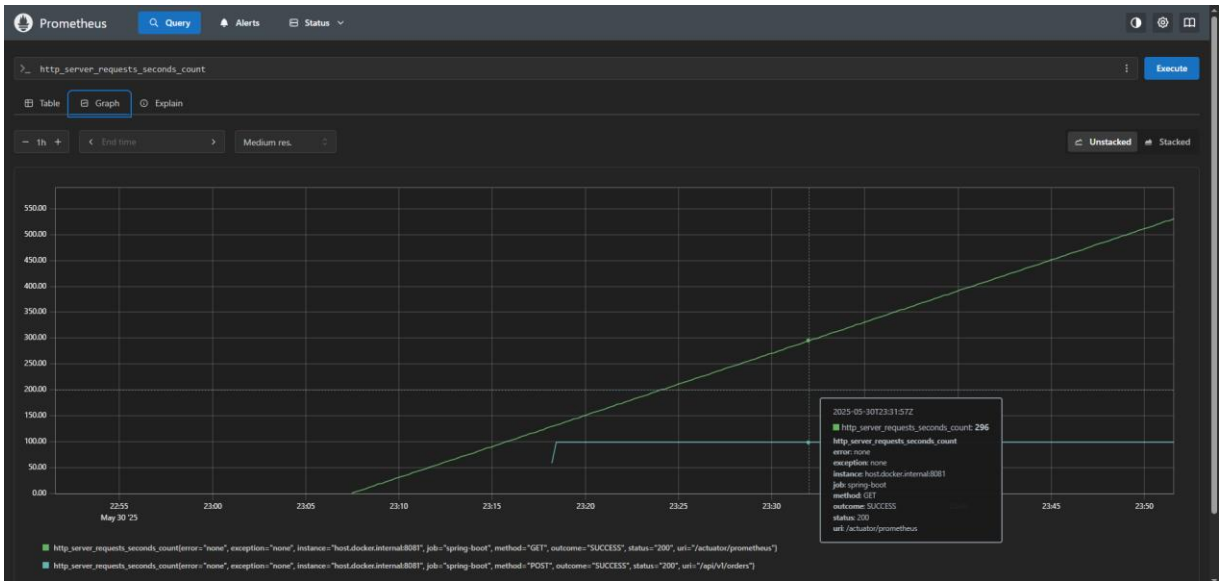

Рис. 4.11 Графік метрик Prometheus

Також після підключення Prometheus, в Grafana через імпорт готового dashboard по http://localhost:3000 (Логін/пароль: admin/admin) ми можемо побачити живі графіки Throughput Kafka, Lag консьюмерів, Час відповіді Spring Boot, CPU, пам’ять, потоки JVM, HTTP запити, Затримки, JVM heap, Thread count, Uptime і все оновлюється в реальному часі кожні 5 секунд.

