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

Навчально-науковий інститут телекомунікаційних систем

Кафедра телекомунікацій

«На правах рукопису»
УДК _____

«До захисту допущено»
Завідувач кафедри
_____ Сергій КРАВЧУК
«___»_____2025 р.

Магістерська дисертація
на здобуття ступеня магістра
за освітньо-професійною програмою «Інженерія та програмування інфокомунікацій»
зі спеціальності 172 «Електронні комунікації та радіотехніка»
на тему: «Використання мікросервісів для оптимізації продуктивності високонавантажених електронних комунікаційних систем»

Виконала:
студентка ІІ курсу, групи ЦК-41мп
Зінькевич Богдана Романівна _____

Керівник:
Доцент кафедри ТК НН ІТС, к.т.н., доцент
Явіся Валерій Сергійович _____

Рецензент:
Доцент кафедри ІТТ НН ІТС, к.т.н., доцент
Правило Валерій Володимирович _____

Засвідчую, що у цій магістерській дисертації немає запозичень з праць інших авторів без відповідних посилань.
Студентка _____

Київ – 2025 року

Національний технічний університет України

«Київський політехнічний інститут імені Ігоря Сікорського»

Навчально-науковий інститут телекомунікаційних систем
Кафедра телекомунікацій

Рівень вищої освіти – другий (магістерський)

Спеціальність – 172 «Електронні комунікації та радіотехніка»

Освітньо-професійна програма «Інженерія та програмування інфокомунікацій»

ЗАТВЕРДЖУЮ

Завідувач кафедри

_____ Сергій КРАВЧУК

«___»_____2025 р.

ЗАВДАННЯ
на магістерську дисертацію студентці

Зінькевич Богдані Романівні

1. Тема дисертації «Використання мікросервісів для оптимізації продуктивності високонавантажених електронних комунікаційних систем», науковий керівник дисертації Явіся Валерій Сергійович, к.т.н., доцент, затверджені наказом по університету від «03» листопада 2025 р. № 4772-с.

2. Термін подання студентом дисертації 15.12.2025 р.

3.Об’єкт дослідження: процеси функціонування високонавантажених електронних комунікаціний систем в умовах пікових навантажень

4.Предмет дослідження: мікросервісна архітектура та методи міжсервісної комунікації для побудови високопродуктивних систем обробки SMS-повідомлень з різними каналами доставки.

5. Перелік завдань, які потрібно розробити:

- Аналіз сучасних підходів до побудови високонавантажених систем обробки SMS.

- Проектування системи обробки SMS на базі мікросервісної архітектури.

- Реалізація та тестування системи обробки SMS.

- Розробка стартап-проєкту.

6. Орієнтовний перелік ілюстративного матеріалу:
1) Тема роботи
2) Мета та завдання дослідження
3) Порівняння архітектур.
5) Методи міжсервісної комунікації
6) Архітектура системи
7) Результати тестування продуктивності
8) Ключові метрики продуктивності
9) Економічна ефективність
10) Загальні висновки по роботі

7. Дата видачі завдання “ 02 ” вересня 2024 р.

Календарний план

№ з/п	Назва етапів виконання магістерської дисертації	Термін виконання етапів магістерської дисертації	Примітка
1	Розробка, оформлення, узгодження та затвердження технічного завдання на роботу. Аналітичний огляд інформаційних матеріалів. Підбір та опрацювання необхідної науково-технічної літератури.	02.09.2024 – 02.01.2025	виконано
2	Написання першого розділу	03.01.2025 – 06.03.2025	виконано
3	Написання другого розділу	07.03.2025 – 31.05.2025	виконано
4	Написання третього розділу	01.06.2025 – 01.08.2025	виконано
5	Розробка стартап проекту	02.08.2025 – 02.09.2025	виконано
6	Підготовка матеріалів до друку та оформлення пояснювальної записки	03.09.2025 – 10.12.2025	виконано

Студент Богдана ЗІНЬКЕВИЧ

Науковий керівник дисертації Валерій ЯВІСЯ

РЕФЕРАТ

Магістерська дисертація містить 133 сторінки, 14 рисунків, 5 таблиць. В роботі було використано 67 джерел.

Актуальність роботи. Мікросервісна архітектура стає ключовим підходом для підвищення продуктивності та масштабованості високонавантажених електронних комунікаційних систем. Зростання обсягів трафіку, вимоги до безперервної роботи та швидкого реагування роблять монолітні рішення малоефективними. Мікросервіси дозволяють гнучко масштабувати окремі компоненти, ізолювати збої та оптимізувати використання ресурсів, що забезпечує стабільну роботу системи навіть під піковими навантаженнями.

Мета роботи. Основною метою даної роботи є дослідження та порівняльний аналіз сучасних підходів до побудови високонавантажених систем обробки SMS-повідомлень на базі мікросервісної архітектури з різними методами міжсервісної комунікації, а також розробка практичної системи з можливістю вибору оптимального каналу доставки повідомлень.

Об'єкт дослідження – процеси обробки та доставки SMS-повідомлень у високонавантажених розподілених телекомунікаційних системах під час пікових навантажень.

Предмет дослідження – мікросервісна архітектура та методи міжсервісної комунікації для побудови високопродуктивних систем обробки SMS-повідомлень з різними каналами доставки.

Використані методи дослідження. Для досягнення поставленої мети були використані методи системного аналізу для дослідження архітектур електронних комунікаційних систем, методи порівняльного аналізу для оцінки ефективності синхронних та асинхронних підходів до передачі даних, методи об'єктно-орієнтованого проектування для розробки структури мікросервісної системи, методи експериментального дослідження для тестування

продуктивності системи під різними навантаженнями, методи статистичного аналізу для обробки результатів експериментів та формулювання висновків.

Отримані результати: виконано комплексне дослідження ефективності різних методів міжсервісної комунікації в контексті систем обробки SMS-повідомлень. Проведено порівняльний аналіз синхронних HTTP-запитів, асинхронних HTTP-запитів та асинхронної передачі через Apache Kafka за критеріями продуктивності, надійності та масштабованості при різних рівнях навантаження. Розроблено та реалізовано архітектурне рішення на базі мікросервісної архітектури, що дозволяє в межах однієї системи використовувати різні канали доставки залежно від специфічних вимог до обробки повідомлень. Тестування показало, що синхронний HTTP забезпечує пропускну здатність 40 повідомлень на секунду, асинхронний HTTP – 250 повідомлень на секунду, Apache Kafka – 450 повідомлень на секунду з нульовим відсотком помилок під максимальним навантаженням. Економічний аналіз показав чисту вигоду 10 мільйонів доларів за п'ятирічний період від впровадження мікросервісної архітектури переважно від зменшення вартості простою та додаткового доходу від швидшого виходу на ринок нових функцій.

Рекомендації щодо використання: розроблена система може бути використана як основа для побудови промислових рішень обробки повідомлень у телекомунікаційних компаніях, фінансових установах та інших організаціях, що потребують надійної обробки великих обсягів SMS. Результати порівняльного аналізу надають практичні рекомендації щодо вибору оптимального підходу залежно від конкретних вимог до системи: синхронний HTTP оптимальний для навантажень до 5 тисяч повідомлень на день, асинхронний HTTP для 5-30 тисяч повідомлень на день, Apache Kafka для понад 30 тисяч повідомлень на день або критичних систем з жорсткими вимогами надійності. Розроблений програмний код та архітектурні рішення можуть бути адаптовані для інших типів систем обробки повідомлень та розподілених

застосунків. Методика тестування продуктивності може використовуватися для оцінки ефективності аналогічних систем.

КЛЮЧОВІ СЛОВА: МІКРОСЕРВІСНА АРХІТЕКТУРА, SMS, APACHE KAFKA, HTTP, АСИНХРОННА КОМУНІКАЦІЯ, СИНХРОННА КОМУНІКАЦІЯ, РОЗПОДІЛЕНІ СИСТЕМИ, ПРОДУКТИВНІСТЬ, МАСШТАБОВАНІСТЬ, ТЕЛЕКОМУНІКАЦІЙНІ СИСТЕМИ, ОБРОБКА ПОВІДОМЛЕНЬ, REST API, SPRING BOOT, POSTGRESQL, DOCKER, ПРОПУСКНА ЗДАТНІСТЬ, НАДІЙНІСТЬ, МІЖСЕРВІСНА КОМУНІКАЦІЯ.

ABSTRACT

The master's thesis contains 133 pages, 14 figures, and 5 tables. A total of 67 sources were used in the work.

Relevance of the study. Microservice architecture has become a key approach for increasing the performance and scalability of high-load electronic communication systems. The growing volume of traffic, together with the requirements for uninterrupted operation and rapid response, makes monolithic solutions inefficient. Microservices enable flexible scaling of individual components, fault isolation, and optimized resource utilization, ensuring stable system operation even under peak loads.

Purpose of the work. The main purpose of this thesis is to research and perform a comparative analysis of modern approaches to building high-load SMS message processing systems based on microservice architecture with different inter-service communication methods, as well as to develop a practical system capable of selecting the optimal message delivery channel.

Object of research – SMS message processing and delivery processes in high-load distributed telecommunication systems during peak loads.

Subject of research – microservice architecture and inter-service communication methods for building high-performance SMS processing systems with various delivery channels.

Methods used. System analysis methods were applied to study electronic communication system architectures; comparative analysis methods were used to evaluate the efficiency of synchronous and asynchronous data transmission approaches; object-oriented design methods were used to develop the structure of the microservice system; experimental research methods were used to test system performance under different loads; statistical analysis methods were applied to process the experimental results and formulate conclusions.

Results obtained: comprehensive research of different inter-service communication method efficiency in the context of SMS message processing systems was performed. Comparative analysis of synchronous HTTP requests, asynchronous HTTP requests and asynchronous transmission via Apache Kafka by performance, reliability and scalability criteria at different load levels was conducted. Architectural solution based on microservice architecture was developed and implemented, allowing the use of different delivery channels within one system depending on specific message processing requirements. Testing showed that synchronous HTTP provides throughput of 40 messages per second, asynchronous HTTP – 250 messages per second, Apache Kafka – 450 messages per second with zero error rate under maximum load. Economic analysis showed net benefit of 10 million dollars over five-year period from microservice architecture implementation, mainly from reduced downtime costs and additional revenue from faster time-to-market for new features.

Recommendations for use: the developed system can be used as foundation for building industrial message processing solutions in telecommunication companies, financial institutions and other organizations requiring reliable processing of large SMS volumes. Comparative analysis results provide practical recommendations for optimal approach selection depending on specific system requirements: synchronous HTTP is optimal for loads up to 5 thousand messages per day, asynchronous HTTP for 5-30 thousand messages per day, Apache Kafka for over 30 thousand messages per day or critical systems with strict reliability requirements. Developed program code and architectural solutions can be adapted for other types of message processing systems and distributed applications. Performance testing methodology can be used for evaluating similar system efficiency.

KEYWORDS: MICROSERVICE ARCHITECTURE, SMS, APACHE KAFKA, HTTP, ASYNCHRONOUS COMMUNICATION, SYNCHRONOUS COMMUNICATION, DISTRIBUTED SYSTEMS, PERFORMANCE, SCALABILITY, TELECOMMUNICATION SYSTEMS, MESSAGE PROCESSING,

REST API, SPRING BOOT, POSTGRESQL, DOCKER, THROUGHPUT, RELIABILITY, INTER-SERVICE COMMUNICATION.

ЗМІСТ

ПЕРЕЛІК СКОРОЧЕНЬ

API	Application Programming Interface (програмний інтерфейс додатку)
ACID	Atomicity, Consistency, Isolation, Durability (атомарність, консистентність, ізоляція, довговічність)
AMQP	Advanced Message Queuing Protocol (протокол просунутої черги повідомлень)
AWS	Amazon Web Services (веб-сервіси Amazon)
CAP	Consistency, Availability, Partition tolerance (консистентність, доступність, стійкість до розділень)
CCPA	California Consumer Privacy Act (Каліфорнійський закон про конфіденційність споживачів)
CI/CD	Continuous Integration/Continuous Deployment (безперервна інтеграція/безперервне розгортання)
CQRS	Command Query Responsibility Segregation (розділення відповідальності команд та запитів)
CRM	Customer Relationship Management (управління взаємовідносинами з клієнтами)
CSS	Cascading Style Sheets (каскадні таблиці стилів)
CSV	Comma-Separated Values (значення, розділені комами)
DDoS	Distributed Denial of Service (розподілена атака відмови в обслуговуванні)
DevOps	Development and Operations (розробка та експлуатація)
DNS	Domain Name System (система доменних імен)
DTO	Data Transfer Object (об'єкт передачі даних)
ELK	Elasticsearch, Logstash, Kibana (стек технологій для логування та аналізу)

GDPR	General Data Protection Regulation (загальний регламент захисту даних)
gRPC	Google Remote Procedure Call (система віддалених викликів процедур від Google)
HATEOAS	Hypermedia as the Engine of Application State (гіпермедіа як двигун стану додатку)
HTML	HyperText Markup Language (мова розмітки гіпертексту)
HTTP	HyperText Transfer Protocol (протокол передачі гіпертексту)
IDL	Interface Definition Language (мова опису інтерфейсів)
JMeter	Java performance testing tool (інструмент тестування продуктивності Java)
JPA	Java Persistence API (API персистентності Java)
JSON	JavaScript Object Notation (нотація об'єктів JavaScript)
JWT	JSON Web Token (веб-токен JSON)
Kafka	Apache Kafka (розподілена платформа потокової обробки)
MDC	Mapped Diagnostic Context (відображений діагностичний контекст)
MVC	Model-View-Controller (модель-вид-контролер)
OAuth	Open Authorization (відкрита авторизація)
OpenID	Open Identification (відкрита ідентифікація)
ORM	Object-Relational Mapping (об'єктно-реляційне відображення)
PostgreSQL	Post-gres-Q-L (система управління реляційними базами даних)
RabbitMQ	Message broker (брокер повідомлень)
RCS	Rich Communication Services (послуги розширеного зв'язку)
REST	Representational State Transfer (передача репрезентативного стану)
RSID	Revision Session ID (ідентифікатор сесії ревізії)

SAML	Security Assertion Markup Language (мова розмітки тверджень безпеки)
SASL/SCRAM	Simple Authentication and Security Layer/Salted Challenge Response Authentication Mechanism (механізм автентифікації)
SMS	Short Message Service (служба коротких повідомлень)
SMPP	Short Message Peer-to-Peer (протокол обміну короткими повідомленнями)
SQL	Structured Query Language (мова структурованих запитів)
SQS	Simple Queue Service (проста служба черг)
SSL	Secure Sockets Layer (рівень захищених сокетів)
TCO	Total Cost of Ownership (загальна вартість володіння)
TCP	Transmission Control Protocol (протокол керування передачею)
TLS	Transport Layer Security (безпека транспортного рівня)
TTL	Time To Live (час життя)
UUID	Universally Unique Identifier (універсальний унікальний ідентифікатор)
XML	eXtensible Markup Language (розширювана мова розмітки)
ІКС	інформаційно-комунікаційна система
БД	база даних
ЗВВ	журнал випереджувального запису

ВСТУП

Сучасні електронні комунікаційні системи є невід'ємною частиною глобальної інфраструктури зв'язку, що забезпечує обмін даними між мільярдами користувачів у всьому світі. SMS-повідомлення, незважаючи на появу численних альтернативних каналів комунікації, залишаються одним із найпопулярніших та найнадійніших способів передачі інформації завдяки їхній універсальності, високій доступності та незалежності від наявності інтернет-з'єднання. Телекомунікаційні оператори, фінансові установи, медичні заклади, державні організації та комерційні підприємства активно використовують SMS для критично важливих повідомлень, двофакторної автентифікації, інформування клієнтів та масових розсилок.

Однак зростання обсягів SMS-трафіку, особливо під час пікових навантажень при масових розсилках, святкових подіях або надзвичайних ситуаціях, ставить перед розробниками серйозні виклики щодо забезпечення стабільної та швидкої роботи систем обробки повідомлень. Традиційні монолітні архітектури часто не справляються з різкими змінами навантаження, що призводить до затримок у доставці, втрати повідомлень або повної недоступності сервісу. Це створює потребу в дослідженні та впровадженні сучасних архітектурних підходів, здатних забезпечити високу пропускну здатність, надійність та еластичне масштабування систем обробки SMS.

Актуальність теми дослідження обумовлюється зростаючими вимогами до продуктивності та надійності систем обробки SMS-повідомлень в умовах постійного збільшення обсягів трафіку та непередбачуваних пікових навантажень. Мікросервісна архітектура та сучасні методи міжсервісної комунікації відкривають нові можливості для побудови високомасштабованих розподілених систем, здатних ефективно обробляти мільйони повідомлень на секунду. Порівняльний аналіз синхронних та асинхронних методів передачі даних є критично важливим для прийняття обґрунтованих архітектурних рішень

при проектуванні систем, що повинні забезпечувати стабільну роботу як під нормальним, так і під екстремальним навантаженням. Дослідження цих питань дозволяє сформулювати практичні рекомендації щодо вибору оптимальних технологій та архітектурних патернів для конкретних умов експлуатації.

Зв'язок роботи з науковими програмами, планами, темами. Робота виконується в рамках освітньо-професійної програми підготовки бакалаврів за спеціальністю 172 "Телекомунікації та радіотехніка" кафедри інформаційно-комунікаційних технологій та систем Навчально-наукового інституту телекомунікаційних систем КПІ ім. Ігоря Сікорського. Тематика дослідження узгоджується з науковими напрямками кафедри щодо розробки та дослідження сучасних телекомунікаційних систем і мереж, методів підвищення їх ефективності та надійності функціонування.

Мета роботи полягає у дослідженні та порівняльному аналізі сучасних підходів до побудови високонавантажених систем обробки SMS-повідомлень на базі мікросервісної архітектури з різними методами міжсервісної комунікації, а також розробці практичної системи з можливістю вибору оптимального каналу доставки повідомлень залежно від вимог до продуктивності та надійності.

Для досягнення поставленої мети необхідно вирішити наступні задачі:

— Провести огляд та аналіз сучасних архітектур електронних комунікаційних систем, виявити основні проблеми обробки повідомлень під час пікових навантажень та способи їх вирішення.

— Дослідити принципи мікросервісної архітектури та проаналізувати основні способи міжсервісної комунікації в телекомунікаційних системах, включаючи синхронні та асинхронні методи передачі даних.

— Виконати порівняльний аналіз синхронних та асинхронних методів передавання даних у розподілених системах з точки зору продуктивності, надійності та складності реалізації.

— Розробити структуру системи обробки SMS на базі мікросервісної архітектури з підтримкою множинних каналів доставки повідомлень.

— Спроектувати механізми синхронної та асинхронної доставки повідомлень через HTTP та Apache Kafka, а також розробити модель даних для зберігання SMS та результатів обробки.

— Реалізувати сервіс генерації SMS з користувацьким інтерфейсом для запуску тестових сценаріїв та сервіс обробки повідомлень з підтримкою трьох каналів доставки.

— Провести тестування продуктивності системи при різних обсягах навантаження та виконати порівняльний аналіз ефективності різних каналів доставки за метриками часу обробки, відсотка помилок та пропускної здатності.

— Сформулювати рекомендації щодо вибору оптимального методу комунікації залежно від конкретних умов експлуатації та обмежень ресурсів, а також оцінити економічну ефективність впровадження мікросервісної архітектури.

Об'єкт дослідження -- процеси обробки та доставки SMS-повідомлень у високонавантажених розподілених телекомунікаційних системах під час пікових навантажень.

Предмет дослідження -- мікросервісна архітектура та методи міжсервісної комунікації для побудови високопродуктивних систем обробки SMS-повідомлень з різними каналами доставки.

Методи дослідження. У роботі використовуються методи системного аналізу для дослідження архітектур електронних комунікаційних систем, методи порівняльного аналізу для оцінки ефективності синхронних та асинхронних підходів до передачі даних, методи об'єктно-орієнтованого проектування для розробки структури мікросервісної системи, методи експериментального дослідження для тестування продуктивності системи під різними

навантаженнями, а також методи статистичного аналізу для обробки результатів експериментів та формулювання висновків.

Наукова новизна одержаних результатів. Виконано комплексне дослідження ефективності різних методів міжсервісної комунікації в контексті систем обробки SMS-повідомлень. Проведено порівняльний аналіз синхронних HTTP-запитів та асинхронної передачі через Apache Kafka за критеріями продуктивності, надійності та масштабованості при різних рівнях навантаження. Розроблено архітектурне рішення, що дозволяє в межах однієї системи використовувати різні канали доставки залежно від специфічних вимог до обробки повідомлень. Отримані експериментальні дані щодо порівняння продуктивності різних підходів створюють основу для прийняття обґрунтованих архітектурних рішень при проектуванні високонавантажених телекомунікаційних систем.

Практичне значення одержаних результатів. Розроблена система обробки SMS на базі мікросервісної архітектури може бути використана як основа для побудови промислових рішень обробки повідомлень у телекомунікаційних компаніях, фінансових установах та інших організаціях, що потребують надійної обробки великих обсягів SMS. Результати порівняльного аналізу різних методів комунікації надають практичні рекомендації щодо вибору оптимального підходу залежно від конкретних вимог до системи. Розроблений програмний код та архітектурні рішення можуть бути адаптовані для інших типів систем обробки повідомлень та розподілених застосунків. Методика тестування продуктивності може використовуватися для оцінки ефективності аналогічних систем.

Структура та обсяг роботи. Магістерська дисертація складається зі вступу, чотирьох розділів, загальних висновків, списку використаних джерел та додатків. Додатки містять лістинги програмного коду, результати тестування та додаткові матеріали.

У першому розділі проведено аналіз сучасних підходів до побудови високонавантажених систем обробки SMS, включаючи огляд архітектур електронних комунікаційних систем, аналіз мікросервісної архітектури та способів міжсервісної комунікації, а також порівняльний аналіз синхронних та асинхронних методів передавання даних у розподілених системах.

У другому розділі представлено проектування системи обробки SMS на базі мікросервісної архітектури, включаючи розробку структури системи та взаємодії мікросервісів, проектування механізмів синхронної та асинхронної доставки повідомлень через HTTP та Kafka, а також розробку моделі даних та схеми баз даних.

У третьому розділі описано реалізацію та тестування системи обробки SMS, включаючи розробку сервісу генерації SMS з користувацьким інтерфейсом, реалізацію сервісу обробки повідомлень з підтримкою трьох каналів доставки, а також результати тестування продуктивності системи при різних обсягах навантаження.

У четвертому розділі виконано розроблення стартап-проекту системи обробки SMS, включаючи порівняльний аналіз метрик продуктивності для різних каналів доставки, рекомендації щодо вибору оптимального методу комунікації, а також оцінку економічної ефективності впровадження мікросервісної архітектури в системах обробки SMS-повідомлень.

РОЗДІЛ 1

АНАЛІЗ СУЧАСНИХ ПІДХОДІВ ДО ПОБУДОВИ ВИСОКОНАВАНТАЖЕНИХ СИСТЕМ ОБРОБКИ SMS

1.1. Огляд архітектур електронних комунікаційних систем та проблеми обробки повідомлень під час пікових навантажень

Сучасні електронні комунікаційні системи є невід'ємною частиною глобальної інфраструктури зв'язку, що забезпечує обмін даними між мільярдами користувачів по всьому світу. Телекомунікаційні оператори, фінансові установи, соціальні мережі та багато інших організацій покладаються на надійну та швидку доставку повідомлень для підтримки критично важливих бізнес-процесів. SMS-повідомлення залишаються одним із найпопулярніших каналів комунікації завдяки їхній універсальності, високій доступності та незалежності від інтернет-з'єднання. Однак зростання обсягів трафіку, особливо під час пікових навантажень, ставить перед розробниками серйозні виклики щодо забезпечення стабільної роботи систем. У цьому підрозділі розглядаються основні архітектурні підходи до побудови систем обробки SMS та проблеми, з якими стикаються оператори в умовах високого навантаження.

Традиційно системи обробки SMS будувалися на основі монолітної архітектури, де всі компоненти об'єднані в єдиний додаток, що виконується на одному або кількох серверах. Така архітектура має свої переваги: простота розгортання, легкість налагодження та відносно невисока складність інфраструктури на початкових етапах. Монолітні системи добре працюють при невеликих та середніх навантаженнях, коли кількість повідомлень не перевищує певний поріг. Однак із зростанням користувацької бази та розширенням функціональності монолітна архітектура стає вузьким місцем [1]. Модифікація

окремих компонентів вимагає перезапуску всієї системи, що призводить до простоїв і зниження доступності сервісу (табл. 1.1).

Таблиця 1.1

Порівняння монолітної та мікросервісної архітектури

Характеристика	**Монолітна архітектура**	**Мікросервісна архітектура**
Масштабованість	Вертикальна (обмежена)	Горизонтальна (необмежена)
Складність розгортання	Низька	Висока
Незалежність компонентів	Низька	Висока
Відмовостійкість	Низька (одна точка відмови)	Висока (ізоляція сервісів)
Технологічна гнучкість	Обмежена однією технологією	Різні технології для різних сервісів
Час розробки нових функцій	Повільніше (залежність компонентів)	Швидше (незалежні команди)

Проблема пікових навантажень особливо гостро проявляється під час масових подій, свят, рекламних кампаній або надзвичайних ситуацій, коли кількість SMS-повідомлень різко зростає за короткий проміжок часу. У таких умовах монолітні системи часто не справляються з потоком запитів, що призводить до затримок у доставці повідомлень, втрати даних або повного відмовлення в обслуговуванні. Оператори змушені передбачати такі сценарії та резервувати додаткові обчислювальні ресурси, що значно підвищує операційні витрати. Крім того, відсутність механізмів динамічного масштабування не дозволяє ефективно використовувати наявну інфраструктуру в періоди низького навантаження. Ця проблема стимулює пошук альтернативних архітектурних рішень, здатних адаптуватися до змінних умов експлуатації.

Розподілені системи обробки повідомлень пропонують більш гнучкий підхід до управління навантаженням шляхом розподілу обробки між множиною

незалежних вузлів. Такі системи дозволяють горизонтально масштабувати інфраструктуру, додаючи нові сервери в міру зростання трафіку. Кожен вузол відповідає за обробку певної частини повідомлень, що знижує ризик перевантаження окремих компонентів. Балансування навантаження між вузлами здійснюється за допомогою спеціалізованих алгоритмів, що враховують поточну завантаженість кожного сервера та пріоритетність повідомлень. Однак розподілені системи вносять додаткову складність у процеси моніторингу, налагодження та забезпечення консистентності даних [2]. Необхідність синхронізації стану між вузлами та обробка помилок мережевої взаємодії вимагають від розробників високої кваліфікації та ретельного проектування.

Одним із ключових аспектів архітектури комунікаційних систем є вибір моделі взаємодії між компонентами. Синхронна модель передбачає, що відправник очікує на підтвердження отримання повідомлення перед продовженням роботи. Це забезпечує гарантовану доставку та можливість негайної обробки помилок, однак призводить до блокування ресурсів на час очікування відповіді. В умовах високого навантаження синхронна взаємодія може стати причиною каскадних відмов, коли затримка в одному компоненті поширюється на всю систему. Асинхронна модель, навпаки, дозволяє відправнику продовжувати роботу одразу після відправлення повідомлення, не чекаючи підтвердження. Це підвищує пропускну здатність системи та зменшує ймовірність блокувань, однак ускладнює гарантування доставки та обробку помилок у реальному часі.

Чергування повідомлень (message queuing) є поширеним механізмом для побудови асинхронних систем обробки даних. Черги дозволяють згладжувати пікові навантаження, зберігаючи надлишкові повідомлення в буфері до моменту, коли обробні вузли зможуть їх прийняти. Це забезпечує стабільну роботу системи навіть при значних коливаннях трафіку. Провідні реалізації систем черг, такі як RabbitMQ, Apache Kafka та Amazon SQS, пропонують різні гарантії

доставки, механізми персистентності та можливості масштабування. Вибір конкретної технології залежить від вимог до швидкості обробки, надійності, обсягів даних та бюджету проекту. Однак використання черг вносить додаткову затримку в ланцюжок обробки повідомлень, що може бути критичним для додатків реального часу.

Проблема втрати повідомлень під час пікових навантажень тісно пов'язана з відсутністю механізмів контролю потоку (flow control) та обмеження швидкості (rate limiting). Якщо вхідний потік повідомлень перевищує пропускну здатність системи, черги можуть переповнюватися, а нові запити відхилятися без обробки. Для запобігання таким ситуаціям застосовуються стратегії дроселювання (throttling), що обмежують кількість прийнятих запитів за одиницю часу. Це дозволяє захистити критичні компоненти від перевантаження, але може призвести до відмови в обслуговуванні легітимних користувачів. Альтернативним підходом є пріоритизація повідомлень, коли система в першу чергу обробляє найважливіші запити, відкладаючи менш критичні на пізніший час. Реалізація таких механізмів вимагає глибокого розуміння бізнес-логіки та передбачення можливих сценаріїв використання.

Моніторинг та аналіз продуктивності є критично важливими для своєчасного виявлення проблем та прийняття рішень щодо масштабування інфраструктури. Сучасні системи обробки SMS повинні збирати метрики про кількість оброблених повідомлень, час відгуку, відсоток помилок, завантаженість процесорів, пам'яті та мережі. Централізоване логування та трасування запитів дозволяють швидко ідентифікувати вузькі місця та аномалії в роботі системи. Інструменти, такі як Prometheus, Grafana та ELK Stack, надають зручні інтерфейси для візуалізації метрик та налаштування алертів. Однак надмірний моніторинг може сам по собі стати джерелом додаткового навантаження, тому важливо знаходити баланс між деталізацією даних та продуктивністю системи. Автоматизація процесів реагування на інциденти

через системи оркестрації та автомасштабування значно знижує вплив людського фактора.

Забезпечення відмовостійкості (fault tolerance) є однією з головних вимог до систем обробки критично важливих повідомлень. Відмова окремих компонентів не повинна призводити до повної недоступності сервісу або втрати даних. Для досягнення цієї мети застосовуються методи реплікації, резервування та розподілу навантаження між географічно розподіленими центрами обробки даних. Механізми автоматичного перемикання на резервні вузли (failover) дозволяють швидко відновлювати функціональність після збоїв. Однак підтримка консистентності даних між репліками в розподілених системах є нетривіальною задачею, особливо в умовах мережевих розділень (network partitions). Теорема CAP стверджує, що неможливо одночасно гарантувати консистентність, доступність та стійкість до розділень, тому розробники повинні свідомо вибирати компроміси відповідно до специфіки застосування.

Еволюція хмарних технологій та контейнеризації відкрила нові можливості для побудови масштабованих та відмовостійких систем обробки SMS. Платформи, такі як Kubernetes, надають механізми автоматичного розгортання, масштабування та управління контейнеризованими додатками. Використання хмарних сервісів дозволяє динамічно адаптувати обсяг обчислювальних ресурсів до поточного навантаження, оплачуючи лише фактично використані ресурси. Це особливо актуально для систем із нерівномірним розподілом трафіку протягом дня або року. Однак перехід на хмарну інфраструктуру вимагає переосмислення архітектури додатків та впровадження практик DevOps для забезпечення безперервної інтеграції та доставки. Крім того, зберігання критичних даних у публічних хмарах може викликати питання щодо безпеки та дотримання регуляторних вимог.

Безпека комунікаційних систем є не менш важливим аспектом, що впливає на вибір архітектури та методів обробки повідомлень. SMS-повідомлення часто

містять конфіденційну інформацію, таку як одноразові паролі, коди підтвердження транзакцій або особисті дані користувачів. Захист від несанкціонованого доступу, підробки повідомлень та витоку інформації вимагає впровадження криптографічних методів, механізмів автентифікації та авторизації. Крім того, системи повинні бути захищені від атак типу відмови в обслуговуванні (DDoS), що можуть паралізувати інфраструктуру шляхом генерації величезної кількості фальшивих запитів. Використання систем виявлення вторгнень, брандмауерів та методів фільтрації трафіку допомагає мінімізувати ризики, однак може вносити додаткові затримки в обробку легітимних повідомлень [3].

Стандартизація протоколів та інтерфейсів взаємодії між компонентами комунікаційних систем спрощує інтеграцію різнорідних технологій та забезпечує сумісність між системами різних виробників. Протокол SMPP (Short Message Peer-to-Peer) є де-факто стандартом для обміну SMS-повідомленнями між телекомунікаційними операторами та додатками третіх сторін. Використання SMPP дозволяє абстрагуватися від специфіки конкретних операторів та будувати універсальні рішення для масової розсилки. Однак протокол має обмеження щодо швидкості передачі та підтримки сучасних форматів повідомлень, таких як Rich Communication Services (RCS). Альтернативні підходи на основі HTTP API надають більшу гнучкість та простоту інтеграції, але можуть не підтримуватися всіма операторами зв'язку. Вибір протоколу взаємодії повинен враховувати вимоги до продуктивності, надійності та сумісності з існуючою інфраструктурою.

Тестування продуктивності та стрес-тестування є обов'язковими етапами перед впровадженням системи обробки SMS у промислову експлуатацію. Симуляція пікових навантажень дозволяє виявити вузькі місця, оцінити граничну пропускну здатність та налаштувати параметри системи для оптимальної роботи. Інструменти навантажувального тестування, такі як JMeter,

Gatling або Locust, дозволяють генерувати реалістичні сценарії використання та аналізувати поведінку системи під різними умовами. Важливо проводити тестування не лише на етапі розробки, але й регулярно під час експлуатації для виявлення деградації продуктивності внаслідок змін у коді або інфраструктурі. Результати тестів використовуються для прийняття рішень щодо необхідності масштабування, оптимізації алгоритмів або модернізації обладнання.

Вартість утримання високонавантаженої системи обробки SMS включає не лише витрати на обладнання та хмарні сервіси, але й затрати на персонал, підтримку, оновлення програмного забезпечення та дотримання регуляторних вимог. Економічна ефективність архітектурних рішень оцінюється через співвідношення капітальних та операційних витрат до отриманих бізнес-результатів. Монолітні системи можуть мати нижчі початкові витрати на розробку, але виявляються дорожчими в довгостроковій перспективі через складність масштабування та підтримки. Мікросервісна архітектура, навпаки, вимагає значних інвестицій на початкових етапах, але забезпечує гнучкість та можливість оптимізації ресурсів відповідно до реальних потреб. Розрахунок загальної вартості володіння (TCO) повинен враховувати весь життєвий цикл системи від проектування до виведення з експлуатації.

Міграція з монолітної архітектури на розподілену або мікросервісну є складним процесом, що вимагає ретельного планування та поетапного впровадження. Повна заміна існуючої системи за один раз несе значні ризики простою сервісу та втрати даних. Тому частіше застосовується стратегія поступового виділення окремих функціональних блоків у незалежні сервіси зі збереженням можливості взаємодії з монолітом. Цей підхід, відомий як Strangler Fig Pattern, дозволяє мінімізувати ризики та поступово нарощувати функціональність нової системи. На перехідному етапі необхідно забезпечити сумісність інтерфейсів, синхронізацію даних та можливість повернення до попередньої версії у разі виникнення критичних проблем. Досвід успішних

міграцій показує, що тривалість процесу може становити від кількох місяців до декількох років залежно від масштабу системи.

Інтеграція систем обробки SMS з іншими корпоративними додатками, такими як CRM, системи аналітики, платіжні шлюзи та соціальні мережі, розширює можливості автоматизації бізнес-процесів та покращує клієнтський досвід. Уніфіковані API та стандартизовані формати обміну даними спрощують інтеграцію та знижують витрати на розробку. Використання мікросервісної архітектури з чітко визначеними контрактами взаємодії дозволяє різним командам незалежно розробляти та впроваджувати нові функції без ризику порушення роботи суміжних систем. Однак збільшення кількості інтеграцій підвищує складність управління залежностями та моніторингу, що вимагає впровадження централізованих інструментів оркестрації та трасування запитів між сервісами.

Регуляторні вимоги та стандарти індустрії накладають додаткові обмеження на архітектуру та процеси обробки SMS-повідомлень. Законодавство про захист персональних даних, таке як GDPR в Європі або CCPA в Каліфорнії, вимагає забезпечення конфіденційності користувачів, надання можливостей для видалення даних та звітування про витоки інформації. Телекомунікаційні регулятори можуть встановлювати вимоги до надійності сервісу, швидкості доставки повідомлень та процедур обробки скарг. Дотримання цих вимог вимагає впровадження механізмів аудиту, шифрування даних як у спокої, так і під час передачі, а також регулярного проведення перевірок безпеки. Недотримання регуляторних вимог може призвести до значних штрафів та репутаційних втрат.

Сучасні тенденції розвитку комунікаційних систем включають широке впровадження штучного інтелекту та машинного навчання для оптимізації обробки повідомлень, виявлення аномалій та прогнозування навантаження. Алгоритми машинного навчання дозволяють автоматично класифікувати

повідомлення за пріоритетом, виявляти спам та шахрайські схеми, а також персоналізувати взаємодію з користувачами. Прогнозна аналітика допомагає передбачати пікові навантаження та завчасно масштабувати інфраструктуру для забезпечення стабільної роботи сервісу. Однак впровадження ШІ вимагає наявності великих обсягів навчальних даних, обчислювальних ресурсів для тренування моделей та експертизи в галузі аналізу даних. Крім того, необхідно враховувати етичні аспекти використання ШІ, такі як можливість дискримінації користувачів або непрозорість прийняття рішень.

Підсумовуючи, архітектура електронних комунікаційних систем обробки SMS еволюціонує від монолітних до розподілених та мікросервісних рішень під впливом зростаючих вимог до масштабованості, надійності та швидкості обробки. Проблема пікових навантажень залишається актуальною та вимагає комплексного підходу, що включає вибір оптимальної моделі взаємодії між компонентами, впровадження механізмів чергування та обмеження швидкості, забезпечення відмовостійкості та безпеки [4]. Сучасні технології, такі як хмарні платформи, контейнеризація, системи черг та інструменти моніторингу, надають широкий спектр можливостей для побудови ефективних рішень. Однак вибір конкретної архітектури повинен ґрунтуватися на глибокому аналізі бізнес-вимог, технічних обмежень, бюджету та перспектив розвитку системи.

1.2. Аналіз мікросервісної архітектури та способів міжсервісної комунікації в телекомунікаційних системах

Мікросервісна архітектура є сучасним підходом до побудови складних програмних систем, що передбачає розподіл функціональності на набір невеликих незалежних сервісів. Кожен мікросервіс відповідає за конкретну бізнес-функцію і може розроблятися, розгортатися та масштабуватися незалежно від інших компонентів системи. Такий підхід забезпечує високу

гнучкість розробки та можливість використання різних технологічних стеків для різних сервісів. У контексті телекомунікаційних систем мікросервісна архітектура дозволяє ефективно розподіляти навантаження та забезпечувати відмовостійкість критично важливих компонентів обробки SMS-повідомлень.

Основною перевагою мікросервісної архітектури порівняно з монолітною є можливість незалежного масштабування окремих компонентів системи. Якщо певний сервіс обробки повідомлень стикається з підвищеним навантаженням, можна збільшити кількість його екземплярів без необхідності масштабування всієї системи. Це дозволяє оптимально використовувати обчислювальні ресурси та знижувати операційні витрати. Ізоляція сервісів забезпечує кращу відмовостійкість, оскільки збій одного компонента не призводить до відмови всієї системи. Можливість використання різних мов програмування та технологій для різних сервісів дозволяє обирати найбільш придатні інструменти для вирішення конкретних завдань.

Впровадження мікросервісної архітектури супроводжується певними викликами, що стосуються складності управління розподіленою системою. Зростає кількість компонентів, які потребують моніторингу, оркестрації та координації між собою. Необхідно забезпечити ефективну взаємодію між сервісами, що вимагає ретельного проектування інтерфейсів та протоколів комунікації. Проблеми мережевої взаємодії, такі як затримки, часткові відмови та втрата повідомлень, стають більш актуальними в розподіленому середовищі [5, с. 299-305]. Підтримка консистентності даних між різними сервісами потребує застосування складних патернів розподіленого програмування, що підвищує загальну складність системи.

Організація мікросервісів у телекомунікаційних системах зазвичай відбувається навколо бізнес-можливостей та функціональних доменів системи. Типова система обробки SMS може включати сервіс прийому повідомлень, сервіс валідації та фільтрації, сервіс маршрутизації, сервіс доставки та сервіс

зберігання історії. Кожен з цих сервісів має чітко визначену відповідальність та мінімальні залежності від інших компонентів системи. Розмежування відповідальності дозволяє різним командам розробників працювати над окремими сервісами паралельно, що прискорює процес розробки та впровадження нових функцій. Такий розподіл також спрощує розуміння системи та зменшує ризик виникнення помилок при внесенні змін до коду.

Синхронна комунікація між мікросервісами зазвичай реалізується через HTTP-протокол за допомогою RESTful API або gRPC фреймворку. REST API використовує стандартні HTTP-методи для виконання операцій над ресурсами і є найбільш поширеним підходом завдяки простоті реалізації та універсальності використання. Сервіси обмінюються JSON або XML документами, що забезпечує легку інтеграцію різних компонентів та читабельність повідомлень для розробників. Однак синхронний підхід створює жорсткі залежності між сервісами, оскільки клієнт має очікувати на відповідь від сервера перед продовженням роботи. При синхронній взаємодії затримки накопичуються в ланцюжку викликів, що може призвести до погіршення загальної продуктивності системи під час високого навантаження.

gRPC є сучасною альтернативою REST, що базується на Protocol Buffers для серіалізації даних та HTTP/2 протоколі для передачі. Цей підхід забезпечує більш ефективну серіалізацію даних порівняно з JSON та підтримує двонаправлену потокову передачу інформації. gRPC автоматично генерує клієнтський та серверний код на основі IDL-файлів, що зменшує ймовірність помилок інтеграції між сервісами. Продуктивність gRPC зазвичай вища порівняно з REST завдяки компактному бінарному формату даних та можливостям мультиплексування HTTP/2 [6]. Проте складність початкового налаштування та відлагодження може бути вищою, а читабельність повідомлень нижчою через використання бінарного формату замість текстового.