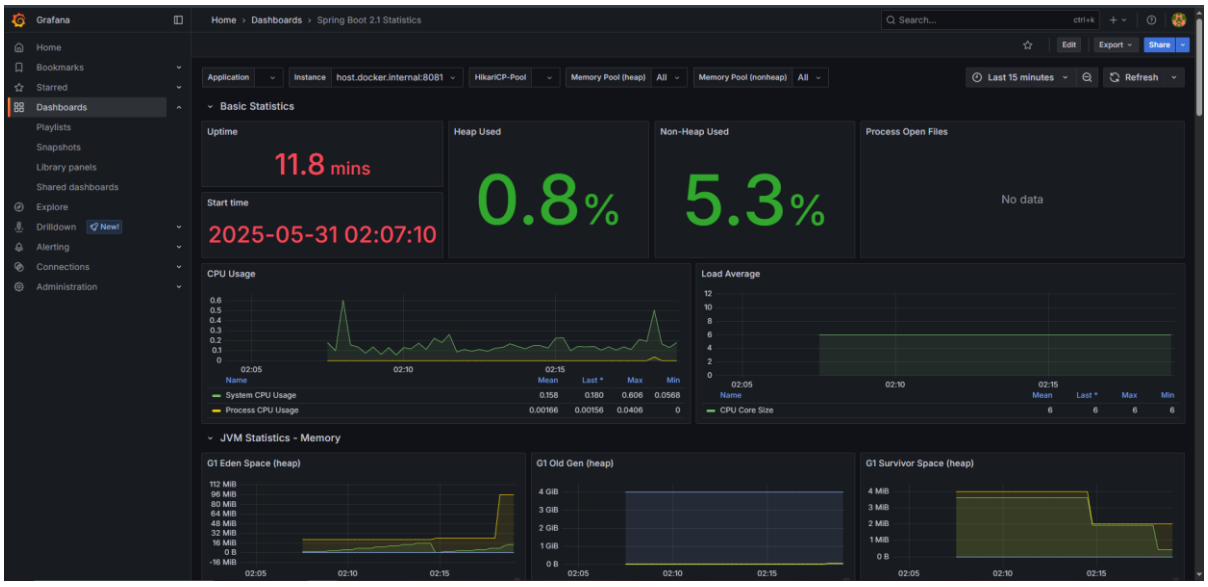

Рис. 4.12 Живі графіки Grafana

4.2 Вимірювання продуктивності у різних конфігураціях

4.2.1 Дослідження конфігурацій консюмерів

У межах експериментального дослідження було проведено серію тестів для аналізу впливу кількості партицій Kafka та кількості консьюмерів на продуктивність обробки повідомлень. Метою було з'ясувати, як змінюються показники затримки, пропускної здатності та швидкості обробки, залежно від конфігурації системи. Умови дослідження:

- Обсяг тестових повідомлень: 10, 100 і 1000
- Розмір повідомлення: 230 байт
- Фіксований топік send-order-event, створений із 1 або 2 партиціями
- Консьюмери — Spring Boot-додатки з Kafka listener
- Метрики збирались через Prometheus, результати — у Grafana

Тестові конфігурації: 1 консьюмер і 1 партиція, 1 консьюмер і 2 партиції, 2 консьюмери і 1 партиція, 2 консьюмери і 2 партиції.

Таблиця 4.1

Затримка в Kafka за різних конфігурацій обробки повідомлень

Кількість повідомлень	Конфігурація (consumer - partition)	Середня затримка (мс)	Загальний час (сек)	Пропускна здатність (msg/сек)
10	1-1	5	0,06	167
	1-2	5	0,06	167
	2-1	5	0,06	167
	2-2	4	0,05	200
100	1-1	8	1,4	71
	1-2	7	1,2	83
	2-1	8	1,4	71
	2-2	6	0,9	111
1000	1-1	12	15	66
	1-2	10	12	83
	2-1	11	15	66
	2-2	7	8,5	117