Асинхронна комунікація базується на обміні повідомленнями через спеціалізовані брокери або черги повідомлень у системі. Популярні рішення включають Apache Kafka, RabbitMQ, Apache ActiveMQ та хмарні сервіси на кшталт Amazon SQS. При асинхронній взаємодії відправник не очікує негайної відповіді від отримувача, що дозволяє сервісам працювати незалежно один від одного та знижує зв'язаність компонентів у системі. Черги повідомлень виступають буфером, який згладжує пікові навантаження на систему та захищає сервіси-споживачі від перевантаження запитами. Асинхронний підхід особливо корисний для обробки подій та виконання довготривалих операцій, що не потребують негайної відповіді користувачу.

Apache Kafka є розподіленою платформою потокової обробки даних, що забезпечує високу пропускну здатність та надійне зберігання повідомлень на диску. Kafka організує повідомлення у топіки, що дозволяє логічно групувати події за типом або призначенням в системі. Механізм партиціонування забезпечує горизонтальне масштабування системи та паралельну обробку повідомлень багатьма споживачами одночасно. Kafka гарантує упорядкованість повідомлень у межах однієї партиції, що критично важливо для багатьох сценаріїв обробки SMS, де послідовність має значення. Висока доступність досягається через реплікацію даних між кількома брокерами кластера, що забезпечує стійкість до відмов окремих вузлів.

RabbitMQ реалізує протокол AMQP та надає гнучку маршрутизацію повідомлень через систему обмінників та черг з різними політиками. Різні типи обмінників дозволяють реалізувати складні сценарії доставки повідомлень, включаючи широкомовну розсилку, маршрутизацію на основі ключів та тематичну маршрутизацію. RabbitMQ підтримує підтвердження доставки повідомлень на рівні клієнта, що дозволяє гарантувати обробку кожного повідомлення принаймні один раз. Система пріоритетів черг дозволяє забезпечити обробку критично важливих повідомлень у першу чергу перед

менш важливими. Однак пропускна здатність RabbitMQ може бути нижчою порівняно з Kafka при дуже високих навантаженнях через архітектурні особливості.

Вибір між синхронною та асинхронною комунікацією залежить від конкретних вимог системи та характеристик оброблюваних операцій. Синхронна взаємодія краще підходить для операцій, що потребують негайної відповіді, таких як валідація даних користувача або запити довідкової інформації. Асинхронна комунікація ефективніша для подієво-орієнтованих сценаріїв, обробки великих обсягів даних та виконання завдань, що не потребують негайного результату для користувача. Часто в одній системі комбінують обидва підходи, використовуючи синхронні виклики для критичних операцій та асинхронні повідомлення для фонових задач обробки. Правильний вибір типу комунікації значно впливає на продуктивність, надійність та складність підтримки всієї системи.

Патерн API Gateway забезпечує єдину точку входу для клієнтських додатків, що взаємодіють з мікросервісною системою обробки повідомлень. Gateway виконує маршрутизацію запитів до відповідних сервісів, агрегацію відповідей від кількох сервісів в один результат та перетворення протоколів за потреби. Цей компонент також відповідає за автентифікацію користувачів, авторизацію доступу до ресурсів та обмеження швидкості надходження запитів. Використання API Gateway спрощує клієнтський код додатків та дозволяє змінювати внутрішню структуру системи без впливу на зовнішні інтерфейси, що використовуються клієнтами. Проте Gateway може стати вузьким місцем продуктивності всієї системи, тому його необхідно проектувати з урахуванням високої доступності та можливості горизонтального масштабування.

Сервісна сітка є інфраструктурним шаром, що керує взаємодією між мікросервісами без необхідності внесення змін у код самих застосунків. Популярні реалізації, такі як Istio та Linkerd, забезпечують автоматичне

балансування навантаження між екземплярами сервісів, шифрування трафіку, збір метрик моніторингу та управління політиками безпеки в системі. Сервісна сітка використовує sidecar-контейнери, що розгортаються поруч з кожним сервісом та перехоплюють весь мережевий трафік для застосування необхідних політик. Це дозволяє централізовано управляти різними аспектами міжсервісної комунікації, такими як повторні спроби при відмовах, тайм-аути та автоматичні вимикачі для запобігання каскадним відмовам. Впровадження сервісна сітка додає певної складності до загальної інфраструктури системи, але значно спрощує розробку та експлуатацію розподілених мікросервісних застосунків (табл. 1.2).

Таблиця 1.2

Порівняння методів міжсервісної комунікації

Метод комунікації	**Переваги**	**Недоліки**	**Випадки використання**
Синхронний HTTP/REST	Простота реалізації, широка підтримка	Затримки, складність обробки помилок	CRUD операції, запит-відповідь
Асинхронний HTTP	Відсутність блокування, кращий час відгуку	Складність налагодження	Довготривалі операції
Apache Kafka	Висока пропускна здатність, персистентність	Складність налаштування	Event-driven системи, потоки даних
gRPC	Висока продуктивність, типізація	Обмежена підтримка браузерів	Міжсервісна комунікація
Message Queue (RabbitMQ)	Гарантія доставки, гнучкість маршрутизації	Необхідність окремого сервера	Асинхронна обробка задач

Виявлення сервісів є критично важливим механізмом у мікросервісній архітектурі, що дозволяє сервісам динамічно знаходити один одного в мережі. Service discovery може бути реалізований через спеціалізовані інструменти, такі як Consul, Eureka або etcd, що підтримують централізований реєстр. Ці системи

підтримують актуальний реєстр доступних екземплярів сервісів та їхніх мережевих адрес для встановлення з'єднань. Механізми перевірки здоров'я регулярно моніторять доступність кожного екземпляра сервісу та автоматично видаляють недоступні екземпляри з реєстру для запобігання помилкам. Клієнтське або серверне балансування навантаження використовує інформацію з service discovery для інтелектуального розподілу запитів між доступними екземплярами сервісів у кластері. Динамічне виявлення сервісів забезпечує еластичність системи та значно спрощує процес автоматизованого розгортання та масштабування компонентів.

Забезпечення консистентності даних у розподілених мікросервісних системах є складним завданням, оскільки кожен сервіс зазвичай має власну ізольовану базу даних. Патерн Saga дозволяє реалізувати розподілені транзакції через послідовність локальних транзакцій, кожна з яких має відповідну компенсуючу операцію для відкату змін. У разі відмови одного з кроків послідовності, система виконує компенсуючі транзакції для відкату всіх попередніх змін та повернення до консистентного стану. Event sourcing є альтернативним підходом, що зберігає всі зміни стану системи як незмінну послідовність подій, що дозволяє відновити стан системи в будь-який момент часу з історії. CQRS розділяє операції читання та запису на різні моделі, оптимізуючи кожен тип операцій окремо відповідно до специфічних вимог. Ці патерни додають певної архітектурної складності до системи, але забезпечують необхідний рівень надійності та консистентності даних у розподіленому середовищі. Моніторинг та спостережуваність мікросервісних систем потребують спеціалізованих інструментів для збору та аналізу розподілених метрик, логів та трейсів з усіх компонентів. Централізоване логування через ELK Stack або аналогічні платформи дозволяє агрегувати логи з усіх сервісів у єдине сховище та ефективно шукати інформацію про проблеми в системі. Розподілене трасування за допомогою інструментів Jaeger або Zipkin допомагає

відстежувати повний шлях запиту через множину сервісів та ідентифікувати вузькі місця продуктивності [7]. Системи збору метрик, такі як Prometheus, забезпечують безперервний моніторинг продуктивності та здоров'я кожного компонента системи в режимі реального часу. Дашборди візуалізації в Grafana надають операторам системи єдиний зручний погляд на стан всієї розподіленої інфраструктури. Належна спостережуваність системи є критично важливою для швидкої діагностики проблем та їх вирішення у виробничому середовищі без тривалих простоїв.

Безпека мікросервісних систем вимагає комплексного підходу, що охоплює автентифікацію користувачів, авторизацію доступу та шифрування даних на всіх рівнях системи. OAuth 2.0 та OpenID Connect є стандартними протоколами для делегованої авторизації та безпечної автентифікації користувачів у розподілених системах. JSON Web Tokens широко використовуються для безпечної передачі інформації про ідентичність та права доступу між різними сервісами системи. Mutual TLS забезпечує надійне шифрування з'єднань та взаємну автентифікацію на рівні транспорту між всіма сервісами в мережі. Політики безпеки можуть централізовано управлятися через service mesh або спеціалізовані security gateway для забезпечення єдиного підходу. Регулярний аудит безпеки та автоматизоване сканування вразливостей допомагають своєчасно виявляти та усувати потенційні загрози безпеці системи. Принцип найменших привілеїв повинен неухильно застосовуватися при наданні сервісам доступу до ресурсів та даних інших компонентів.

Контейнеризація та оркестрація є фундаментальними технологіями для ефективного розгортання та управління мікросервісними застосунками в сучасних умовах. Docker забезпечує стандартизований та переносимий спосіб пакування сервісів разом з усіма їхніми залежностями в ізольовані контейнери (рис.1.1-1.2).

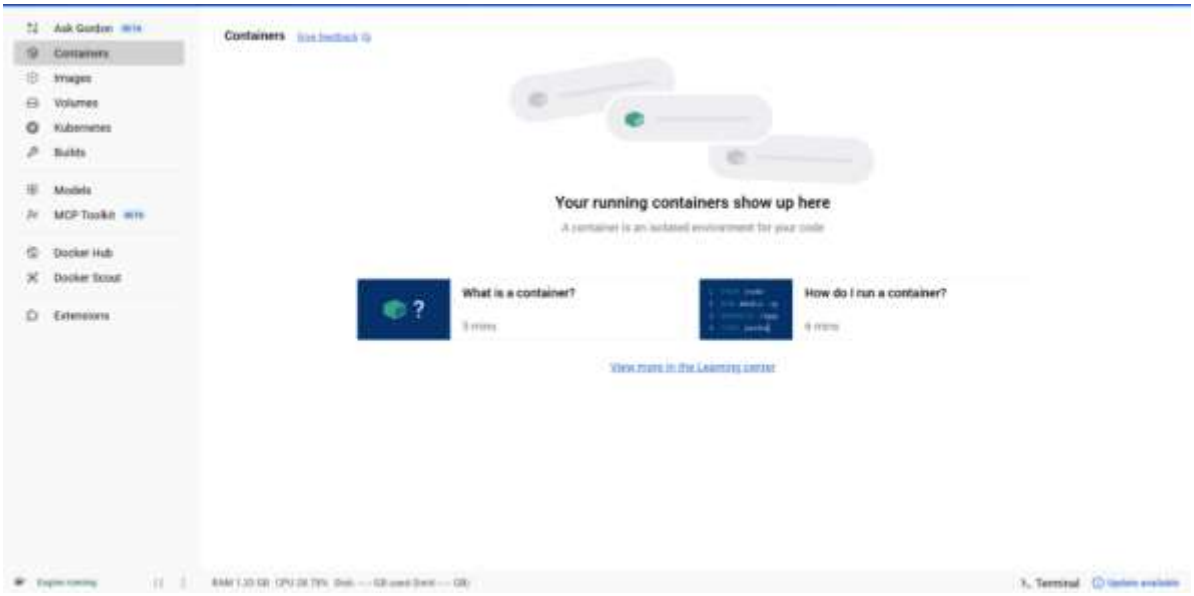

Рис. 1.1 – Інтерфейс Docker Desktop з відображенням розділу Containers у стані без запущених контейнерів

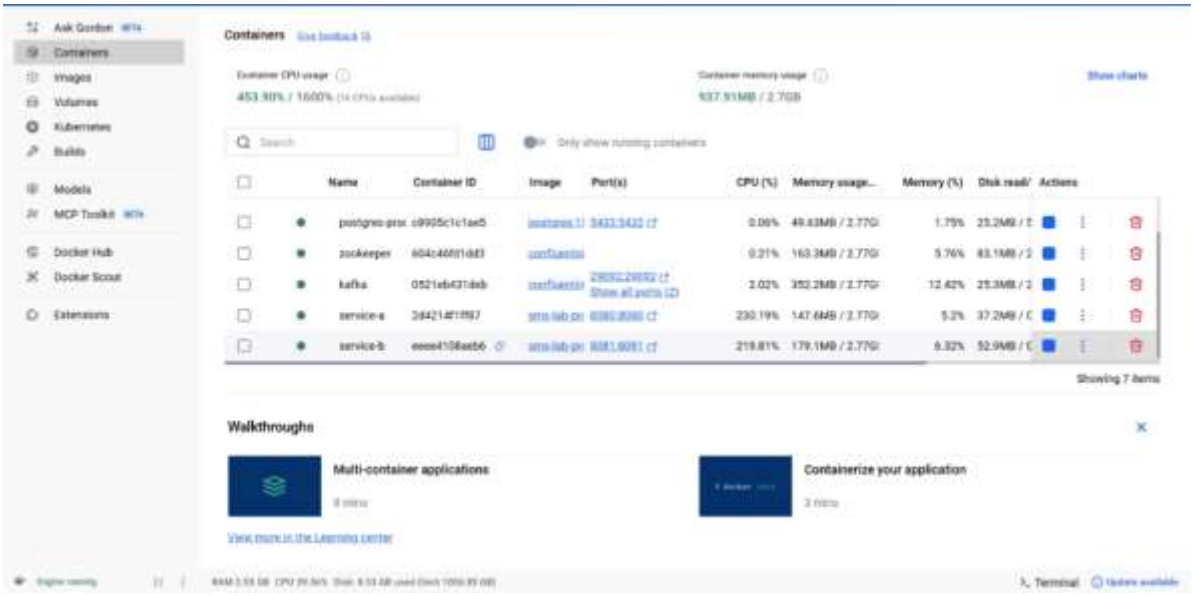

Рис. 1.2 – Інтерфейс Docker Desktop із відображенням активних контейнерів мікросервісної системи, що демонструє використання PostgreSQL, Zookeeper, Kafka та двох сервісів у середовищі контейнеризації

Kubernetes є фактичним стандартом оркестрації контейнерів, що автоматизує розгортання, масштабування та повсякденне управління

контейнеризованими застосунками в кластері. Kubernetes підтримує декларативну конфігурацію всієї інфраструктури, що дозволяє чітко описувати бажаний стан системи у вигляді версійованих маніфестів. Механізми самовідновлення автоматично перезапускають недоступні контейнери та переносять їх на здорові вузли кластера без ручного втручання операторів. Горизонтальне автомасштабування дозволяє динамічно змінювати кількість запущених екземплярів сервісів залежно від поточного рівня навантаження на систему. Використання Kubernetes та екосистеми навколо нього значно спрощує повсякденну експлуатацію навіть дуже складних мікросервісних систем з десятками та сотнями компонентів.

1.3. Порівняльний аналіз синхронних та асинхронних методів передавання даних у розподілених системах

Синхронна та асинхронна моделі комунікації представляють два фундаментально різні підходи до організації взаємодії між окремими компонентами розподілених систем обробки даних. Синхронна комунікація передбачає, що клієнт відправляє запит до сервера та блокується в очікуванні отримання відповіді перед продовженням подальшого виконання своєї роботи. Асинхронна модель взаємодії дозволяє клієнту продовжувати виконання своїх задач одразу після відправлення запиту, не чекаючи на миттєву відповідь від сервера. Вибір між цими двома фундаментально різними підходами суттєво впливає на загальну архітектуру системи, її продуктивність під навантаженням та складність розробки й підтримки. Глибоке розуміння переваг та недоліків кожного методу комунікації є критично важливим для прийняття обґрунтованих архітектурних рішень при проектуванні систем обробки SMS-повідомлень (табл. 1.3).

Таблиця 1.3

Характеристики синхронних та асинхронних протоколів передачі даних

Параметр	**Синхронна комунікація**	**Асинхронна комунікація**
Час відгуку	Залежить від швидкості обробки	Незалежний, миттєвий
Зв'язаність сервісів	Висока (tight coupling)	Низька (loose coupling)
Обробка помилок	Пряма (immediate)	Відкладена (delayed)
Пропускна здатність	1000-5000 req/sec	10000-100000 msg/sec
Гарантія доставки	Відразу або помилка	At-least-once, exactly-once
Складність реалізації	Низька	Середня/Висока

Синхронні HTTP-запити є найпростішим та найбільш широко використовуваним способом комунікації у сучасних веб-системах та застосунках. Клієнт встановлює TCP-з'єднання з сервером, відправляє HTTP-запит з необхідними параметрами та очікує на отримання повної відповіді перед продовженням подальшої роботи. Ця модель взаємодії природно відображає звичну семантику виклику функцій у традиційному програмуванні та інтуїтивно зрозуміла більшості розробників. Синхронна взаємодія забезпечує просту та зрозумілу обробку помилок, оскільки клієнт негайно отримує чітку інформацію про успішність або невдачу виконаної операції на сервері. Однак при високому рівні навантаження на систему синхронні виклики можуть призвести до швидкого вичерпання пулу доступних з'єднань та блокування потоків виконання, що серйозно обмежує загальну пропускну здатність всієї системи обробки.

Основною проблемою синхронної комунікації між сервісами є створення каскадних залежностей у ланцюжках викликів між різними компонентами системи. Якщо сервіс A синхронно викликає сервіс B, який у свою чергу синхронно викликає сервіс C для виконання операції, то будь-яка затримка на довільному рівні цього ланцюжка неминуче накопичується та негативно впливає

на загальний час відповіді користувачу. При повній відмові або тривалій недоступності одного з сервісів у цьому ланцюжку вся складна операція може зазнати невдачі та повернути помилку. Небезпечний ефект доміно може призвести до каскадних відмов, коли локальні проблеми в одному компоненті швидко поширюються на всю систему. Для ефективної мітигації цих серйозних ризиків необхідно обов'язково впроваджувати перевірені патерни стійкості, такі як circuit breaker для запобігання каскадним відмовам, налаштування розумних тайм-аут та продумана стратегія повторити спробу з експоненційним відступом між спробами [8]. Проте навіть з усіма цими захисними механізмами синхронна архітектура залишається більш вразливою до різноманітних часткових відмов компонентів порівняно з асинхронним підходом.

Затримки у синхронній комунікації мають пряму кумулятивну природу, що особливо проявляється при побудові складних систем з багатьма рівнями. Якщо обробка одного користувацького запиту вимагає послідовного виконання п'яти синхронних викликів до різних сервісів, кожен з яких у середньому займає 100 мілісекунд чистого часу виконання, то загальна затримка відповіді становитиме щонайменше 500 мілісекунд плюс додаткові накладні витрати на мережеву взаємодію. У складних системах з глибокою ієрархією викликів між багатьма сервісами це може призвести до категорично неприйнятно довгого часу відповіді для кінцевого користувача системи. Крім того, природна варіативність затримок кожного окремого виклику збільшується на кожному наступному рівні ієрархії, що значно ускладнює точне прогнозування продуктивності всієї системи під навантаженням. Проблема затримка, коли невелика, але помітна частка запитів виконується в рази довше за середній час, може суттєво погіршити загальний користувацький досвід роботи з системою. Ефективна оптимізація продуктивності складних синхронних систем часто вимагає агресивної мінімізації загальної кількості міжсервісних викликів у ланцюжку та впровадження багаторівневого кешування проміжних результатів.

Асинхронна комунікація через спеціалізовані брокери повідомлень принципово розв'язує фундаментальну проблему жорсткого зв'язування між окремими сервісами розподіленої системи. Відправник публікує своє повідомлення у відповідну чергу або тематичний топік, абсолютно не знаючи про конкретних поточних отримувачів цього повідомлення та не очікуючи на отримання будь-якої негайної відповіді від них. Це дозволяє різним сервісам працювати з принципово різною швидкістю обробки та бути повністю незалежними у часі виконання своїх операцій. Черга повідомлень природно виступає ефективним буфером, що м'яко згладжує короткочасні пікові навантаження на систему та надійно захищає сервіси-споживачі від критичного перевантаження великою кількістю одночасних запитів. Якщо один із ключових сервісів системи стає тимчасово недоступним через технічні проблеми, повідомлення безпечно залишаються збереженими у стійкій черзі до моменту його успішного відновлення та повернення до роботи. Така модель взаємодії значно підвищує загальну відмовостійкість розподіленої системи та її природну еластичність при змінах навантаження.

Можливість паралелізму обробки повідомлень є ключовою перевагою правильно спроектованої асинхронної архітектури системи обробки даних. Один сервіс-відправник може за короткий час опублікувати сотні або навіть тисячі повідомлень у чергу, які потім будуть ефективно оброблятися паралельно великою кількістю незалежних екземплярів сервісу-споживача одночасно. Це дозволяє системі досягти принципово вищої пропускної здатності обробки порівняно з обмеженнями традиційного синхронного підходу з послідовною обробкою. Просте горизонтальне масштабування кількості споживачів є інтуїтивно зрозумілим і надзвичайно ефективним способом лінійного збільшення загальної продуктивності системи під зростаючим навантаженням. Кожен окремий споживач може обробляти доручені йому повідомлення у власному комфортному темпі без будь-якого негативного впливу на швидкість

роботи інших незалежних компонентів системи. Вбудовані механізми інтелектуального партиціонування у сучасних брокерах повідомлень, таких як Apache Kafka, дозволяють ефективно розподілити навантаження обробки між множиною паралельних споживачів при обов'язковому збереженні критично важливої упорядкованості повідомлень у межах кожної логічної партиції даних [9]. Надійність гарантованої доставки повідомлень у асинхронних системах забезпечується через вбудовані механізми персистентності на диску та систему підтверджень обробки. Сучасні брокери повідомлень надійно зберігають всі прийняті повідомлення на постійному диску, що категорично гарантує їх збереження навіть у випадку раптової повної відмови окремого вузла кластера. Сервіси-споживачі явно підтверджують факт успішної обробки кожного отриманого повідомлення, після чого брокер може безпечно видалити оброблене повідомлення або позначити його як повністю оброблене в системі. Якщо споживач не надсилає очікуване підтвердження обробки протягом заданого розумного часу очікування, те саме повідомлення може бути автоматично повторно доставлене для обробки іншому доступному споживачу. Спеціалізовані механізми dead letter queue дозволяють безпечно ізолювати проблемні повідомлення, які неодноразово не змогли бути успішно оброблені жодним споживачем, для подальшого детального аналізу причин. Така висока надійність доставки є особливо важливою для критично важливих бізнес-операцій, таких як обробка комерційних SMS-повідомлень, де будь-яка втрата цінних даних є категорично неприйнятною для бізнесу.

Підвищена складність асинхронних систем чітко проявляється у необхідності впровадження додаткових компенсуючих механізмів для надійного забезпечення консистентності даних у розподіленій системі. Оскільки різні операції виконуються повністю незалежно одна від одної у часі та можуть завершитися з принципово різними результатами, закономірно виникає складна потреба в розподілених транзакціях або застосуванні патернів кінцева

узгодженість для досягнення узгодженого стану [10-11]. Відстеження поточного стану складної розподіленої операції вимагає впровадження додаткової спеціалізованої інфраструктури, такої як механізми робочих процесів або складні державні машини для координації кроків. Відлагодження асинхронних систем є значно складнішим завданням, оскільки потік виконання операцій не є простим лінійним та послідовним, а розгалужується між множиною компонентів. Для ефективного моніторингу та діагностики проблем необхідно обов'язково впроваджувати унікальні correlation IDs, що дозволяють надійно відстежувати логічний зв'язок між різними повідомленнями та подіями в системі. Якісна візуалізація складних потоків даних між компонентами стає критично важливою для глибокого розуміння реальної поведінки розподіленої системи під навантаженням.

Детальне порівняння пропускної здатності двох підходів чітко демонструє значні переваги асинхронного підходу при обробці високих навантажень на систему. Синхронна система є принципово обмеженою максимальною кількістю одночасних з'єднань та робочих потоків, що може ефективно обробляти один сервер з наявними ресурсами. Типовий добре налаштований веб-сервер може ефективно обслуговувати десятки або в кращому випадку сотні тисяч одночасних активних з'єднань без деградації продуктивності [12, с. 64-78]. Асинхронна система з правильно підібраним брокером повідомлень може без проблем обробляти мільйони окремих повідомлень на секунду за рахунок масованої паралельної обробки великою кількістю незалежних споживачів одночасно. Apache Kafka, наприклад, здатна стабільно підтримувати вражаючу пропускну здатність понад мільйон повідомлень на секунду навіть на одному середньому за розміром кластері з декількох серверів. Ця драматична різниця у продуктивності стає особливо критичною для реальних систем обробки SMS, де можуть регулярно виникати різкі непередбачувані пікові навантаження під час

проведення масових комерційних розсилок або інформування у надзвичайних ситуаціях.

Латентність окремих операцій є специфічною областю, де традиційна синхронна комунікація часто демонструє помітну перевагу для обробки одиничних ізольованих операцій. Прямий HTTP-запит до сервера може бути успішно виконаний за одиниці або в гіршому випадку десятки мілісекунд при низькому поточному навантаженні на систему та добрій мережевій з'єднаності. Асинхронна обробка через проміжний брокер повідомлень неминуче додає певну додаткову затримку на етапи публікації повідомлення відправником, надійного зберігання у персистентній черзі та подальшого отримання призначеним споживачем для обробки. Для високоінтерактивних операцій, де кінцевий користувач безпосередньо очікує на швидку відповідь від системи, синхронний підхід взаємодії може виявитися більш прийнятним з точки зору користувацького досвіду [13]. Однак при стабільно високому рівні навантаження на систему ситуація кардинально змінюється на протилежну - синхронні системи починають страждати від прогресуючого зростання затримок через поступове виснаження доступних системних ресурсів, тоді як добре спроектовані асинхронні системи здатні підтримувати стабільну передбачувану латентність завдяки ефективній буферизації навантаження в чергах. Остаточний вибір оптимального підходу залежить від конкретних вимог до максимально допустимого часу відповіді та реального характеру навантаження на систему.

Гібридні архітектури, що розумно комбінують синхронну та асинхронну комунікацію у різних частинах системи, часто виявляються оптимальним збалансованим рішенням для побудови складних практичних систем. Критичні операції користувача, що об'єктивно потребують швидкого негайного зворотного зв'язку, можуть ефективно використовувати синхронні HTTP API для мінімізації затримки відповіді. Фонові процеси обробки даних, пакетна обробка великих обсягів інформації та виконання операцій, що не вимагають негайного

результату для користувача, можуть природно виконуватися асинхронно через черги повідомлень. Наприклад, початковий прийом вхідного SMS-повідомлення від клієнта може відбуватися через швидкий синхронний API з миттєвою базовою валідацією формату, після чого прийняте повідомлення негайно поміщається у надійну чергу для подальшої асинхронної обробки та остаточної доставки отримувачу [14]. Такий продуманий гібридний підхід дозволяє майже миттєво відповісти клієнту про успішне прийняття його повідомлення до обробки, при цьому вся складна багатоетапна обробка ефективно виконується у фоновому режимі без блокування клієнта. Розумна комбінація обох підходів у різних частинах системи забезпечує оптимальний баланс між простотою розробки, високою продуктивністю під навантаженням та необхідною надійністю обробки критичних даних.

Обробка помилок у синхронних системах є відносно прямолінійним та інтуїтивно зрозумілим процесом - клієнт безпосередньо отримує код помилки та може негайно відреагувати відповідною логікою обробки. У асинхронних системах коректна обробка помилок вимагає впровадження більш складної та продуманої логіки з урахуванням відкладеного характеру обробки. Необхідно чітко визначити оптимальну стратегію автоматичних повторних спроб обробки, встановити розумну максимальну кількість дозволених спроб та продумати правильну поведінку системи при остаточній невдачі обробки повідомлення. Спеціалізовані dead letter queues дозволяють ефективно ізолювати хронічно проблемні повідомлення для подальшого детального ручного аналізу причин невдач розробниками. Складні компенсуючі транзакції можуть бути необхідні для коректного відкату частково виконаних змін у розподілених операціях при виникненні помилок. Системи моніторингу та автоматичного алертингу повинні безперервно відстежувати небезпечне накопичення великої кількості необроблених повідомлень у чергах або аномально високий рівень помилок обробки для швидкого реагування. Ці додаткові захисні механізми неминуче

збільшують загальну складність проектування та підтримки системи, але забезпечують життєво необхідну надійність для критичних бізнес-застосувань.

Тестування синхронних та асинхронних систем вимагає застосування принципово різних підходів, методологій та спеціалізованих інструментів для перевірки коректності. Синхронні API досить легко та швидко тестуються за допомогою традиційних перевірених інструментів HTTP-тестування та простих моків зовнішніх залежностей. Асинхронні системи об'єктивно потребують розробки значно складніших тестових сценаріїв, що правильно враховують принципи eventual consistency та можливу недетерміновану послідовність надходження подій у систему. Інтеграційні тести повинні ретельно перевіряти коректність обробки повідомлень у різних послідовностях, включаючи складні граничні випадки з дублікатами, затримками та частковими відмовами компонентів. Навантажувальне тестування асинхронних систем має симулювати реалістичні пікові навантаження для перевірки стабільності роботи черг, споживачів та брокерів під екстремальним тиском. Автоматизація всього процесу тестування через добре налаштовані CI/CD пайплайни забезпечує швидкий та об'єктивний зворотній зв'язок розробникам про якість внесених змін у код.

Вибір оптимального підходу до комунікації між компонентами має базуватися на ретельному аналізі специфічних вимог конкретної системи та характеристик оброблюваного навантаження. Синхронна комунікація залишається виправданим вибором для простих систем з передбачуваним помірним навантаженням, де важлива простота розробки та підтримки коду. Асинхронний підхід стає необхідним для високонавантажених систем, що потребують високої пропускної здатності, відмовостійкості та еластичного масштабування під змінним навантаженням. Для систем обробки SMS-повідомлень, що характеризуються різкими піковими навантаженнями під час масових розсилок, гібридна архітектура з асинхронною обробкою через надійні

черги повідомлень зазвичай забезпечує найкращий баланс продуктивності та надійності [15, с. 102-121]. Правильне розуміння компромісів між різними підходами дозволяє архітекторам приймати обґрунтовані рішення, що оптимально відповідають реальним потребам бізнесу та технічним обмеженням проекту.

Висновки

У першому розділі проведено комплексний аналіз сучасних підходів до побудови високонавантажених систем обробки SMS-повідомлень, що дозволило виявити ключові архітектурні рішення та методи комунікації для забезпечення надійної роботи під піковими навантаженнями.

Аналіз архітектур електронних комунікаційних систем показав, що традиційні монолітні рішення не здатні ефективно справлятися з різкими змінами навантаження, що характерні для систем обробки SMS. Виявлено, що проблеми пікових навантажень призводять до затримок у доставці повідомлень, втрати даних та перевантаження системних ресурсів. Розподілені системи з механізмами горизонтального масштабування та балансування навантаження демонструють значно кращу стійкість до перевантажень, хоча і вносять додаткову складність у процеси моніторингу та забезпечення консистентності даних.

Дослідження мікросервісної архітектури підтвердило її придатність для побудови високонавантажених телекомунікаційних систем завдяки можливості незалежного масштабування окремих компонентів. Виявлено, що ключовими перевагами мікросервісного підходу є ізоляція відмов, гнучкість технологічного стеку та можливість паралельної розробки різними командами. Проаналізовано основні способи міжсервісної комунікації, включаючи REST API, gRPC та брокери повідомлень. Встановлено, що вибір методу комунікації суттєво впливає на продуктивність, надійність та складність системи. Особливу увагу

приділено патернам API Gateway та Service Mesh, що спрощують управління взаємодією між сервісами. Порівняльний аналіз синхронних та асинхронних методів передавання даних виявив принципові відмінності у поведінці систем під навантаженням. Синхронна комунікація забезпечує простоту розробки та низьку латентність окремих операцій, але створює каскадні залежності та обмежує загальну пропускну здатність системи. Асинхронна взаємодія через черги повідомлень дозволяє досягти значно вищої пропускної здатності завдяки паралельній обробці та природній буферизації навантаження, що особливо важливо для систем обробки SMS з непередбачуваними піковими навантаженнями. Встановлено, що гібридні архітектури, що комбінують обидва підходи, забезпечують оптимальний баланс між швидкістю відповіді користувачу та загальною продуктивністю системи.

Результати аналізу показують, що для ефективної обробки SMS-повідомлень під піковими навантаженнями необхідно застосовувати мікросервісну архітектуру з асинхронною комунікацією через надійні брокери повідомлень, такі як Apache Kafka або RabbitMQ. Це створює теоретичну основу для проектування практичної системи обробки SMS, що буде детально розглянуто у наступних розділах роботи.

РОЗДІЛ 2

ПРОЕКТУВАННЯ СИСТЕМИ ОБРОБКИ SMS НА БАЗІ МІКРОСЕРВІСНОЇ АРХІТЕКТУРИ

2.1. Розроблення структури системи та взаємодії мікросервісів для генерації та обробки SMS-повідомлень

Проектування високонавантаженої системи обробки SMS-повідомлень вимагає ретельного аналізу функціональних вимог та вибору оптимальної архітектурної парадигми, здатної забезпечити необхідну масштабованість під час пікових навантажень. Система повинна ефективно справлятися з різкими змінами інтенсивності вхідного потоку повідомлень, що характерно для телекомунікаційних служб під час надзвичайних ситуацій, зокрема відключень електроенергії. У таких умовах критично важливою стає можливість динамічного розподілу обмежених обчислювальних ресурсів між компонентами системи для підтримки стабільності роботи. Мікросервісна архітектура природно підходить для вирішення цих завдань, оскільки дозволяє декомпозувати складну монолітну систему на незалежні слабкопов'язані компоненти з чітко визначеними зонами відповідальності. Кожен мікросервіс може масштабуватися незалежно від інших компонентів відповідно до актуального навантаження на конкретну функціональну підсистему. Запропонована архітектура системи базується на розділенні функціональності між двома основними мікросервісами: сервісом генерації та обробки запитів та сервісом обробки повідомлень [16]. Перший сервіс відповідає за взаємодію з користувачем через веб-інтерфейс, генерацію тестових SMS-повідомлень та ініціювання процесу їх передачі до обробного сервісу різними каналами зв'язку. Цей сервіс містить потужний генератор псевдовипадкових даних, що створює

реалістичні SMS з коректними номерами телефонів у форматі українських мобільних операторів та різноманітним текстовим контентом. Другий сервіс концентрується безпосередньо на прийомі, валідації, обробці та надійному збереженні SMS-повідомлень у базі даних з одночасним веденням детальної статистики продуктивності. Такий чіткий поділ відповідальності між сервісами забезпечує високу ступінь модульності системи та спрощує незалежне тестування й налагодження окремих компонентів.

Сервіс генерації реалізує потужний користувацький веб-інтерфейс на базі сучасного технологічного стеку Spring Boot з вбудованим шаблонізатором Thymeleaf для серверного рендерингу HTML-сторінок. Інтерфейс надає інтуїтивно зрозумілу форму для параметризації процесу генерації повідомлень, де користувач може задати бажану кількість SMS від одиниці до ста тисяч штук. Після натискання кнопки генерації система створює відповідну кількість унікальних записів повідомлень з автоматично згенерованими UUID-ідентифікаторами та зберігає їх у власній реляційній базі даних PostgreSQL. Згенеровані повідомлення відображаються у зручній таблиці з підтримкою посторінкового виведення для ефективного перегляду великих обсягів даних. Користувач отримує можливість вибрати один з трьох доступних каналів доставки для відправлення згенерованих повідомлень: синхронний HTTP, асинхронний HTTP або брокер повідомлень Kafka. Кожному каналу відповідає окрема кнопка запуску, що ініціює унікальний сценарій тестування продуктивності.

При натисканні будь-якої з кнопок запуску сервіс генерації створює унікальний ідентифікатор запуску у форматі UUID для точного відстеження метрик конкретного тестового прогону. Цей ідентифікатор використовується для агрегації статистичних даних та асоціації всіх повідомлень певного запуску з їх результатами обробки. Для синхронного HTTP-каналу сервіс послідовно відправляє пакети повідомлень на REST-кінцеву точку сервісу обробки через

бібліотеку OpenFeign, очікуючи підтвердження успішної обробки кожного пакету перед відправкою наступного [17]. Асинхронний HTTP-режим використовує пул потоків виконання для паралельної відправки множини HTTP-запитів, значно збільшуючи швидкість передачі за рахунок неблокуючих операцій вводу-виводу. Канал Kafka публікує повідомлення у попередньо налаштований топік брокера через API виробника Spring Kafka, делегуючи відповідальність за надійну доставку самому брокеру повідомлень. Незалежно від обраного каналу, веб-інтерфейс відображає статистику прогресу обробки повідомлень у реальному часі, опитуючи спеціальну кінцеву точку метрик сервісу обробки з інтервалом одна секунда.

Сервіс обробки побудовано як спеціалізований обробний вузол, що приймає SMS-повідомлення через різні транспортні протоколи та забезпечує їх надійне збереження з детальним відстеженням метрик. Для синхронної та асинхронної HTTP-комунікації сервіс експонує REST-кінцеву точку POST /messages/batch, яка приймає JSON-масив об'єктів повідомлень у тілі HTTP-запиту. Spring MVC контролер перевіряє валідність вхідних даних через анотації перевірки JavaBean, після чого делегує бізнес-логіку обробки відповідному сервісному класу. Для інтеграції з Kafka реалізовано спеціалізований споживач на базі анотації @KafkaListener фреймворку Spring Kafka, що автоматично споживає повідомлення з налаштованого топіку та обробляє їх аналогічно HTTP-запитам. Незалежно від джерела надходження, всі повідомлення проходять через єдиний конвеєр обробки, що гарантує консистентність обробки та збору статистики для коректного порівняння продуктивності різних каналів комунікації [18].

Центральна бізнес-логіка обробки повідомлень у сервісі обробки реалізована у вигляді транзакційного сервісу, що забезпечує атомарність операцій збереження даних та оновлення метрик. При отриманні пакету повідомлень система створює або оновлює запис статистики запуску у таблиці

результатів виконання, де відображаються загальна кількість повідомлень до обробки, тип використаного каналу доставки та час початку обробки. Кожне повідомлення конвертується у сутність доменної моделі оброблених SMS з додатковими полями ідентифікатора запуску для зв'язку зі статистикою, каналу для зазначення каналу надходження та часу обробки з точним часом завершення обробки. Репозиторій Spring Data JPA виконує пакетне збереження цих сутностей у базу даних з оптимізацією через пакетні операції для зменшення накладних витрат на взаємодію з СУБД. Після успішного завершення транзакції система атомарно інкрементує лічильник оброблених повідомлень та оновлює сукупний час обробки поточного запуску для подальших розрахунків середніх метрик продуктивності.

Для забезпечення можливості відстеження прогресу обробки в реальному часі сервіс обробки експонує GET-кінцеву точку /metrics/summary з обов'язковим параметром запиту ідентифікатора запуску для ідентифікації конкретного тестового прогону. Ця кінцева точка повертає JSON-об'єкт з агрегованою статистикою, що включає тип каналу доставки, загальну кількість повідомлень, кількість успішно оброблених та проблемних повідомлень, середній час обробки одного повідомлення у мілісекундах та загальний час виконання запуску. Поле завершеності вказує на завершеність обробки всіх повідомлень запуску, що дозволяє інтерфейсу користувача припинити опитування після досягнення фінального стану. Розрахунок середнього часу обробки виконується шляхом ділення сукупного часу на кількість оброблених повідомлень з округленням до цілого числа мілісекунд для зручності інтерпретації. Ця інформація безперервно оновлюється у веб-інтерфейсі сервісу генерації через JavaScript-опитування з візуалізацією у вигляді динамічного прогрес-бару та числових індикаторів стану обробки.

Після завершення обробки всіх повідомлень конкретного запуску система автоматично фіксує фінальні метрики у відповідному записі таблиці результатів

виконання. Зберігаються критичні параметри продуктивності: загальна кількість повідомлень у запуску, кількість успішно оброблених повідомлень, кількість невдалих спроб обробки, середній час обробки одного повідомлення, загальний час виконання запуску від старту до завершення останньої операції, точні мітки часу початку та завершення роботи [19]. Додаткове опціональне поле приміток може містити метаінформацію про умови тестування, зокрема використані ресурси процесора, виділену оперативну пам'ять, розмір пакетів повідомлень або інші релевантні параметри конфігурації. Для витягування збережених результатів попередніх запусків сервіс обробки надає GET-кінцеву точку /results з підтримкою посторінкового виведення для перегляду історичних даних та окрему GET-кінцеву точку /results/{runId} для отримання детальної інформації про конкретний запуск за його унікальним ідентифікатором.

Користувацький інтерфейс сервісу генерації включає спеціалізовану сторінку порівняння для комплексного порівняльного аналізу продуктивності різних каналів доставки повідомлень. При відкритті цієї сторінки інтерфейс виконує HTTP-запит до кінцевої точки /results сервісу обробки для отримання збережених результатів попередніх тестових запусків. Отримані дані структуруються у зрозумілу табличну форму з колонками для ідентифікатора запуску, типу використаного каналу, загальної кількості повідомлень, кількості успішно оброблених та проблемних повідомлень, середнього часу обробки та загального часу виконання. Таблиця дозволяє безпосередньо порівняти абсолютні та відносні показники ефективності синхронного HTTP, асинхронного HTTP та Kafka для ідентичних обсягів навантаження. Під таблицею система автоматично генерує текстовий висновок з виділенням найшвидшого каналу за критерієм мінімального загального часу обробки, найстабільнішого каналу з найменшою кількістю помилок та оптимального каналу за середнім часом обробки одного повідомлення.

Логіка автоматичного формування висновків базується на простих але ефективних евристиках порівняння збережених метрик продуктивності. Система перебирає всі записи результатів у таблиці, ідентифікуючи запис з мінімальним значенням загального часу як найшвидший канал за абсолютним часом виконання повного циклу обробки. Для визначення найстабільнішого каналу порівнюються значення поля невдалих обробок з вибором каналу з найменшою кількістю невдалих спроб обробки повідомлень. Оптимальний канал за ефективністю обробки окремого повідомлення визначається порівнянням значень середнього часу обробки. У випадку приблизної рівності ключових метрик між каналами система коректно відображає статус паритету продуктивності замість некоректного визначення переможця. Цей автоматизований аналітичний підхід надає користувачеві об'єктивну базу для прийняття обґрунтованих архітектурних рішень щодо оптимального методу комунікації в продакшн-середовищі з урахуванням специфічних вимог та обмежень системи.