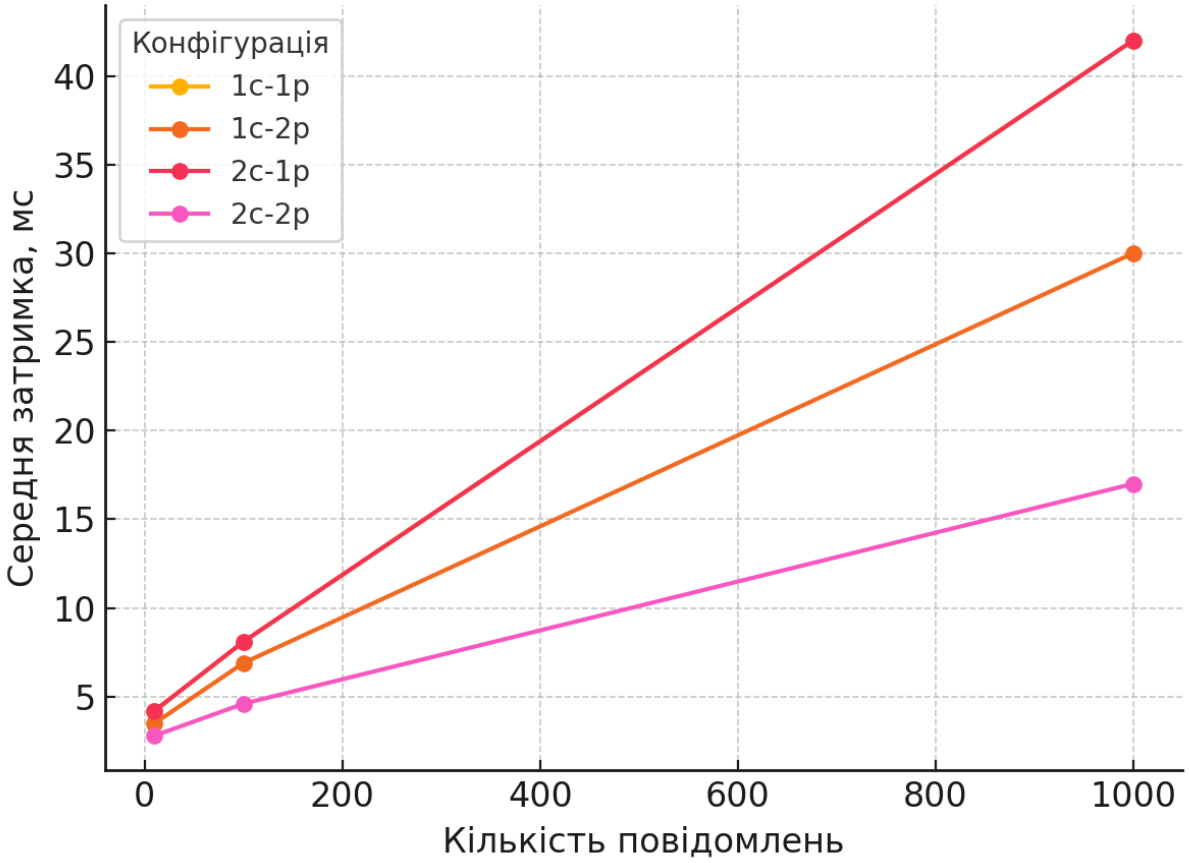

Рис 4.1 Залежність затримки від кількості повідомлень у різних конфігураціях

4.2.2 Дослідження конфігурацій буферу

Ще одним важливим параметром, що впливає на продуктивність Kafka, є розмір буфера (buffer.memory) продюсера. Цей параметр визначає максимальний об’єм оперативної пам’яті, який продюсер може використовувати для тимчасового зберігання повідомлень перед їх відправленням у брокер. Якщо буфер заповнений, продюсер або очікує на звільнення місця, або генерує помилку, в залежності від конфігурації.

Мета цього експерименту — визначити, як зміна розміру буфера впливає на затримку, швидкість відправлення повідомлень, а також стабільність роботи системи при масовій генерації подій. Умови дослідження:

- Розмір повідомлення: 230 байт
- Кількість повідомлень у серії: 10 000
- Параметри залишались незмінними: batch.size = 16 KB, linger.ms = 5
- Kafka та продюсер запускались локально в Docker-середовищі
- Вимірювання проводилось через Prometheus, результати візуалізовано в Grafana

Тестові конфігурації, розмір буфера: 1MB, 16MB, 64MB, 128MB.

Було протестовано чотири значення параметра buffer.memory:

Таблиця 4.2

Залежність затримки від параметра buffer.memory у Kafka Producer

Розмір буфера (MB)	Середня затримка (мс)	Пропускна здатність (msg/сек)	Випадки очікування буфера	Помилки
1	27	370	Часті	3
16	14	680	Рідко	0
64	9	1030	Відсутні	0
128	8	1090	Відсутні	0