Взаємодія між сервісами генерації та обробки через різні канали комунікації реалізована з використанням перевірених корпоративних технологій екосистеми Spring. Для HTTP-комунікації застосовується Spring Cloud OpenFeign, що надає декларативний підхід до опису REST-клієнтів через Java-інтерфейси з автоматичною генерацією проксі-класів під час виконання. Клієнт Feign автоматично обробляє серіалізацію та десеріалізацію JSON, підтримує налаштовані таймаути, політики повторних спроб та журналювання HTTP-взаємодій для спрощення налагодження. Інтеграція Kafka базується на Spring for Apache Kafka, що абстрагує низькорівневі деталі роботи з брокером через високорівневі анотації та конфігураційні класи. Виробник сервісу генерації використовує шаблон Kafka для публікації повідомлень з автоматичною серіалізацією доменних об'єктів у JSON через серіалізатор Jackson. Споживач сервісу обробки декларується через анотацію @KafkaListener з параметрами

топіку та групи споживачів, автоматично отримуючи десеріалізовані Java-об'єкти для обробки [20, с. 30-37]. Така уніфікована технологічна база забезпечує консистентність коду, спрощує адаптацію нових розробників та знижує технічний борг проекту.

Особливу увагу при проектуванні приділено питанням відмовостійкості та плавної деградації функціональності системи при частковій недоступності окремих компонентів. Синхронний HTTP-канал найбільш вразливий до каскадних відмов через жорстку синхронну залежність між сервісами, тому критично важливим є налаштування розумних таймаутів на рівні клієнта Feign для запобігання довгому блокуванню потоків виконання. Асинхронний HTTP-підхід з пулом потоків частково пом'якшує цю проблему через обмеження максимальної кількості одночасних запитів та швидке завершення з помилкою при досягненні ліміту. Kafka забезпечує найвищий рівень відмовостійкості завдяки гарантованому збереженню повідомлень на диску брокера та можливості повторного споживання з останнього збереженого зміщення у випадку краху споживача. Для продакшн-розгортання рекомендується налаштування Kafka з фактором реплікації не менше трьох для забезпечення високої доступності при відмовах окремих вузлів кластера брокерів. Патерн автоматичного вимикача може бути інтегрований через Spring Cloud Circuit Breaker для автоматичного переключення на резервну логіку при деградації сервісу обробки.

Масштабованість спроектованої системи забезпечується як вертикально через збільшення ресурсів окремих інстансів, так і горизонтально через запуск множини реплік сервісів. Сервіс генерації проектується переважно без збереження стану з збереженням даних виключно в базі, що дозволяє запускати довільну кількість його інстансів за балансувальником навантаження для розподілу навантаження від користувачів. Сервіс обробки також підтримує горизонтальне масштабування з важливим врахуванням особливостей кожного

каналу: HTTP-кінцеві точки природно масштабуються через розподіл вхідних запитів балансером, а споживачі Kafka формують групу споживачів для автоматичного розподілу навантаження між інстансами. Критично важливою є правильна конфігурація кількості розділів топіку Kafka, яка визначає максимальний рівень паралелізму споживання повідомлень: кількість активних споживачів не може перевищувати кількість розділів топіку. Додаткові оптимізації продуктивності досягаються через налаштування параметрів пулу з'єднань до бази даних, налаштування розміру пакета для JPA-операцій та використання кешування на рівні Spring Cache для частих операцій читання.

Моніторинг стану та продуктивності розподіленої системи забезпечується інтеграцією кінцевих точок Spring Boot Actuator, що експонують детальні метрики роботи віртуальної машини Java, статистику HTTP-запитів, стан пулу з'єднань та інші критичні параметри здоров'я додатку. Prometheus може збирати ці метрики через спеціалізовану кінцеву точку actuator для довгострокового зберігання часових рядів та побудови панелей управління у Grafana. Розподілене трасування запитів між сервісами реалізується через Spring Cloud Sleuth з автоматичним додаванням уніфікованих ідентифікаторів трасування та проміжків до всіх журналів та HTTP-заголовків. Ці траси агрегуються у спеціалізованій системі типу Zipkin або Jaeger для візуалізації повного шляху запиту через мікросервіси та ідентифікації вузьких місць продуктивності. Структуроване журналювання через Logback з форматом JSON полегшує централізовану агрегацію журналів у стек ELK або аналогічні системи для повнотекстового пошуку та аналізу патернів помилок. Правильно налаштований стек спостережуваності є необхідною умовою ефективної експлуатації продакшн-системи та швидкої реакції на інциденти.

Безпека розподіленої системи забезпечується шляхом впровадження сучасних стандартів аутентифікації та авторизації для захисту від несанкціонованого доступу. Веб-інтерфейс сервісу генерації може бути

захищений через Spring Security з підтримкою аутентифікації на основі форм або інтеграцією з корпоративними системами єдиного входу через протоколи SAML або OAuth 2.0. Міжсервісна комунікація через HTTP потребує додаткового захисного шару у вигляді взаємного TLS для верифікації ідентичності обох сторін з'єднання або JWT-токенів для передачі контексту безпеки між сервісами. Kafka може бути налаштована з механізмом аутентифікації SASL/SCRAM та правилами контролю доступу для контролю доступу до топіків, запобігаючи неавторизованому читанню чутливих даних або публікації шкідливих повідомлень. Шифрування даних при передачі забезпечується TLS для HTTP-з'єднань та Kafka SSL для брокера повідомлень, а шифрування під час зберігання реалізується засобами СУБД PostgreSQL через прозоре шифрування даних або дискове шифрування на рівні операційної системи [21]. Чутливі конфігураційні параметри типу облікових даних бази даних та ключів API зберігаються у захищених сховищах типу HashiCorp Vault або AWS Secrets Manager замість жорсткого кодування у файлах налаштувань додатку. На рисунку 2.1 представлено діаграму компонентів розробленої мікросервісної системи обробки SMS.

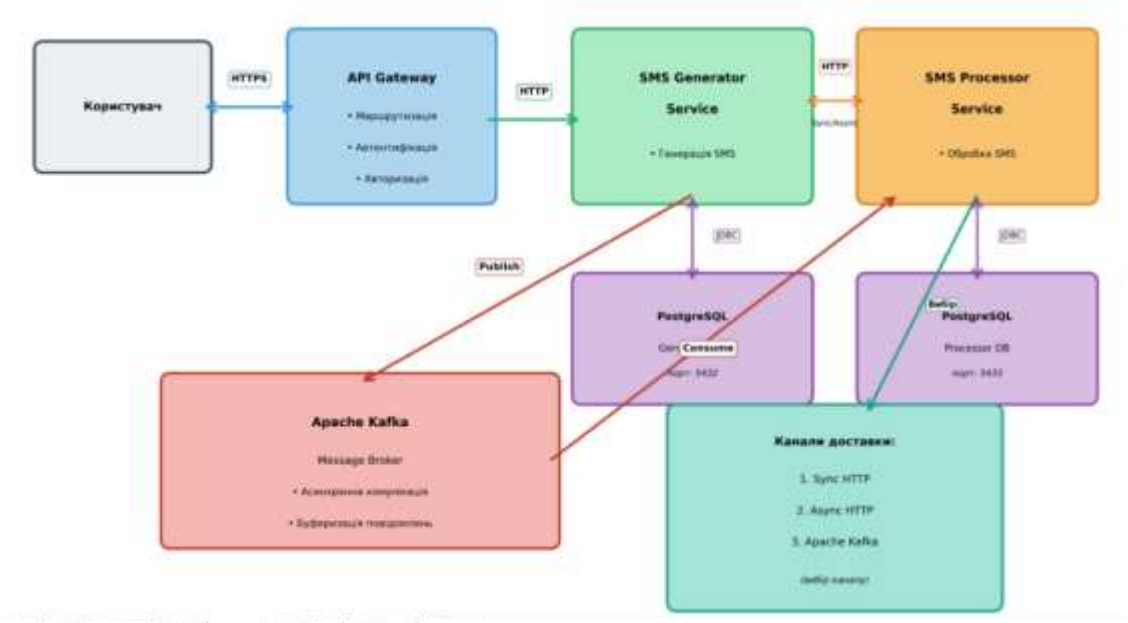

Рис. 2.1 – Діаграма компонентів мікросервісної системи обробки SMS

Розгортання системи автоматизується через контейнеризацію всіх компонентів за допомогою Docker з оркестрацією через Docker Compose для локальної розробки або Kubernetes для продакшн-середовища. Кожен мікросервіс пакується у власний Docker-образ на базі офіційного образу OpenJDK з багатоетапною збіркою для мінімізації розміру фінального контейнера (рис.2.2).

Рис. 2.2 – Логи запуску контейнера мікросервісу service-a, що відображають процес ініціалізації Spring Boot застосунку та встановлення з'єднання з базою даних PostgreSQL через HikariCP

Файл Docker Compose декларативно описує всю інфраструктуру системи включно з базами даних PostgreSQL для обох сервісів, брокером Kafka з необхідними залежностями у вигляді Zookeeper, та власними копіями сервісів

генерації і обробки з налаштованими мережевими зв'язками. Єдиною командою розробник може підняти повнофункціональне локальне середовище для тестування без складного ручного налаштування множини процесів. Продакшн-розгортання через Kubernetes надає потужні можливості автоматичного масштабування через горизонтальний автомасштабувач подів, самовідновлення через перевірки життєздатності та готовності, та оновлення без простою для безперервного оновлення версій сервісів без зупинки системи для кінцевих користувачів.

2.2. Проектування механізмів синхронної та асинхронної доставки повідомлень через HTTP та Kafka

Ефективна обробка високих навантажень у телекомунікаційних системах критично залежить від вибору оптимального механізму комунікації між розподіленими компонентами системи. Синхронна та асинхронна моделі взаємодії демонструють принципово різні характеристики продуктивності, надійності та складності реалізації, що робить детальне проектування цих механізмів ключовим етапом розробки масштабованої системи обробки SMS. Синхронний підхід базується на прямих запит-відповідь взаємодіях через HTTP-протокол з блокуванням виконання до отримання результату операції, що забезпечує простоту розуміння та відлагодження потоку даних але створює жорсткі часові залежності між компонентами. Асинхронна модель розв'язує ці залежності через введення проміжного буферного шару у вигляді черги повідомлень або архітектури на основі подій, дозволяючи відправнику продовжувати роботу незалежно від швидкості обробки повідомлень отримувачем [22, с. 109-117]. Правильне поєднання обох підходів дозволяє оптимізувати систему для різних типів операцій залежно від їх критичності, частоти виконання та вимог до затримки.

Синхронна HTTP-комунікація у спроектованій системі реалізується через класичний RESTful API з використанням фреймворку Spring Web MVC для серверної частини та Spring Cloud OpenFeign для клієнтської взаємодії. Сервіс генерації використовує декларативно описаний інтерфейс Feign з анотацією @FeignClient для автоматичної генерації HTTP-клієнта до сервісу обробки під час старту додатку. Метод інтерфейсу маркується анотацією @PostMapping з вказівкою URL кінцевої точки та очікуваного формату даних у вигляді JSON через заголовок Content-Type application/json. При виклику цього методу Feign автоматично серіалізує список SMS-об'єктів у JSON, виконує HTTP POST запит до сервісу обробки та блокує поточний потік виконання до отримання HTTP-відповіді від сервера. Сервіс обробки обробляє вхідний запит через Spring MVC контролер з анотаціями @RestController та @RequestBody для автоматичної десеріалізації JSON у Java-об'єкти, виконує бізнес-логіку збереження у базу даних та повертає HTTP-статус 200 з порожнім тілом відповіді у випадку успіху або відповідний код помилки при виникненні винятку.

Критичним обмеженням синхронного підходу є лінійна залежність загального часу виконання від кількості послідовних HTTP-запитів та затримки кожного окремого запиту. При відправці великої кількості повідомлень сервіс генерації змушений послідовно чекати підтвердження успішної обробки кожного пакету перед відправкою наступного, що призводить до накопичення затримок та низького використання мережевого каналу і ресурсів сервісу обробки. Для пом'якшення цієї проблеми реалізовано пакетування повідомлень у пакети фіксованого розміру, зазвичай від 100 до 1000 SMS на запит залежно від конфігурації, що значно зменшує накладні витрати HTTP-протоколу та кількість циклів передачі. Розмір пакету обирається емпірично через балансування між зменшенням накладних витрат та збільшенням ризику втрати великої кількості даних при невдалому запиті. Додаткова оптимізація досягається через постійні з'єднання на рівні HTTP/1.1 для повторного

використання TCP-з'єднання між множинними запитами до того ж хоста замість накладного встановлення нового з'єднання для кожного запиту [23]. Мультиплексування HTTP/2 потенційно дозволяє паралельну відправку множини запитів через єдине TCP-з'єднання, але вимагає відповідної підтримки на рівні інфраструктури.

Асинхронна HTTP-комунікація кардинально покращує використання ресурсів через розпаралелювання множини незалежних HTTP-запитів замість їх послідовного виконання. Сервіс генерації створює пул потоків через Java ExecutorService з налаштованою кількістю робочих потоків, зазвичай кратною кількості доступних процесорних ядер для оптимального балансу між паралелізмом та накладними витратами управління потоками. Кожен пакет повідомлень обгортається у завдання, що виконує HTTP-запит через клієнт Feign та повертає результат операції у вигляді об'єкта Future для асинхронного відстеження завершення. Сервіс виконання розподіляє ці завдання між доступними робочими потоками, дозволяючи одночасну відправку множини HTTP-запитів обмежену лише розміром пулу потоків замість блокування на кожному запиті. Після надсилання всіх завдань головний потік збирає результати через виклик методу get() на кожному Future, що блокується до завершення відповідного запиту але дозволяє паралельне виконання усіх запитів у фоні. API CompletableFuture надає більш потужні можливості композиції асинхронних операцій та обробки помилок порівняно з базовим інтерфейсом Future.

Ключовою перевагою асинхронного HTTP-підходу є драматичне зменшення загального часу виконання відправки повідомлень через паралельну обробку замість послідовної. При налаштованому пулі потоків розміром N система може одночасно виконувати N HTTP-запитів, теоретично зменшуючи загальний час у N разів порівняно з синхронним підходом за умови достатньої пропускної здатності мережі та обчислювальних ресурсів сервісу обробки. На

практиці прискорення обмежується законом Амдала через наявність послідовних вузьких місць у вигляді створення завдань, збору результатів та конкуренції за спільні ресурси типу мережевих буферів сокетів. Оптимальний розмір пулу потоків визначається емпірично через навантажувальне тестування і залежить від співвідношення обчислювальних та операцій вводу-виводу у запиті: для переважно операцій вводу-виводу ефективний розмір може значно перевищувати кількість процесорних ядер завдяки блокуванню потоків на мережевих операціях. Занадто великий пул призводить до неефективних накладних витрат управління потоками та потенційного виснаження пулу потоків сервісу обробки при надто інтенсивному навантаженні [24].

Обидва HTTP-підходи вразливі до проблеми зворотного тиску коли сервіс обробки не встигає обробляти вхідні запити з швидкістю їх надходження від сервісу генерації, призводячи до накопичення черги необроблених запитів та потенційного виснаження пулу з'єднань або пулу потоків на сервері. Синхронний варіант природно обмежує навантаження через блокування відправника до завершення обробки попереднього запиту, хоча це негативно впливає на пропускну здатність. Асинхронний підхід потребує явного механізму обмеження швидкості або адаптивного дроселювання для запобігання перевантаження сервера: простий семафор може обмежити максимальну кількість одночасних активних запитів, або складніший алгоритм токенів у відрі дозволяє плавно контролювати середню швидкість відправки з допустимими піками навантаження. Spring WebFlux з Project Reactor надає реактивний неблокуючий підхід до HTTP-комунікації через оператори з підтримкою зворотного тиску, але вимагає повної асинхронності всього стеку включно з драйверами баз даних. Правильно налаштовані таймаути з'єднання та читання є критичними для запобігання безстроковому блокуванню при проблемах мережі або повільній обробці на сервері, дозволяючи клієнту швидко завершитися з помилкою та звільнити ресурси для обробки інших запитів. Процес синхронної

обробки SMS-повідомлення представлено на рисунку 2.3. При синхронному підході клієнт очікує на завершення всього ланцюжка операцій.

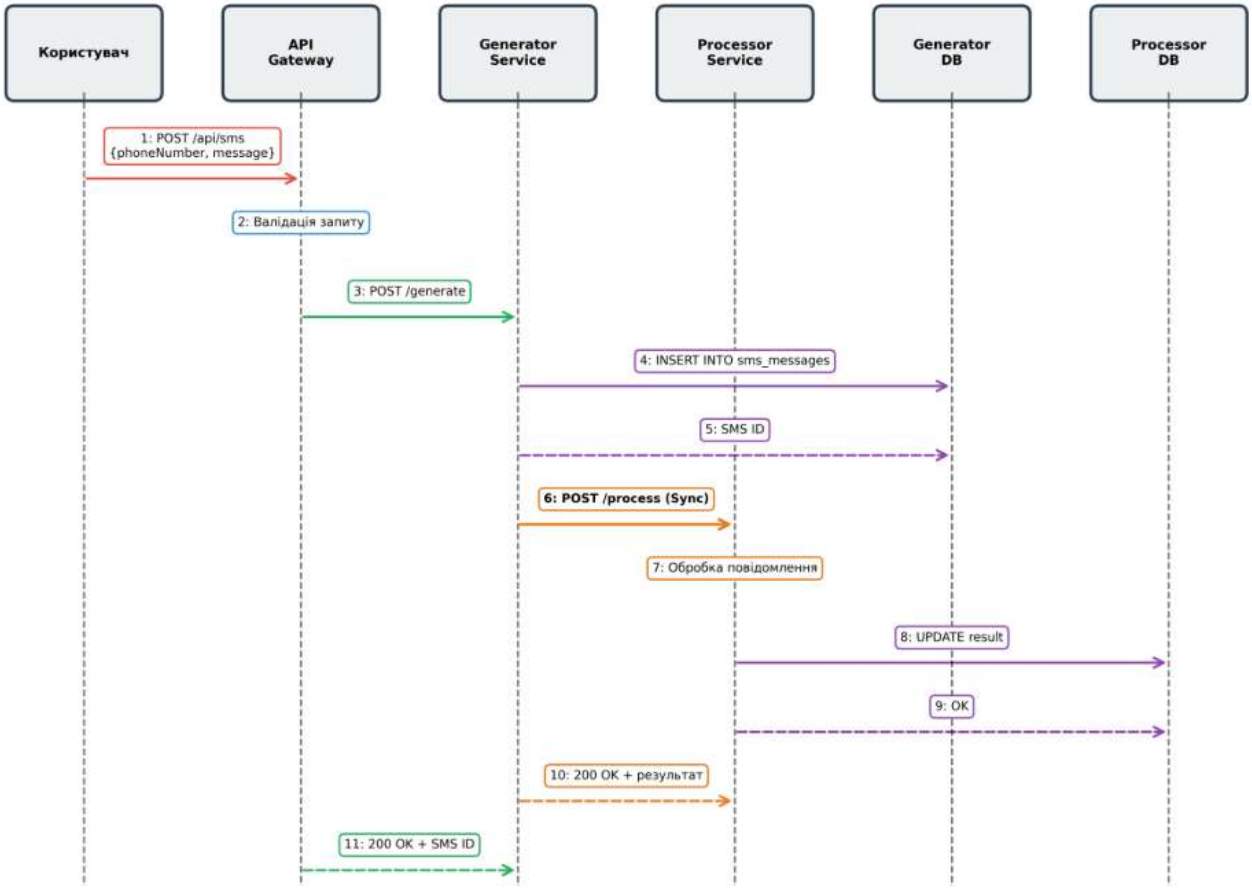

Рис. 2.3 – Діаграма послідовності синхронної обробки SMS-повідомлення через HTTP

Асинхронний підхід до обробки SMS-повідомлень через Apache Kafka зображено на рисунку 2.4.

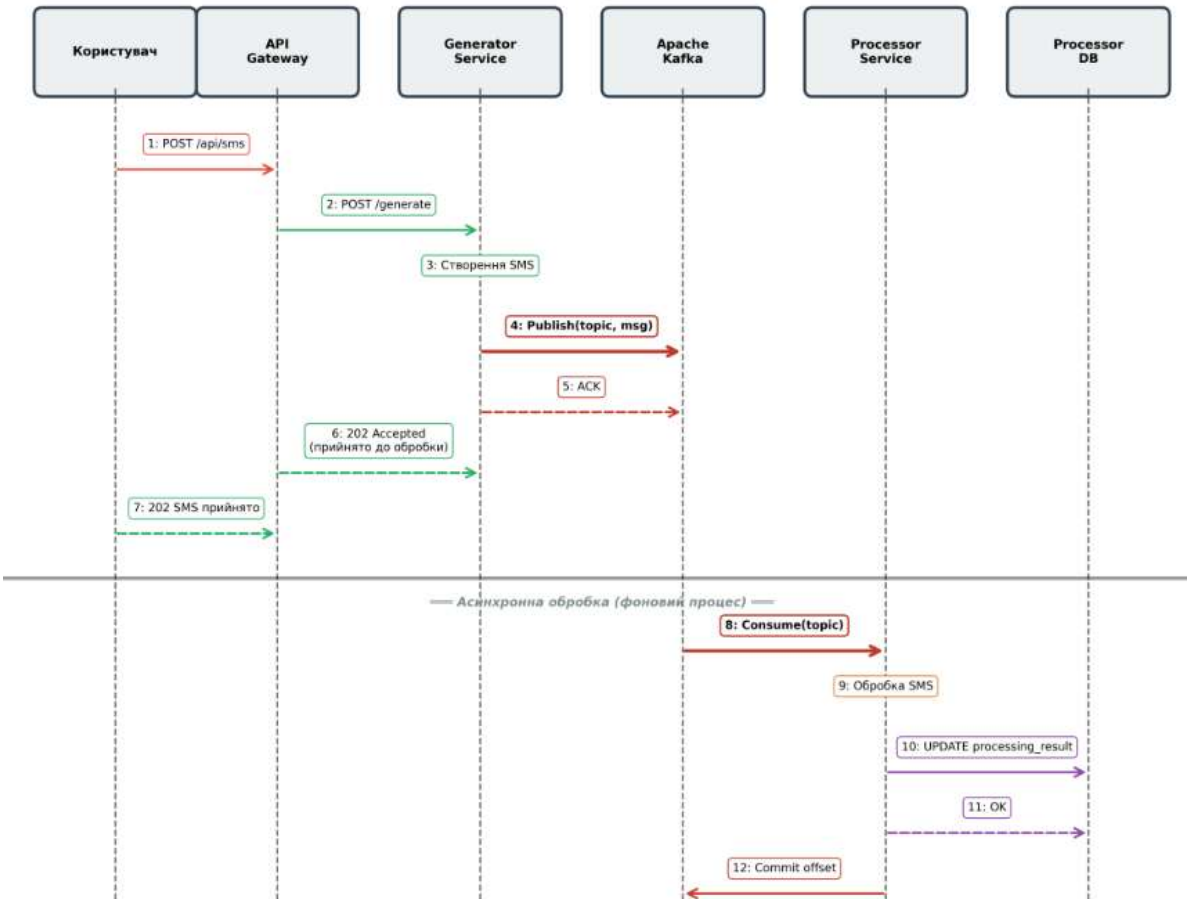

Рис. 2.4 – Діаграма послідовності асинхронної обробки SMS-повідомлення через Apache Kafka

Асинхронна комунікація на основі Kafka фундаментально відрізняється від HTTP-підходів через введення постійного розподіленого брокера повідомлень як роз'єднуючого шару між виробником та споживачем. Сервіс генерації виступає виробником Kafka, що публікує SMS-повідомлення у попередньо створений топік з налаштованою кількістю розділів для паралельної обробки. Spring Kafka надає утилітний клас шаблону Kafka для зручної

високорівневої взаємодії з брокером, автоматично обробляючи серіалізацію Java-об'єктів у байтові масиви через налаштований серіалізатор JSON Jackson. API виробника є асинхронним за замовчуванням: метод send() повертає об'єкт Future, що вирішується після успішного запису повідомлення у реплікацію розділу-лідера на брокері, дозволяючи виробнику продовжувати публікацію наступних повідомлень без блокування. Ключові конфігураційні параметри виробника включають налаштування підтверджень для контролю гарантій стійкості (підтвердження всіх вимагає підтвердження від усіх синхронізованих реплік для максимальної надійності), розмір пакета для групування множини записів у єдиний мережевий запит для зменшення накладних витрат, та тип стиснення для зменшення пропускної здатності мережі через gzip або snappy-стиснення корисного навантаження повідомлень.

Сервіс обробки реалізує споживача Kafka через анотацію Spring Kafka @KafkaListener на методі, що автоматично обробляє весь шаблонний код створення споживача, підписки на топік, циклу опитування та десеріалізації повідомлень. Споживач налаштовується з унікальним ідентифікатором групи для формування групи споживачів, де Kafka автоматично розподіляє розділи топіку між активними членами групи для паралельної обробки: кожен розділ може споживатися лише одним споживачем у групі у кожен момент часу, що природно обмежує максимальний паралелізм кількістю розділів топіку. При додаванні нового споживача у групу Kafka виконує ребалансування для перерозподілу розділів між всіма членами, тимчасово призупиняючи споживання під час цього процесу [25]. Управління зміщеннями автоматично обробляється Spring Kafka з періодичним збереженням прочитаних зміщень назад у спеціальний внутрішній топік зміщень споживачів для забезпечення семантики доставки принаймні один раз або рівно один раз залежно від налаштувань автоматичного збереження та ручної логіки збереження. Ручне

управління зміщеннями надає найточніший контроль над гарантіями за рахунок збільшення складності коду споживача.

Фундаментальною перевагою архітектури Kafka є природна підтримка зворотного тиску через постійне зберігання повідомлень на диску брокера та можливість споживача опитувати повідомлення зі швидкістю їх обробки без ризику перевантаження. Якщо споживач тимчасово не може обробляти повідомлення через високе навантаження або технічні проблеми, повідомлення просто накопичуються у відповідних розділах топіку до моменту відновлення нормальної роботи споживача, після чого обробка продовжується з останнього збереженого зміщення. Це контрастує з HTTP-підходами де сповільнення сервера негайно поширюється назад до клієнта через помилки таймауту або заповнені пули з'єднань. Політика утримання топіку визначає максимальний час або розмір накопичених повідомлень перед їх видаленням, зазвичай налаштовується на кілька днів для забезпечення достатнього вікна відновлення при тривалих відмовах. Реплікація Kafka з налаштовуваним фактором реплікації забезпечує відмовостійкість до відмов окремих вузлів брокера: при налаштуванні фактора реплікації 3 кожен розділ реплікується на три різні брокери, гарантуючи нульову втрату даних при відмові до двох вузлів одночасно. Вибори лідера автоматично підвищують одну з реплік-послідовників у лідера при недоступності поточного лідера для безперервної обробки [26].

Продуктивність системи Kafka суттєво залежить від правильної стратегії розподілу при публікації повідомлень у топік. За замовчуванням Kafka використовує розподіл на основі хешу ключа повідомлення для детермінованого розподілу повідомлень з однаковим ключем у той самий розділ, що важливо для збереження гарантій упорядкування у межах логічного ключа. Для SMS-системи розумним ключем може бути номер телефону відправника для забезпечення обробки всіх повідомлень від конкретного відправника у порядку їх надходження, або нульовий ключ для циклічного розподілу між розділами для

максимального паралелізму при відсутності вимог до впорядкування. Кількість розділів топіку є критичним параметром налаштування що прямо визначає максимальний рівень паралелізму споживання: більше розділів дозволяє більше паралельних споживачів але збільшує накладні витрати на брокері через більшу кількість відкритих дескрипторів файлів та управління метаданими. Емпірично рекомендується починати з кількості розділів кратної очікуваній кількості інстансів споживачів з невеликим запасом для майбутнього масштабування, зазвичай від 3 до 12 розділів для малих та середніх навантажень.

Характеристики затримки трьох спроектованих підходів принципово відрізняються і мають бути враховані при виборі оптимального каналу для конкретних випадків використання. Синхронний HTTP демонструє найнижчу наскрізну затримку для обробки окремого повідомлення завдяки прямому патерну запит-відповідь без проміжних переходів: типові значення у межах 1-10 мілісекунд для локальної мережі за умови низького навантаження на сервері. Асинхронний HTTP має порівнянну затримку окремого запиту але значно кращу сукупну пропускну здатність через паралельну обробку. Kafka вносить додаткову затримку через необхідність запису повідомлення на диск брокера та наступного опитування споживача, зазвичай у діапазоні 5-50 мілісекунд залежно від налаштувань затримки виробника для пакетування та мінімальних байтів отримання споживача для мінімізації кількості запитів опитування. Для чутливих до затримки операцій типу сповіщень у реальному часі синхронний HTTP може бути кращим вибором, тоді як для високопродуктивної пакетної обробки Kafka забезпечує оптимальну продуктивність. Компроміс між затримкою та пропускною здатністю є фундаментальним при проектуванні розподілених систем і потребує ретельного розгляду на основі специфічних вимог.

Надійність доставки та гарантії обробки суттєво відрізняються між HTTP та Kafka підходами. HTTP-запит може провалитися з множини причин: таймаут

мережі, відмова з'єднання, внутрішня помилка сервера, або проблеми десеріалізації, вимагаючи клієнта самостійно реалізовувати логіку повторів з експоненційною затримкою для тимчасових помилок. При успішній HTTP-відповіді 200 клієнт може бути впевнений що сервер прийняв запит, але не має гарантій що операція була атомарно збережена у базу даних без додаткової логіки ідемпотентності. Kafka надає сильніші гарантії через налаштовуваний параметр підтверджень: підтвердження одного чекає лише підтвердження реплікації-лідера (швидко але ризик втрати даних при відмові лідера), підтвердження всіх чекає підтвердження всіх синхронізованих реплік (найповільніше але максимальна стійкість). Споживач може реалізувати обробку принаймні один раз через збереження зміщень лише після успішної обробки повідомлень, або рівно один раз через API транзакцій для атомарного збереження зміщень та змін бази даних у єдиній розподіленій транзакції. Продакшн-системи зазвичай обирають принаймні один раз з ідемпотентною логікою обробки для балансу надійності та продуктивності [27]. Моніторинг та спостережуваність кожного каналу комунікації вимагають специфічних метрик для ефективного виявлення вузьких місць продуктивності. HTTP-кінцеві точки експонують стандартні метрики через Spring Boot Actuator: швидкість запитів, середній час відповіді, швидкість помилок на кінцеву точку, кількість активних з'єднань та використання пулу потоків. Додаткові користувацькі метрики можуть відстежувати розмір корисного навантаження запиту та відповіді та розподіл часів відповіді через гістограму. Метрики виробника Kafka включають швидкість відправки записів для пропускної здатності, швидкість помилок записів для надійності, середній розмір пакета для ефективності пакетування та середню швидкість стиснення для оптимізації мережі. Метрики споживача відстежують відставання записів для кількості необроблених повідомлень у розділах (критичний індикатор зворотного тиску), швидкість отримання та швидкість спожитих байтів для пропускної здатності, та затримку збереження

для відстеження продуктивності управління зміщеннями. Розподілене трасування через Sleuth автоматично поширює контекст трасування між сервісами для HTTP-запитів та може бути вручну інтегровано для повідомлень Kafka через користувацькі заголовки, дозволяючи наскрізну видимість потоку запитів через множину сервісів та рівнів інфраструктури.

Тестування різних каналів комунікації вимагає комплексного підходу з модульними, інтеграційними та тестами продуктивності на різних рівнях абстракції. Модульні тести перевіряють коректність серіалізації та десеріалізації доменних об'єктів, правильність налаштування клієнтів та обробку граничних випадків типу нульових значень або надмірно великих корисних навантажень. Інтеграційні тести використовують вбудований брокер Kafka через анотацію @EmbeddedKafka або WireMock для імітації HTTP-кінцевих точок, перевіряючи реальну інтеграцію компонентів без залежності від зовнішньої інфраструктури. Тести продуктивності через JMeter або Gatling симулюють реалістичні патерни навантаження з варіюючою кількістю одночасних користувачів та швидкістю повідомлень, вимірюючи пропускну здатність, процентилі затримки та швидкість помилок під різними рівнями стресу. Практики хаос-інженерії через контрольоване введення відмов типу затримок мережі, крахів брокерів або недоступності бази даних валідують механізми стійкості типу автоматичних вимикачів, повторів та резервної логіки. Комплексне покриття тестами на всіх рівнях є критичним для впевненості у продакшн-розгортанні високонавантаженої системи. Оптимізація продуктивності кожного каналу досягається через ретельне налаштування множини конфігураційних параметрів на основі емпіричних вимірювань. Синхронний HTTP отримує користь від збільшення розміру пулу з'єднань на клієнті і розміру пулу потоків на сервері для обробки більшої кількості одночасних запитів, налаштування відповідних таймаутів читання та запису для запобігання зависанню з'єднань, та увімкнення постійних з'єднань HTTP для повторного використання з'єднань. Асинхронний

HTTP потребує балансування розміру пулу потоків проти накладних витрат перемикання контексту та споживання пам'яті на потік, налаштування ємності черги для очікуючих завдань та політики відхилення при виснаженні пулу. Виробник Kafka оптимізується через збільшення розміру пакета та затримки для зменшення накладних витрат але за рахунок збільшеної затримки, стиснення для зменшення пропускної здатності мережі але додаткових накладних витрат процесора на серіалізацію. Налаштування споживача включає максимальну кількість записів опитування для контролю розміру пакета обробленого за одне опитування, мінімальні байти отримання для зменшення кількості запитів опитування але потенційного збільшення затримки, та таймаут сесії для балансу між швидкістю виявлення відмов споживачів та толерантністю до тимчасових сповільнень. Систематична оптимізація через ітеративне вимірювання, налаштування та повторне вимірювання є необхідною для досягнення пікової продуктивності продакшн-системи.

Остаточний вибір між синхронним HTTP, асинхронним HTTP та Kafka для конкретного випадку використання базується на комплексному аналізі функціональних та нефункціональних вимог системи. Синхронний підхід оптимальний для сценаріїв запит-відповідь з низькою затримкою та помірним навантаженням де простота реалізації та налагодження переважає над максимальною пропускною здатністю, наприклад кінцеві точки API для користувачів для запитів у реальному часі. Асинхронний HTTP забезпечує суттєво кращу пропускну здатність для пакетних операцій при збереженні відносно простої архітектури без додаткової інфраструктури типу брокера повідомлень, підходить для періодичної синхронізації даних або масового імпорту. Kafka стає необхідним для справді високопродуктивних сценаріїв з непередбачуваними піками навантаження, вимог до сильних гарантій стійкості та потреби у роз'єднанні життєвих циклів виробника та споживача, типово архітектури на основі подій або конвеєри обробки потоків. Гібридний підхід

комбінує переваги множини патернів: синхронний API для запитів користувачів з негайною відповіддю, асинхронна обробка через Kafka для важких фонових завдань, забезпечуючи оптимальний баланс між користувацьким досвідом та масштабованістю системи.

2.3. Розроблення моделі даних та схеми баз даних для зберігання SMS та результатів обробки

Проектування ефективної моделі даних є критичним аспектом створення високопродуктивної системи обробки SMS, оскільки структура та організація даних безпосередньо впливають на швидкість виконання запитів, простоту масштабування та надійність зберігання інформації. У контексті розподіленої мікросервісної архітектури кожен сервіс має власну автономну базу даних згідно з принципом бази даних на сервіс, що забезпечує слабке зв'язування між сервісами та їх незалежну еволюцію. Сервіс генерації відповідає за зберігання первинних згенерованих SMS-повідомлень до їх відправлення на обробку, тоді як сервіс обробки зберігає оброблені повідомлення разом з детальними метриками продуктивності кожного тестового запуску (рис.2.5-2.7).

Рис. 2.5 – Інтерфейс веб-додатку модуля генерації SMS із введенням кількості повідомлень та відображенням процесу їх створення.

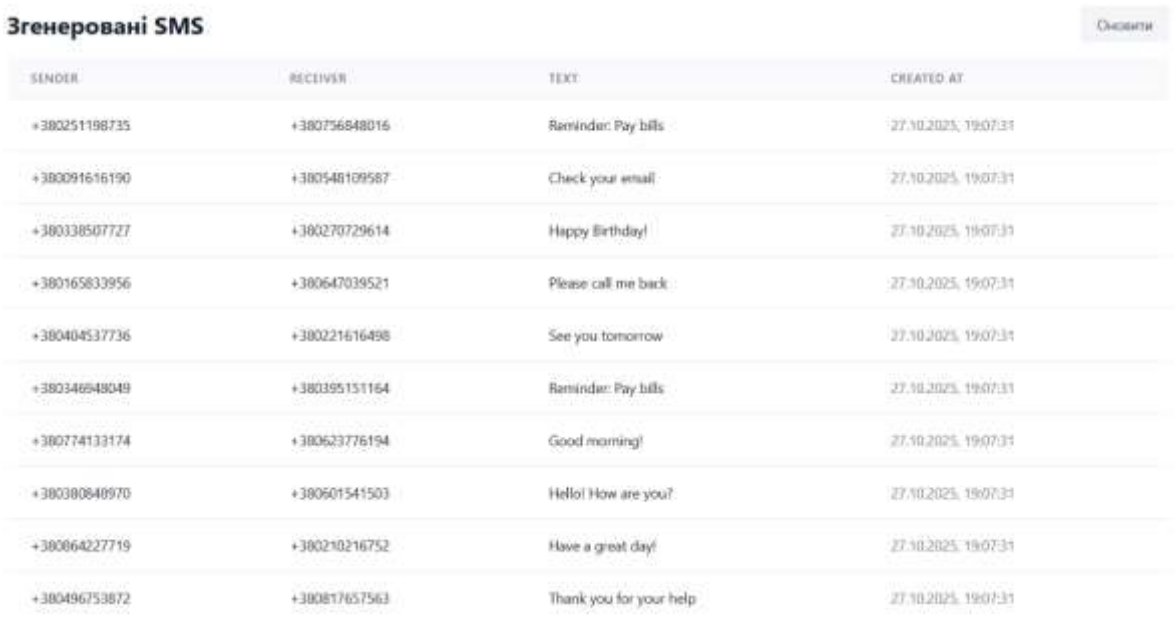

Рис. 2.6 – Відображення згенерованих SMS-повідомлень у таблиці користувацького інтерфейсу з даними про відправника, одержувача, текст і час створення.

Рис. 2.7 – Результати тестування відправлення 1000 SMS через канал Apache Kafka із відображенням повної обробки повідомлень без помилок та середнього часу виконання 0.06 мс

Вибір PostgreSQL як системи управління базами даних обумовлений її надійністю, відповідністю властивостям ACID, потужною підтримкою транзакцій та індексування, що критично важливо для забезпечення консистентності даних під високим навантаженням. Модель даних проектується з урахуванням майбутньої можливості горизонтального масштабування через розподіл таблиць або міграцію на розподілені SQL-рішення при необхідності обробки понад мільйон повідомлень на секунду.

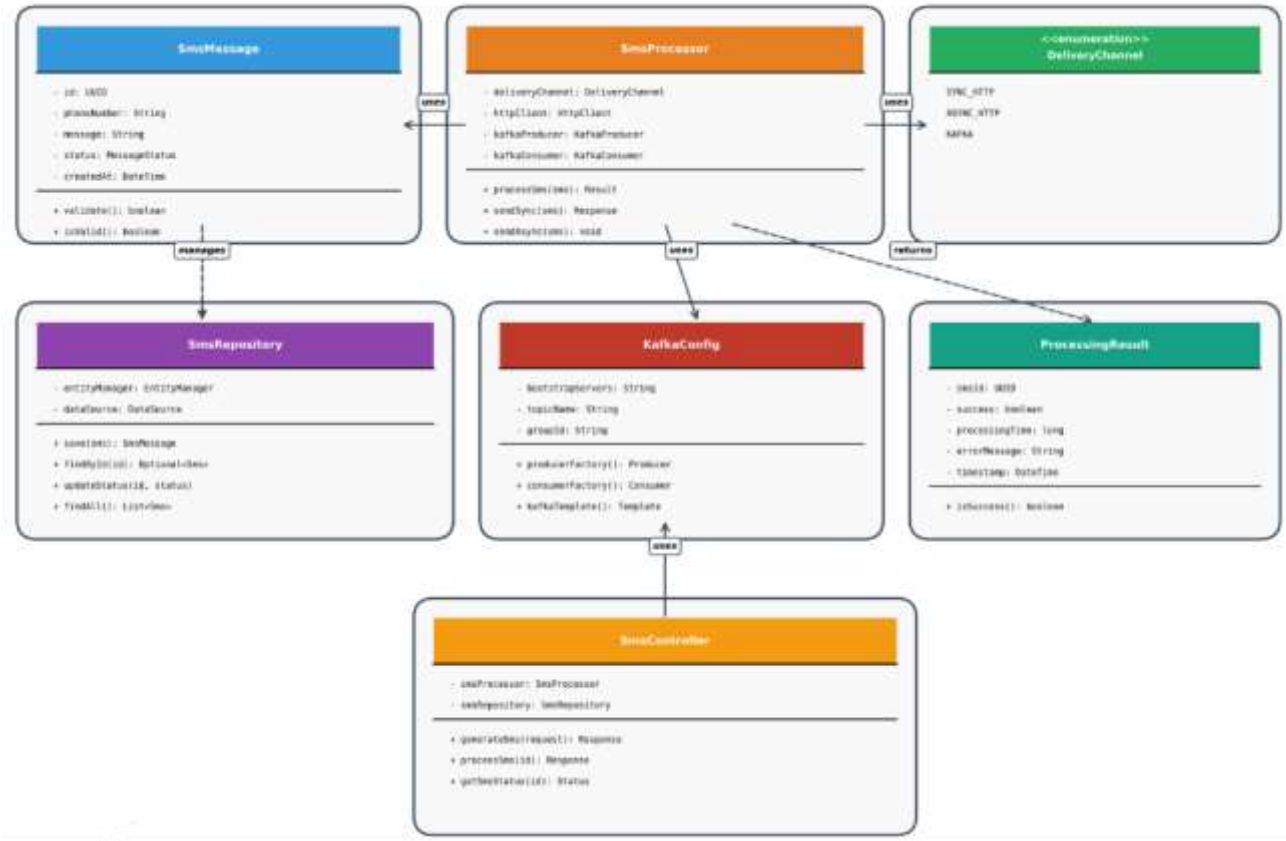

Рис. 2.8 – Діаграма класів основних сутностей системи обробки SMS

База даних сервісу генерації містить єдину центральну таблицю sms для зберігання всіх згенерованих повідомлень перед їх відправленням до сервісу обробки. Ця таблиця має мінімалістичну але достатню структуру з п'ятьма колонками: id як первинний ключ типу UUID для глобально унікальної ідентифікації повідомлень без ризику колізій при розподіленій генерації, sender типу VARCHAR(15) для зберігання номера відправника у міжнародному форматі з кодом країни, receiver типу VARCHAR(15) для номера отримувача, text типу TEXT для необмеженої довжини текстового вмісту SMS, та created_at типу TIMESTAMP для точної фіксації моменту створення запису у базі. Використання UUID замість послідовних цілочисельних колонок забезпечує можливість розподіленої генерації ідентифікаторів без необхідності централізованої координації або ризику дублювання значень при паралельній роботі множини інстансів сервісу. VARCHAR(15) достатньо для зберігання будь-якого міжнародного телефонного номера у форматі E.164 з максимальною довжиною 15 цифр включно з кодом країни, наприклад +380671234567 для українських мобільних номерів [28].