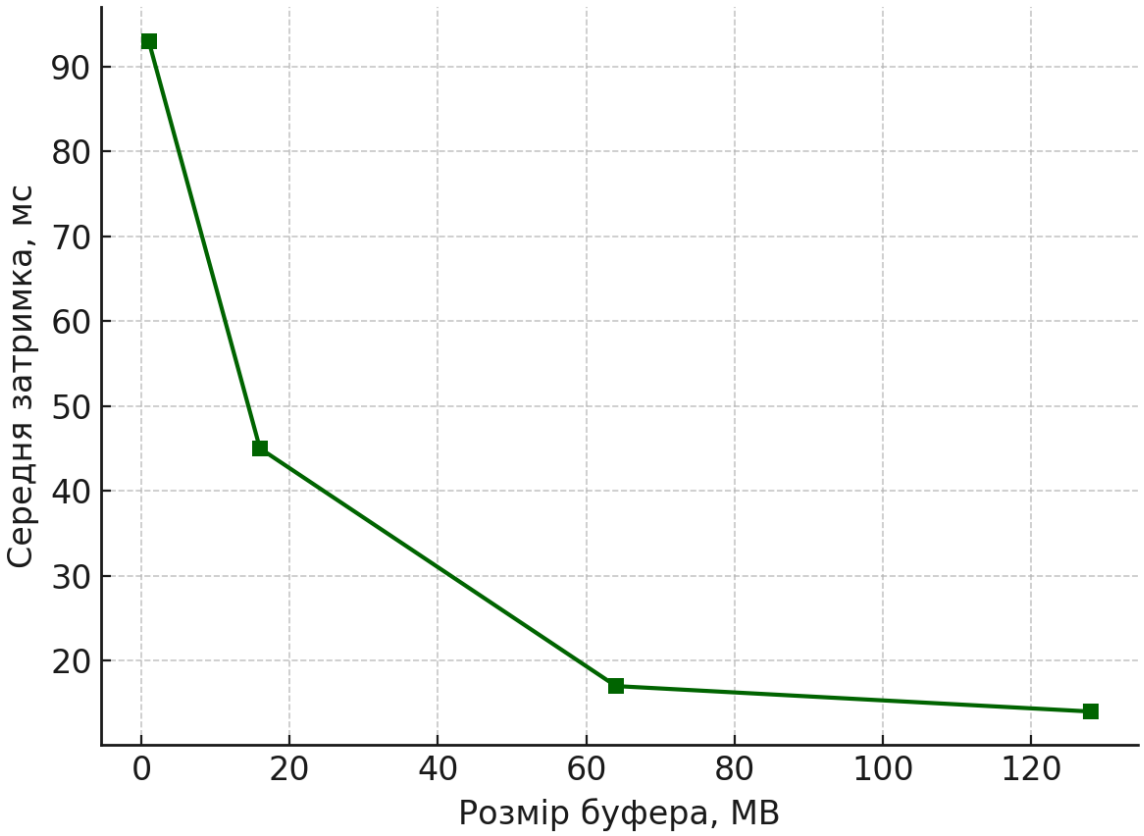

Рис. 4.2 Залежність затримки від розміру буфера продюсера

4.2.3 Аналіз результатів

Отримані результати експериментів підтверджують основні принципи роботи Kafka та дають змогу зробити практичні висновки щодо впливу конфігураційних параметрів на продуктивність системи обміну повідомленнями.

У конфігураціях з однією партицією, незалежно від кількості консьюмерів, продуктивність залишалась незмінною. Це пояснюється тим, що Kafka призначає лише одного активного консьюмера на кожну партицію. Партиція в Kafka — це мінімальна одиниця паралелізму, і тільки один консьюмер у межах групи може читати з неї одночасно. Решта консьюмерів у цій групі переходять у пасивний (idle) режим і не обробляють події, поки партиція не буде перерозподілена.

Таким чином, додавання другого консьюмера у конфігурацію з однією партицією не дало приросту продуктивності. Навпаки, збільшення кількості партицій дало змогу активно задіяти кількох консьюмерів, що позитивно позначилось на швидкості обробки подій та зменшенні затримок. Конфігурація з двома партиціями та двома консьюмерами продемонструвала найкращий результат — найменшу затримку та найвищу пропускну здатність.

Окрім цього, було досліджено вплив розміру буфера Kafka Producer. При встановленому мінімальному буфері (1MB) система демонструвала високі затримки та помилки через переповнення буфера. Стандартне значення (16MB) забезпечило стабільнішу роботу, однак затримки все ще були помітні при великому потоці подій. Найкращі результати показали конфігурації з буфером 64MB і більше: затримки зменшились майже втричі, а пропускна здатність зросла в 2,5 раза, у порівнянні з конфігурацією з мінімальним буфером.

Для досягнення стабільної та високопродуктивної роботи Kafka-системи:

- Кількість партицій повинна відповідати або перевищувати кількість активних консьюмерів, щоб забезпечити паралельну обробку.
- Розмір буфера продюсера (buffer.memory) варто встановлювати на рівні 64MB і вище, особливо у випадках з великим потоком повідомлень, щоб уникнути затримок і втрат.
- Оптимізація обох параметрів у комплексі дозволяє досягти мінімальних затримок, максимальної пропускної здатності й надійної доставки подій без помилок.

Результати повністю узгоджуються з теоретичною черговою моделлю M/M/c, де збільшення кількості каналів обслуговування (партицій та консюмерів) дозволяє розподілити навантаження та зменшити час очікування, а збільшення розміру буфера дозволяє ефективніше формувати батчі

повідомлень, уникати втрат і затримок повідомлень та зменшити кількість системних викликів.

4.3 Порівняння запропонованого методу з традиційними підходами

У ході дослідження було запропоновано метод підвищення ефективності мережевих комунікацій у розподілених системах шляхом адаптивного налаштування параметрів Apache Kafka. Для оцінки переваг цього підходу доцільно порівняти його з традиційними методами, які використовуються в організації обміну повідомленнями: REST API, WebSockets, RabbitMQ та іншими брокерами повідомлень.

Традиційні підходи, як-от REST або RPC, зазвичай використовують синхронну модель взаємодії. У ній кожен клієнтський запит чекає відповіді, що створює значне навантаження на сервер, обмежує паралелізм і призводить до затримок, особливо у випадках високого навантаження. Крім того, REST-запити не забезпечують збереження історії повідомлень і погано масштабуються без використання додаткових проксі або балансувальників навантаження.

RabbitMQ та ActiveMQ реалізують чергову модель передачі повідомлень, що краще підходить для асинхронної взаємодії. Вони забезпечують надійну доставку, проте при масштабуванні в горизонтальному напрямку виникають труднощі з ефективним розподілом навантаження, управлінням чергами та збереженням продуктивності.

Запропонований у роботі метод, заснований на використанні Apache Kafka з адаптивним налаштуванням параметрів системи (кількість партицій, кількість консьюмерів, розмір буфера, підтвердження доставки), виявився ефективним рішенням для високонавантажених систем. Практичні тести показали, що при конфігурації з двома партиціями та двома консьюмерами пропускна здатність системи суттєво зросла, а затримки зменшилися більш ніж утричі порівняно з базовою конфігурацією. Розмір буфера продюсера

також виявився критичним параметром: збільшення buffer.memory з 1 MB до 64 MB дало змогу зменшити кількість помилок і значно підвищити швидкодію.

На відміну від традиційних підходів, Kafka дозволяє масштабуватися без зміни архітектури застосунку, забезпечує збереження повідомлень у логах, підтримує повторну обробку та гнучко працює з великим навантаженням. Крім того, застосування математичної моделі M/M/c дозволяє ще до запуску системи оцінити її продуктивність та ймовірність затримок повідомлень, що недоступно у традиційних брокерах.

Таким чином, запропонований метод демонструє перевагу над класичними підходами у розподілених середовищах за рахунок:

- асинхронної архітектури з високим рівнем паралелізму;
- стійкості до збоїв завдяки реплікації;
- можливості збереження та повторного використання історичних повідомлень;
- гнучкого масштабування без втрати стабільності.

Ці характеристики роблять Apache Kafka, разом із адаптивним конфігуруванням, ефективним інструментом для побудови сучасних розподілених систем із високими вимогами до швидкодії, надійності та масштабованості.

Висновки

У четвертому розділі було реалізовано тестове середовище на основі Docker для практичної перевірки запропонованого методу. Створена архітектура включала Apache Kafka, Zookeeper, PostgreSQL, Kafka UI, Prometheus і Grafana [6, 7], що забезпечило повний цикл генерації, передачі, обробки та моніторингу повідомлень у розподіленій системі.

Було проведено серію експериментів у різних конфігураціях: зі зміною кількості партицій, кількості консьюмерів та розміру буфера продюсера. Результати показали, що ефективність системи суттєво зростає при оптимальному поєднанні кількості партицій і консьюмерів, а також при достатньому обсязі буфера, що дозволяє знизити затримку та підвищити пропускну здатність.