Для оптимізації продуктивності запитів на вибірку даних з таблиці sms створюється B-дерево індекс на колонці created_at, що дозволяє ефективно виконувати часові запити для отримання останніх згенерованих повідомлень з логарифмічною складністю замість повного сканування таблиці. Цей індекс критично важливий для реалізації посторінкового виведення у веб-інтерфейсі, де користувач може переглядати сторінки згенерованих SMS відсортованих за часом створення у спадному порядку. Додатковий складений індекс на (created_at, id) може покращити продуктивність посторінкового виведення де наступна сторінка запитується через умову WHERE created_at < ? OR (created_at = ? AND id < ?) замість простого зміщення який стає неефективним на великих значеннях зміщення. Для запобігання необмеженому зростанню таблиці sms можна реалізувати періодичну задачу очищення що видаляє старі записи після

їх успішної обробки сервісом обробки та збереження фінальних результатів, зазвичай через налаштовуваний період утримання типу 30 днів. Розподіл таблиці за датою created_at (наприклад щоденні або щомісячні розділи) полегшує ефективне видалення застарілих даних через просте видалення розділу замість повільних операцій DELETE та покращує продуктивність запитів що торкаються лише свіжих даних.

Доменна модель сервісу генерації представлена Java-класом Sms з анотаціями JPA для відображення на таблицю бази даних через фреймворк ORM Spring Data JPA. Клас декларує приватні поля що відповідають колонкам таблиці: id типу UUID з анотаціями @Id та @GeneratedValue для автоматичної генерації унікальних ідентифікаторів, sender та receiver типу String з @Column(nullable = false, length = 15) для застосування обмеження NOT NULL та обмеження довжини, text типу String з @Column(columnDefinition = "TEXT") для необмеженої довжини, та createdAt типу LocalDateTime з @Column(nullable = false, updatable = false) для незмінної мітки часу. Методи отримання та встановлення значень або анотації Lombok @Data надають доступ до полів згідно з конвенцією JavaBeans. Інтерфейс репозиторію Spring Data JPA SmsRepository розширює JpaRepository<Sms, UUID> автоматично надаючи методи створення читання оновлення видалення плюс користувацькі методи запитів через конвенції назв методів, наприклад findByCreatedAtBeforeOrderByCreatedAtDesc для посторінкових запитів старих записів. Компоненти сервісного шару транзакцій використовують цей репозиторій для операцій бізнес-логіки з автоматичним управлінням транзакціями через анотації @Transactional [29].

База даних сервісу обробки має більш складну структуру з двома основними таблицями для розділення операційних даних про оброблені повідомлення від аналітичних даних про результати тестових запусків. Таблиця processed_sms зберігає копію кожного успішно обробленого SMS-повідомлення

з додатковими метаданими про контекст обробки. Структура включає id (UUID PRIMARY KEY), sender (VARCHAR(15) NOT NULL), receiver (VARCHAR(15) NOT NULL), text (TEXT NOT NULL) - дублюючи базові поля з таблиці sms сервісу генерації для повної автономності сервісу обробки без потреби міжсервісних запитів до бази даних. Додаткові колонки created_at (TIMESTAMP NOT NULL) зберігає оригінальний час створення повідомлення переданий від сервісу генерації, processed_at (TIMESTAMP NOT NULL DEFAULT CURRENT_TIMESTAMP) фіксує точний момент завершення обробки у сервісі обробки для розрахунку затримки обробки, run_id (UUID NOT NULL) є зовнішнім ключем що зв'язує кожне повідомлення з конкретним тестовим запуском для агрегації метрик, та channel (VARCHAR(20) NOT NULL) зберігає значення переліку типу SYNC_HTTP, ASYNC_HTTP або KAFKA для ідентифікації каналу надходження повідомлення.

Таблиця run_results агрегує фінальну статистику кожного завершеного тестового запуску для подальшого порівняльного аналізу продуктивності різних каналів. Структура таблиці: run_id (UUID PRIMARY KEY) унікально ідентифікує кожен запуск, channel (VARCHAR(20) NOT NULL) зберігає тип використаного каналу доставки, total (INTEGER NOT NULL) містить загальну заплановану кількість повідомлень у запуску, processed (INTEGER NOT NULL) відображає фактичну кількість успішно оброблених повідомлень, failed (INTEGER NOT NULL DEFAULT 0) рахує кількість невдалих спроб обробки, average_processing_ms (INTEGER NOT NULL) зберігає середній час обробки одного повідомлення у мілісекундах розрахований як загальний час поділений на оброблені, total_time_ms (BIGINT NOT NULL) містить загальний час виконання запуску від першого до останнього обробленого повідомлення, started_at (TIMESTAMP NOT NULL) та finished_at (TIMESTAMP) фіксують часові межі запуску, notes (TEXT) опціонально зберігає додаткові метадані про умови тестування типу специфікацій обладнання або параметрів конфігурації.

Використання BIGINT для total_time_ms необхідно для уникнення переповнення при тривалих запусках що можуть тривати години при обробці мільйонів повідомлень де максимум INTEGER може бути недостатнім [30].

Критично важливими для продуктивності є правильно спроектовані індекси на таблиці processed_sms що забезпечують швидкі запити агрегації для розрахунку метрик. Складений B-дерево індекс на (run_id, processed_at) оптимізує запити що рахують кількість оброблених повідомлень у конкретному запуску та обчислюють мінімальні та максимальні мітки часу для розрахунку загального часу. Окремий індекс на run_id покращує продуктивність обмежень зовнішнього ключа та операцій об'єднання між таблицями processed_sms та run_results. Індекс на колонці channel дозволяє ефективно фільтрувати повідомлення за типом каналу для аналітики специфічної для каналу. Для таблиці run_results індекс первинного ключа на run_id автоматично забезпечує унікальність та швидкий пошук за ідентифікатором запуску. Складений індекс на (channel, finished_at) оптимізує запити що витягують останні завершені запуски для конкретного каналу відсортовані за часом для вікна порівняння. Періодичне обслуговування PostgreSQL VACUUM ANALYZE забезпечує актуальність статистики таблиць для планувальника запитів що критично для вибору оптимальних планів виконання під час високого навантаження.

Доменна модель сервісу обробки містить дві сутності JPA що відображаються на відповідні таблиці. Сутність ProcessedSms дублює поля Sms плюс додаткові поля для метаданих обробки: processedAt типу LocalDateTime з @Column(nullable = false) та автоматичним заповненням через зворотній виклик або стандартне значення бази даних, runId типу UUID з @Column(nullable = false) та зв'язком @ManyToOne до сутності RunResult для навігації, channel типу переліку DeliveryChannel з @Enumerated(EnumType.STRING) для зрозумілих людині значень у базі даних замість крихких порядкових цілих чисел. Сутність RunResult містить поля відповідні колонкам run_results з відповідними

анотаціями JPA: runId як @Id без @GeneratedValue оскільки призначається вручну сервісною логікою, channel як перелік з анотацією, числові поля з обмеженнями @Column(nullable = false), поля міток часу з відповідними тимчасовими типами, та двонаправлений зв'язок @OneToMany до сутностей ProcessedSms для навігації від агрегату до окремих повідомлень. Анотації Lombok @Data або ручні методи отримання встановлення плюс конструктори завершують визначення сутностей для сумісності з фреймворком.

Транзакційна консистентність обробки повідомлень та оновлення метрик забезпечується через методи сервісу Spring з анотацією @Transactional що атомарно виконують множину операцій бази даних. При отриманні пакету повідомлень сервіс спочатку витягує або створює сутність RunResult для поточного ідентифікатора запуску, потім пакетно зберігає сутності ProcessedSms через метод репозиторію saveAll що використовує пакетну обробку Hibernate для ефективності, та нарешті оновлює метрики RunResult через інкремент лічильника оброблених та перерахунок середніх. Вся операція виконується у єдиній транзакції бази даних що автоматично відкочується при будь-якому винятку гарантуючи консистентність. Для споживачів Kafka це означає що збереження зміщення відбувається лише після успішного завершення транзакції через транзакційну підтримку Spring Kafka, забезпечуючи семантику обробки принаймні один раз з обробкою ідемпотентності для потенційних дублікатів при повторах. Рівень ізоляції транзакції READ_COMMITTED балансує між консистентністю та продуктивністю для сценаріїв паралельної обробки, допускаючи неповторювані читання але запобігаючи брудним читанням незбережених даних. Оптимістичне блокування через поле з анотацією @Version може запобігти втраті оновлень при паралельних модифікаціях хоча малоймовірно у поточному випадку використання з окремим ідентифікатором запуску на потік виконання.

Денормалізація даних через дублювання полів з таблиці sms у processed_sms є свідомим архітектурним рішенням для підвищення автономності сервісу обробки та усунення міжсервісних залежностей бази даних. Хоча це порушує традиційні принципи нормалізації бази даних та створює надлишковість даних, переваги у вигляді спрощених запитів, усунення мережевої затримки для віддалених викликів бази даних, та стійкості до недоступності сервісу генерації перевагують недоліки збільшеного накладу зберігання. У контексті високооб'ємної обробки SMS вартість зберігання є незначною порівняно з операційною складністю потенційних розподілених об'єднань або двоетапних запитів. Кінцева консистентність між таблицями sms та processed_sms є прийнятною оскільки вони служать різним цілям: sms для тимчасової проміжної області очікування обробки, processed_sms для постійного архіву успішно оброблених повідомлень. Очищення старих записів sms після успішної обробки не порушує цілісність даних processed_sms що залишається самодостатнім повним записом. Цей компроміс приклад принципу проектування мікросервісів віддачі переваги дублюванню над тісним зв'язуванням для досягнення незалежної розгортаємості та масштабованості.

Схема бази даних включає відповідно визначені обмеження зовнішнього ключа для референційної цілісності між пов'язаними таблицями. У таблиці processed_sms колонка run_id має обмеження зовнішнього ключа REFERENCES run_results(run_id) ON DELETE CASCADE що автоматично видаляє всі пов'язані повідомлення при видаленні запису тестового запуску, спрощуючи операції очищення та запобігаючи осиротілим записам. Альтернативно можна використовувати ON DELETE SET NULL якщо потрібно зберігати історичні дані повідомлень навіть після видалення статистики запуску, хоча менш практично у поточному сценарії орієнтованому на аналітику [31, с. 107-127]. Обмеження CHECK застосовують бізнес-правила безпосередньо на рівні бази даних: колонка channel у обох таблицях має CHECK (channel IN ('SYNC_HTTP',

'ASYNC_HTTP', 'KAFKA')) для дозволу лише дійсних значень переліку, числові поля типу processed та failed мають обмеження CHECK для невід'ємності та логічної консистентності типу CHECK (processed >= 0 AND failed >= 0 AND processed + failed <= total). Ці обмеження надають фінальну захисну сітку проти недійсних даних що можуть обійти валідацію рівня додатку через помилки або пряме маніпулювання базою даних, хоча первинна валідація правильно відбувається на сервісному шарі.

Оптимізація запитів агрегації метрик виконується через ефективні SQL-патерни та використання обчислень на стороні бази даних для мінімізації передачі даних мережею. Розрахунок фінальних метрик для RunResult після завершення обробки виконується через єдиний запит агрегації SELECT COUNT(*), MIN(processed_at), MAX(processed_at) FROM processed_sms WHERE run_id = ? що одночасно отримує кількість оброблених повідомлень та часові межі для обчислення загального часу. Середній час обробки розраховується як різниця між максимальною та мінімальною міткою часу поділена на кількість безпосередньо у запиті бази даних замість витягування всіх індивідуальних міток часу до пам'яті додатку для обчислення. Використання віконних функцій бази даних може далі оптимізувати складні аналітичні запити типу накопичувальних підсумків або обчислень процентилів для розширеного аналізу продуктивності. Матеріалізовані представлення для часто доступних агрегацій типу підсумку порівняння каналів можуть драматично покращити продуктивність запитів за рахунок злегка застарілих даних що прийнятно для цілей звітування не в реальному часі, з періодичним оновленням через заплановані завдання або тригери на оновлення базових таблиць.

Конфігурація пулу з'єднань є критичною для підтримки оптимальної продуктивності бази даних під високою паралельністю. Spring Boot автоматично налаштовує пул з'єднань HikariCP з розумними стандартними значеннями але

продакшн-розгортання потребують ретельного налаштування на основі доступних з'єднань бази даних та очікуваного паралельного навантаження. Максимальний розмір пулу має балансувати між наданням достатніх з'єднань для пікового навантаження та уникненням перевантаження бази даних з надто багатьма паралельними з'єднаннями що можуть фактично зменшити продуктивність через накладні витрати перемикання контексту. Емпіричне правило пропонує розмір пулу близький до кількості процесорних ядер сервера бази даних (типово 10-50 з'єднань для помірних навантажень) з таймаутом з'єднання налаштованим достатньо високо для тимчасових піків навантаження але достатньо низько для запобігання безстроковому блокуванню. Моніторинг метрик використання пулу через кінцеві точки Actuator дозволяє ідентифікувати вузькі місця як недостатній розмір пулу (високий час очікування) чи неефективні патерни запитів (довго утримувані з'єднання), дозволяючи рішення оптимізації на основі даних. Кешування підготовлених виразів на рівні з'єднання зменшує накладні витрати розбору для багаторазово виконуваних запитів що часто відбувається у конвеєрі обробки з параметризованими виразами INSERT.

Масштабування ємності бази даних для обробки зростаючих обсягів даних та пропускної здатності може відбуватися через вертикальне масштабування (потужніше обладнання) або техніки горизонтального масштабування. Вертикальне масштабування є найпростішим підходом що включає оновлення до більшого інстансу бази даних з більшою кількістю процесорних ядер, пам'яті, та швидшого сховища, типово достатньо для досягнення десятків тисяч транзакцій на секунду з правильно оптимізованими запитами та індексами. Горизонтальне масштабування через сегментацію бази даних розподіляє дані між множиною інстансів бази даних на основі ключа сегментації, наприклад розподіл processed_sms за хешем run_id для маршрутизації запитів до відповідного сегмента. Це суттєво складніше

вимагаючи логіки маршрутизації на рівні додатку або проміжного програмного забезпечення бази даних але дозволяє майже необмежену масштабованість. Репліки для читання можуть розвантажити запити звітування від первинної бази даних для покращення продуктивності транзакційного навантаження, з асинхронною реплікацією що надає кінцево консистентні копії лише для читання для аналітичних цілей. Розподілені SQL-бази даних типу CockroachDB або YugabyteDB надають автоматичну сегментацію та багаторегіональну реплікацію з сильнішими гарантіями консистентності порівняно з традиційними схемами реплікації, хоча за ціну збільшеної операційної складності та потенційних накладних витрат продуктивності протоколів розподіленого консенсусу.

Стратегії резервного копіювання та відновлення після катастроф є істотними для продакшн-розгортань гарантуючи стійкість даних та безперервність бізнесу. Фізичні резервні копії PostgreSQL через pg_basebackup створюють повну копію всього кластера бази даних що може бути відновлена до консистентного стану у випадку катастрофічної відмови, типово планується щоденно з утриманням множини поколінь для опцій відновлення до точки в часі. Архівування журналу випереджувального запису дозволяє інкрементні резервні копії та безперервне архівування для мінімізації потенційної втрати даних, з цільовою точкою відновлення типово під 15 хвилин для правильно налаштованих установок. Логічні резервні копії через pg_dump надають агностичний до бази даних формат експорту що корисно для міграції між версіями PostgreSQL або відновлення окремих таблиць, хоча повільніше та більше порівняно з бінарними резервними копіями. Автоматизоване тестування процедур відновлення резервних копій критично для валідації цілісності резервних копій та забезпечення відновлюваності коли фактично потрібно - нетестовані резервні копії по суті марні. Реплікація до географічно розподілених резервних серверів надає додатковий захист проти регіональних катастроф з

механізмами автоматичного переключення для мінімізації простою, хоча за ціну збільшеної складності інфраструктури та тягаря обслуговування [32].

Політики утримання даних визначають як довго історичні дані мають зберігатися перед архівацією чи видаленням для балансу між аналітичними потребами та вартістю зберігання. SMS-повідомлення у таблиці processed_sms можуть агресивно очищатися після агрегації метрик у run_results з типовим періодом утримання 30-90 днів достатнім для налагодження недавніх проблем. Аналітичні дані run_results зазвичай утримуються довше (6-12 місяців або постійно якщо зберігання дозволяє) для довгострокового аналізу трендів та планування ємності. Розподіл часових рядів дозволяє ефективне архівування найстаріших даних до холодного сховища типу S3 або Glacier через дамп старих розділів перед їх видаленням з баз даних, драматично зменшуючи розмір активної бази даних при збереженні історичної запитуваності через відновлення архівованих дампів. Вимоги відповідності (наприклад обмеження утримання даних GDPR) можуть мандатувати максимальні періоди утримання незалежно від операційних потреб вимагаючи автоматизованих процесів очищення що забезпечують законне відповідність. Чітка документація політик утримання та застосування через автоматизовані завдання критично для уникнення неконтрольованого зростання бази даних що врешті впливає на продуктивність та збільшує вартість інфраструктури.

Розроблена модель даних та схема баз даних забезпечують міцну основу для надійної високопродуктивної обробки SMS-повідомлень у спроектованій мікросервісній системі. Розділення concerns між операційними базами даних сервісів генерації та обробки дозволяє їх незалежне масштабування та еволюцію відповідно специфічним характеристикам навантаження: таблиця sms сервісу генерації оптимізована для високопродуктивних вставок генерованих повідомлень з мінімальними накладними витратами, тоді як processed_sms та run_results сервісу обробки структуровані для ефективних запитів агрегації та

аналітичної звітності. Денормалізація даних жертвує ефективністю зберігання заради операційної простоти та автономії сервісу що зазвичай правильний компроміс у мікросервісних архітектурах. Комплексна стратегія індексування балансує продуктивність запитів проти накладних витрат запису забезпечуючи швидку поведінку системи під важким паралельним навантаженням. Правильно налаштовані транзакційні межі гарантують консистентність даних при паралельній обробці повідомлень множиною споживачів. Масштабованість схеми від тисяч до мільйонів повідомлень через розподіл, сегментацію, та технології розподілених баз даних надає чіткий шлях зростання для майбутнього розширення. Надійні політики резервного копіювання та утримання забезпечують стійкість даних та відповідність регуляторним вимогам завершуючи надійну готову до продакшн архітектуру даних.

Висновки

У другому розділі виконано комплексне проектування системи обробки SMS-повідомлень на базі мікросервісної архітектури з детальною специфікацією структури компонентів, механізмів комунікації та моделі даних.

Розроблена архітектура базується на розділенні функціональності між двома автономними мікросервісами де сервіс генерації забезпечує взаємодію з користувачем та створення тестових даних, а сервіс обробки концентрується на прийомі, валідації та надійному збереженні повідомлень з веденням детальної статистики. Така декомпозиція забезпечує високу модульність, незалежне масштабування компонентів та спрощує тестування окремих частин системи. Користувацький інтерфейс надає можливість вибору між трьома каналами доставки з відображенням статистики у реальному часі та автоматизованим порівняльним аналізом результатів.

Спроектовані механізми комунікації демонструють принципово різні характеристики де синхронний HTTP забезпечує найнижчу затримку окремих операцій але обмежену пропускну здатність, асинхронний HTTP драматично покращує продуктивність через паралелізм, а Kafka забезпечує найвищу надійність та масштабованість завдяки постійному зберіганню та природній підтримці зворотного тиску. Детальний аналіз компромісів дозволяє обґрунтовано обирати оптимальний метод залежно від специфічних вимог.

Розроблена модель даних з окремими базами для кожного сервісу гарантує автономність компонентів та їх незалежну еволюцію. Денормалізація даних свідомо жертвує ефективністю зберігання заради операційної простоти та усунення міжсервісних залежностей. Комплексна стратегія індексування та правильно налаштовані транзакційні межі забезпечують швидку поведінку системи під важким паралельним навантаженням з гарантованою консистентністю даних створюючи міцну основу для практичної реалізації та тестування.