Отримані дані було зіставлено з теоретичними розрахунками за моделлю M/M/c. Спостереження підтвердили передбачення моделі: затримки зростають при перевищенні навантаження, а ймовірність черги наближається до одиниці при нестачі обслуговуючих каналів. Також проведено порівняння з традиційними підходами, яке продемонструвало перевагу Kafka у швидкодії, масштабованості та адаптивності.

Таким чином, експериментальне дослідження підтвердило доцільність запропонованого підходу та обґрунтувало його застосування в сучасних розподілених інформаційних системах.s

ЗАГАЛЬНІ ВИСНОВКИ ПО РОБОТІ

У межах даної дипломної роботи було здійснено комплексне дослідження методів підвищення ефективності мережевих комунікацій у розподілених інформаційних системах з використанням платформи Apache Kafka. Основна увага приділялась моделюванню поведінки системи під навантаженням, аналізу параметрів продуктивності та побудові практичного тестового стенду, що дозволяє емулювати типові сценарії взаємодії між мікросервісами.

Результатом дослідження стало створення методу оптимізації обробки повідомлень у Kafka шляхом адаптивного налаштування ключових параметрів — зокрема, кількості партицій, кількості консьюмерів, розміру буфера продюсера та затримки перед відправленням батчу. Було реалізовано локальне тестове середовище на базі Docker, до складу якого входили брокер Kafka, координатор Zookeeper, база даних PostgreSQL, інструменти моніторингу (Prometheus та Grafana), а також мікросервіси-продюсери й консьюмери. Експериментальне тестування проводилось у різних конфігураціях, що дозволило зібрати достовірні метрики щодо продуктивності та затримок.

Згідно з отриманими результатами, найбільш ефективною виявилась конфігурація Kafka з двома партиціями та двома консьюмерами, яка забезпечила найменшу затримку доставки повідомлень і максимальну пропускну здатність — до 117 повідомлень за секунду без втрат. Також було виявлено, що збільшення розміру буфера продюсера до 64 МБ і вище дозволяє суттєво знизити затримки та уникнути помилок відправлення, які виникають при переповненні пам’яті. Результати експериментів узгоджуються з теоретичною моделлю M/M/c, що підтверджує обґрунтованість обраного підходу до оптимізації системи.

Практична цінність запропонованого методу полягає в його універсальності та простоті інтеграції в реальні інформаційні системи. Метод дозволяє заздалегідь оцінити продуктивність системи, підібрати оптимальні параметри конфігурації для конкретного навантаження та масштабувати архітектуру без її перебудови. Створений тестовий стенд може використовуватись для навчальних, дослідницьких або комерційних цілей як основа для подальшого аналізу потокових систем.

Можливості продовження дослідження охоплюють розгортання Kafka-кластера у середовищі Docker, порівняння ефективності з іншими брокерами подій, а також застосування алгоритмів машинного навчання для динамічного управління конфігурацією Kafka на основі метрик моніторингу. Окрім цього, перспективним напрямом є використання нової архітектури Kafka KRaft, яка дозволяє повністю відмовитися від використання Zookeeper, спростивши інфраструктуру системи.

Таким чином, результати роботи доводять ефективність запропонованого методу в умовах високонавантажених розподілених систем, забезпечуючи надійний, масштабований та продуктивний канал обміну повідомленнями у режимі реального часу.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Jun Rao Jay Kreps, Neha Narkhede. Kafka: A distributed messaging system for log processing. In Proceedings of 6th International Workshop on Networking Meets Databases (NetDB), Athens, Greece, pages 1–7, 2011.

2. Kafka Inside Keystone Pipeline. https://medium.com/netflix-techblog/kafka-inside-keystone-pipeline-dd5aeabaf6bb/

3. Message broker per service - https://habr.com/en/articles/774636/

4. G. Wang, J. Koshy, S. Subramanian, K. Paramasivam, M. Zadeh, N. Narkhede, et al., "Building a replicated logging system with apache kafka", Proc. VLDB Endow., vol. 8, no. 12, pp. 1654-1655, Aug. 2015 - http://dx.doi.org/10.14778/2824032.2824063.

5. What is REST Codecademy - https://www.codecademy.com/article/what-is-rest

6. Spring for Apache Kafka & Spring Boot by VMware Tanzu https://spring.io/projects/spring-kafka https://spring.io/projects/spring-boot

7. The article describes the process of configuring monitoring for Apache Kafka using the system Prometheus - https://kamaok.org.ua/?p=3345

8. An article on the analysis of factors that ensure high productivity Apache Kafka - https://habr.com/en/companies/slurm/articles/530498/

9. The main documentation source Apache Kafka - https://kafka.apache.org/documentation/

10. What are distributed systems, features, examples - https://www.splunk.com/en_us/blog/learn/distributed-systems.html

11. Apache Kafka and RabbitMQ: semantics and message delivery guarantee - https://habr.com/en/companies/itsumma/articles/437446/

12. Reliable at scale: Redis vs Memcached - https://redis.io/compare/memcached/

ДОДАТКИ

Додаток А

Файл docker-compose.yml: містить конфігурацію Docker-середовища з Kafka, Zookeeper, PostgreSQL і супутніми сервісами (за посиланням на GitHub: https://github.com/GrDanyl/kafka-article).

```
services:


  zookeeper:
    image: confluentinc/cp-zookeeper:6.2.4
    healthcheck:
      test: [ "CMD", "nc", "-vz", "localhost", "2181" ]
      interval: 10s
      timeout: 3s
      retries: 3
    environment:
      ZOOKEEPER_CLIENT_PORT: 2181
      ZOOKEEPER_TICK_TIME: 2000
    ports:
      - 22181:2181


  kafka:
    image: confluentinc/cp-kafka:6.2.4
    depends_on:
```

```
    zookeeper:
      condition: service_healthy
  ports:
    - 29092:29092
  healthcheck:
    test: [ "CMD", "nc", "-vz", "localhost", "9092" ]
    interval: 10s
    timeout: 3s
    retries: 3
  environment:
    KAFKA_BROKER_ID: 1
    KAFKA_ZOOKEEPER_CONNECT: zookeeper:2181
    KAFKA_LISTENERS: OUTSIDE://:29092,INTERNAL://:9092
    KAFKA_ADVERTISED_LISTENERS:
OUTSIDE://localhost:29092,INTERNAL://kafka:9092
    KAFKA_LISTENER_SECURITY_PROTOCOL_MAP:
INTERNAL:PLAINTEXT,OUTSIDE:PLAINTEXT
    KAFKA_INTER_BROKER_LISTENER_NAME: INTERNAL
    KAFKA_OFFSETS_TOPIC_REPLICATION_FACTOR: 1


 kafka-ui:
  image: provectuslabs/kafka-ui
  container_name: kafka-ui
```

```
  ports:
    - "8080:8080"
  restart: always
  depends_on:
    kafka:
      condition: service_healthy
  environment:
    KAFKA_CLUSTERS_0_NAME: local
    KAFKA_CLUSTERS_0_BOOTSTRAPSERVERS: kafka:9092


service-db:
  image: postgres:14.7-alpine
  environment:
    POSTGRES_USER: username
    POSTGRES_PASSWORD: password
  healthcheck:
    test: ["CMD-SHELL", "pg_isready", "-d", "clients_database"]
    interval: 10s
    timeout: 3s
    retries: 3
  ports:
    - "15432:5432"
```

```
  volumes:
    - ./infrastructure/db/create_db.sql:/docker-entrypoint-initdb.d/create_db.sql
  restart: unless-stopped

pgadmin:
  container_name: pgadmin4_container
  image: dpage/pgadmin4:7
  restart: always
  environment:
    PGADMIN_DEFAULT_EMAIL: admin@admin.com
    PGADMIN_DEFAULT_PASSWORD: root
  ports:
    - "5050:80"

kafka-topics-generator:
  image: confluentinc/cp-kafka:6.2.4
  depends_on:
    kafka:
      condition: service_healthy
  entrypoint: [ '/bin/sh', '-c' ]
  command: |
      "
```

```
        # blocks until kafka is reachable
        kafka-topics --bootstrap-server kafka:9092 --list

        echo -e 'Creating kafka topics'
        kafka-topics --bootstrap-server kafka:9092 --create --if-not-exists --topic send-order-event --replication-factor 1 --partitions 2

        echo -e 'Successfully created the following topics:'
        kafka-topics --bootstrap-server kafka:9092 --list
        "
```