РОЗДІЛ 3

РЕАЛІЗАЦІЯ ТА ТЕСТУВАННЯ СИСТЕМИ ОБРОБКИ SMS

3.1. Розроблення сервісу генерації SMS та користувацького інтерфейсу для запуску тестових сценаріїв

Реалізація сервісу генерації SMS-повідомлень розпочинається зі створення базової структури Spring Boot додатку з необхідними залежностями для веб-розробки, роботи з базою даних та клієнтської комунікації. Файл конфігурації Maven містить залежності spring-boot-starter-web для REST контролерів та вбудованого Tomcat сервера, spring-boot-starter-data-jpa для об'єктно-реляційного відображення, spring-boot-starter-thymeleaf для серверного рендерингу HTML-шаблонів, postgresql драйвер для підключення до бази даних, та spring-cloud-starter-openfeign для декларативних HTTP-клієнтів. Структура проекту організована згідно з типовою багатошаровою архітектурою Spring з чітким розділенням на пакети controller для обробки HTTP-запитів, service для бізнес-логіки, repository для доступу до даних, model для доменних сутностей, dto для об'єктів передачі даних, та config для конфігураційних класів. Головний клас додатку анотується @SpringBootApplication для автоматичної конфігурації та сканування компонентів, а також @EnableFeignClients для активації підтримки Feign-клієнтів. Файл application.properties містить налаштування підключення до бази даних PostgreSQL включаючи URL, ім'я користувача, пароль, та параметри пулу з'єднань HikariCP для оптимальної продуктивності.

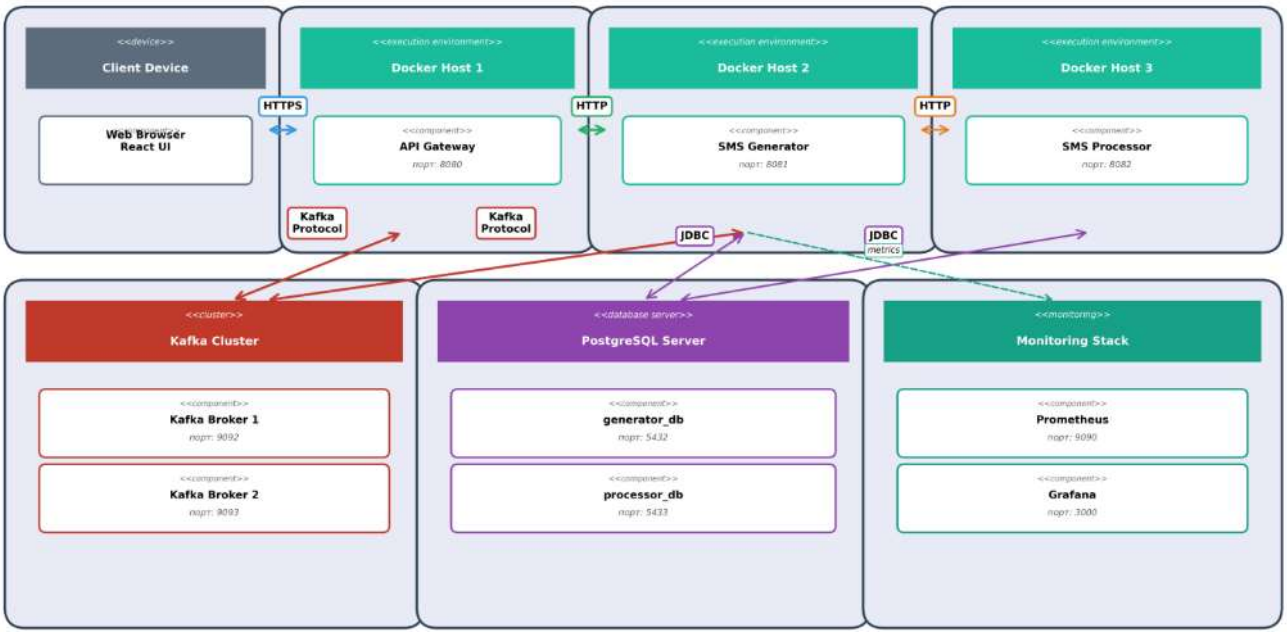

Рис. 3.1 – Діаграма розгортання інфраструктури мікросервісної системи обробки SMS

Доменна модель сервісу генерації представлена JPA-сутністю Sms з полями що відповідають спроектованій схемі бази даних. Клас анотується @Entity для позначення як сутність JPA та @Table(name = "sms") для явного вказання назви таблиці. Поле id типу UUID анотується @Id для первинного ключа та @GeneratedValue(generator = "uuid2") разом з @GenericGenerator для автоматичної генерації унікальних ідентифікаторів при збереженні нових сутностей. Поля sender та receiver типу String мають анотації @Column(nullable = false, length = 15) для забезпечення обов'язковості та обмеження довжини на рівні схеми бази даних. Поле text анотується @Column(columnDefinition = "TEXT") для підтримки необмеженої довжини текстового вмісту повідомлень. Поле createdAt типу LocalDateTime має анотації @Column(nullable = false, updatable = false) для незмінної мітки часу створення з автоматичним заповненням через @PrePersist колбек-метод або стандартне значення бази даних [33]. Клас використовує анотації Lombok @Data для автоматичної

генерації методів отримання, встановлення, equals, hashCode та toString, що значно скорочує шаблонний код.

Репозиторій для роботи з сутністю Sms визначається як інтерфейс SmsRepository що розширює JpaRepository<Sms, UUID> надаючи повний набір методів створення, читання, оновлення та видалення без необхідності їх явної реалізації. Spring Data JPA автоматично генерує реалізацію цього інтерфейсу під час виконання з підтримкою стандартних операцій типу save, findById, findAll, deleteById та багатьох інших. Для підтримки посторінкового виведення у веб-інтерфейсі додається користувацький метод запиту findAllByOrderByCreatedAtDesc(Pageable pageable) що використовує конвенцію назв методів Spring Data для автоматичної генерації запиту відсортованого за датою створення у спадному порядку з підтримкою пагінації. Додатковий метод countBy() повертає загальну кількість записів у таблиці що необхідно для обчислення кількості сторінок у інтерфейсі користувача. Анотація @Repository на інтерфейсі не є обов'язковою оскільки Spring Data автоматично розпізнає розширення JpaRepository, але може бути додана для явності та кращої читабельності коду. Репозиторій автоматично працює в транзакційному контексті забезпечуючи консистентність операцій з базою даних.

Генератор випадкових SMS-повідомлень реалізується як окремий компонент GeneratorService з методами для створення реалістичних тестових даних. Сервіс використовує Java Random для генерації псевдовипадкових значень з можливістю встановлення seed для відтворюваності результатів під час тестування. Метод генерації номерів телефонів створює коректні українські мобільні номери у форматі +380XXXXXXXXX де перші дві цифри після коду країни вибираються з реального діапазону операторів (39, 50, 63, 66, 67, 68, 73, 91, 92, 93, 94, 95, 96, 97, 98, 99), а решта дев'ять цифр генеруються випадково. Генератор текстового вмісту повідомлень використовує попередньо визначений масив шаблонів типових SMS-повідомлень (привітання, нагадування,

сповіщення, рекламні оголошення) з випадковою підстановкою параметрів типу імен, дат, сум для різноманітності. Довжина тексту обмежується 160 символами згідно зі стандартним розміром одиничного SMS-повідомлення. Метод generateBatch(int count) створює задану кількість унікальних SMS-об'єктів у циклі, встановлює для кожного згенеровані відправник, отримувач, текст та поточну мітку часу створення. Згенеровані об'єкти зберігаються пакетно через репозиторій для оптимізації продуктивності вставки великих обсягів даних.

Сервісний шар відповідальний за бізнес-логіку генерації та відправки повідомлень реалізується класом SmsService з анотацією @Service для автоматичного виявлення Spring. Сервіс інкапсулює складну логіку створення повідомлень, пакетування для відправки та взаємодії з різними каналами комунікації. Метод generateMessages(int count) делегує генерацію даних компоненту GeneratorService та зберігає результати через SmsRepository у єдиній транзакції позначеній анотацією @Transactional. Для підтримки посторінкового виведення реалізовано метод getMessages(int page, int size) що використовує PageRequest.of для створення об'єкта Pageable з параметрами номера сторінки та розміру, викликає репозиторій для отримання відповідної порції даних та повертає Page об'єкт з вмістом та метаданими пагінації. Окремі методи sendViaSync, sendViaAsync та sendViaKafka інкапсулюють логіку відправки повідомлень через відповідні канали доставки з параметрами кількості повідомлень для відправки [34]. Кожен метод створює унікальний ідентифікатор запуску UUID, витягує потрібну кількість повідомлень з бази даних, пакетує їх у групи фіксованого розміру та передає до відповідного клієнта комунікації. Сервіс логує важливі події типу початку та завершення генерації, відправки пакетів повідомлень для спрощення моніторингу та налагодження.

Feign-клієнт для синхронної та асинхронної HTTP-комунікації з сервісом обробки визначається як інтерфейс ProcessorClient з анотацією @FeignClient(name = "processor-service", url = "${processor.service.url}").

Параметр name задає логічне ім'я клієнта а url читається з конфігураційного файлу для гнучкості налаштування різних середовищ розробки, тестування та продакшн. Інтерфейс декларує метод sendBatch з анотацією @PostMapping("/messages/batch") що відображається на відповідну кінцеву точку сервісу обробки. Метод приймає параметр @RequestBody List<SmsDto> messages для автоматичної серіалізації списку об'єктів передачі даних у JSON-формат та параметр @RequestHeader для передачі ідентифікатора запуску у заголовках HTTP. Клас SmsDto є простим об'єктом передачі даних з полями що дублюють сутність Sms але без JPA-анотацій для незалежності від деталей персистентності. Feign автоматично обробляє серіалізацію та десеріалізацію JSON через Jackson, управляє таймаутами з'єднання та читання, та генерує винятки при помилках HTTP для обробки на рівні сервісного шару. Конфігурація Feign включає налаштування таймаутів connectTimeout=5000 та readTimeout=60000 мілісекунд для балансу між швидкою відмовою та достатнім часом для обробки великих пакетів повідомлень.

Реалізація асинхронної відправки через HTTP використовує ExecutorService для управління пулом робочих потоків що виконують паралельні запити. Конфігураційний клас AsyncConfig створює бін ThreadPoolTaskExecutor з налаштованими параметрами corePoolSize=10 для базової кількості потоків, maxPoolSize=50 для максимальної кількості при піках навантаження, queueCapacity=100 для черги очікуючих завдань, та threadNamePrefix="async-http-" для зручності ідентифікації потоків у логах та моніторингу. Метод sendViaAsync у сервісі створює список завдань CompletableFuture де кожне завдання відправляє один пакет повідомлень через Feign-клієнт з анотацією @Async для асинхронного виконання. Після створення всіх завдань метод використовує CompletableFuture.allOf для очікування завершення всіх паралельних операцій з можливістю обробки виключень через exceptionally або handle. Результати виконання агрегуються для підрахунку

успішно відправлених пакетів та помилок з детальним логуванням для аналізу продуктивності. Правильне налаштування розміру пулу потоків критично впливає на баланс між паралелізмом та споживанням ресурсів, тому значення обираються емпірично через навантажувальне тестування реальної системи.

Інтеграція Kafka для асинхронної відправки повідомлень реалізується через Spring Kafka з конфігурацією виробника у класі KafkaProducerConfig. Конфігурація визначає бін KafkaTemplate з налаштуваннями адреси bootstrap-серверів Kafka, серіалізатора ключів StringSerializer та серіалізатора значень JsonSerializer для автоматичного перетворення Java-об'єктів у JSON-формат. Додаткові параметри виробника включають acks=all для максимальної надійності доставки з підтвердженням від усіх синхронізованих реплік, retries=3 для автоматичних повторів при тимчасових помилках, batch-size=16384 байт для групування записів у пакети, linger-ms=10 для невеликої затримки накопичення пакета, та compression-type=snappy для ефективного стиснення корисного навантаження. Метод sendViaKafka у сервісі використовує KafkaTemplate.send для публікації кожного повідомлення у налаштований топік з опціональним ключем розподілу для контролю маршрутизації повідомлень до конкретних розділів. Метод повертає ListenableFuture з колбеками для обробки успішної відправки або помилок через addCallback що дозволяє асинхронне відстеження результатів без блокування основного потоку виконання. Логування успішних публікацій та помилок надає видимість стану відправки для оперативного моніторингу продуктивності каналу Kafka.

Веб-контролер для обробки HTTP-запитів від браузера користувача реалізується класом WebController з анотацією @Controller для рендерингу HTML-сторінок через Thymeleaf. Контролер визначає кінцеві точки для головної сторінки генерації @GetMapping("/") що повертає назву шаблону "index" з передачею моделі даних для відображення, кінцеву точку генерації @PostMapping("/generate") що приймає параметр кількості повідомлень з форми,

викликає сервіс генерації та перенаправляє на головну сторінку з повідомленням про успіх. Окремі кінцеві точки для запуску відправки через різні канали @PostMapping("/send/sync"), @PostMapping("/send/async"), @PostMapping("/send/kafka") приймають параметр кількості повідомлень для відправки, викликають відповідний метод сервісу та повертають ідентифікатор запуску для відстеження прогресу. REST-контролер ApiController з анотацією @RestController обробляє AJAX-запити від JavaScript для отримання списку повідомлень з пагінацією @GetMapping("/api/messages") що повертає JSON-відповідь з даними сторінки та метаданими. Кінцева точка отримання статистики прогресу @GetMapping("/api/progress/{runId}") проксіює запити до сервісу обробки через окремий Feign-клієнт для витягування поточного стану обробки конкретного запуску. Обробка виключень централізована через @ControllerAdvice клас з методами обробки специфічних типів помилок та повернення відповідних HTTP-статусів та повідомлень користувачу.

Користувацький інтерфейс реалізований як Thymeleaf-шаблони з інтеграцією Bootstrap для адаптивного дизайну та jQuery для динамічної взаємодії. Головний шаблон index.html містить форму генерації з полем вводу числа повідомлень та кнопкою відправки, таблицю відображення згенерованих SMS з колонками відправник, отримувач, текст та час створення, елементи управління пагінацією для навігації між сторінками даних. Три окремі кнопки запуску відправки позначені різними кольорами та іконками для візуального розрізнення каналів доставки: зелена для синхронного HTTP, помаранчева для асинхронного HTTP, синя для Kafka. Секція відображення прогресу містить динамічний прогрес-бар Bootstrap з анімацією, текстові поля для відображення кількості оброблених та невдалих повідомлень, показник середнього часу обробки та загального часу виконання що оновлюються в реальному часі. JavaScript-код виконує періодичне опитування кінцевої точки прогресу кожну секунду через AJAX після ініціювання відправки, оновлює елементи інтерфейсу

з отриманими даними та зупиняє опитування коли поле завершеності вказує на закінчення обробки. Сторінка порівняння comparison.html відображає таблицю результатів з рядками для кожного завершеного запуску та автоматично згенерованими висновками про найшвидший та найстабільніший канал на основі аналізу метрик [35].

Налагодження та логування налаштовуються через Logback конфігурацію для детального відстеження роботи додатку у різних середовищах. Конфігурація визначає різні аппендери для виводу логів: ConsoleAppender для виводу у консоль під час розробки з кольоровим форматуванням для кращої читабельності, RollingFileAppender для збереження у файли з ротацією за розміром та часом для продакшн-середовища. Рівні логування налаштовуються окремо для різних пакетів: DEBUG для власного коду додатку під час розробки для детального трасування виконання, INFO для Spring Framework та Hibernate для балансу між інформативністю та обсягом логів, WARN для сторонніх бібліотек для фільтрації несуттєвих повідомлень. Структуроване логування у JSON-форматі може бути налаштоване для продакшн-середовища через logstash-logback-encoder що полегшує централізовану агрегацію та аналіз логів у системах типу ELK стек. MDC контекст використовується для додавання ідентифікаторів кореляції до всіх лог-повідомлень у межах одного запиту що спрощує відстеження потоку виконання через множину компонентів системи. Метрики додатку експонуються через Spring Boot Actuator endpoints включаючи health для перевірки здоров'я, metrics для збору показників продуктивності, info для метаданих додатку, що дозволяє інтеграцію з системами моніторингу типу Prometheus та Grafana.

Юніт-тестування компонентів сервісу генерації виконується через JUnit 5 та Mockito для ізоляції тестованих класів від залежностей. Тестовий клас SmsServiceTest використовує анотації @ExtendWith(MockitoExtension.class) для ініціалізації Mockito, @Mock для створення заглушок репозиторію та клієнтів

комунікації, @InjectMocks для автоматичного впровадження заглушок у тестований сервіс. Тестові методи перевіряють коректність генерації повідомлень через верифікацію викликів методів репозиторію з очікуваними параметрами, правильність пакетування повідомлень для відправки через перевірку розмірів пакетів та їх змісту, обробку помилок через симуляцію виключень заглушками та перевірку відповідного поширення або обгортання виключень. Інтеграційні тести з анотацією @SpringBootTest запускають повний контекст Spring для тестування взаємодії реальних компонентів з вбудованою базою даних H2 замість PostgreSQL для швидшості та ізоляції. Тести перевіряють наскрізні сценарії від HTTP-запиту через контролер до збереження у базу даних та відповіді користувачу. Тестування веб-шару виконується через MockMvc для симуляції HTTP-запитів без запуску повного веб-сервера з верифікацією статусів відповідей, вмісту тіл відповідей та заголовків.

Контейнеризація сервісу генерації виконується через багатоетапний Dockerfile що оптимізує розмір фінального образу та швидкість збірки. Перший етап використовує maven:3.8-openjdk-11 як базовий образ для компіляції вихідного коду, копіює файли pom.xml та src директорію, виконує mvn clean package для створення виконуваного JAR-файлу з усіма залежностями. Другий етап починається з мінімального openjdk:11-jre-slim образу що містить лише середовище виконання Java без інструментів розробки для зменшення розміру образу, копіює зібраний JAR-файл з попереднього етапу, встановлює змінні середовища для конфігурації JVM та додатку, експонує порт 8080 для вхідних HTTP-з'єднань, визначає ENTRYPOINT для запуску Spring Boot додатку. Docker Compose файл описує сервіс генерації разом з його залежностями: контейнер PostgreSQL з налаштованими змінними середовища для ім'я бази даних, користувача та пароля, volume для персистентності даних бази між перезапусками, мережу для комунікації між контейнерами [36-37]. Сервіс генерації налаштований з depends_on для гарантування запуску бази даних

перед додатком, змінними середовища що перевизначають налаштування підключення до бази для використання імені контейнера як хоста, портом мапінгу для доступу ззовні (рис.3.2-3.3).

*docs.txt: Блокнот

Файл Редагування Формат Вигляд Довідка

```
docker logs service-a
docker-compose restart <service-a>

docker-compose up --build
docker-compose down
docker-compose stop service-b

видалити всі контейнери
docker ps -a

Вимкнути все
docker-compose down

docker rmi sms-lab-project-service-b
docker-compose build --no-cache service-b
docker-compose up -d

Щоб вимкнути клієнтську частину (frontend): CTRL + C
```

Рис. 3.2 – Набір основних Docker та Docker Compose команд, що використовуються для керування мікросервісами під час розробки та локального розгортання системи

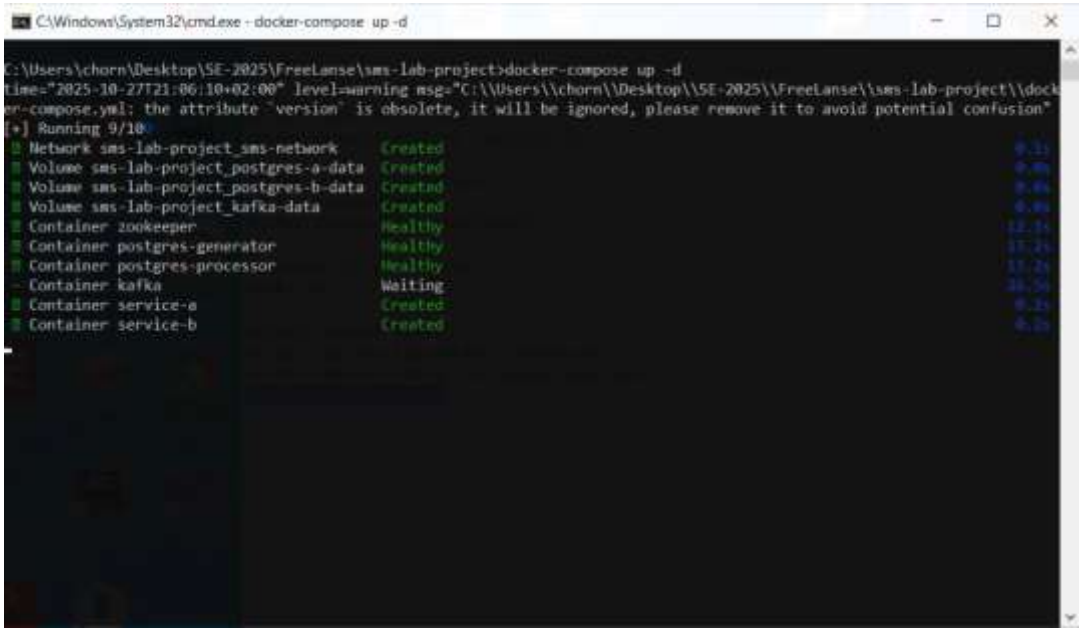

Рис. 3.3 – Процес розгортання мікросервісної інфраструктури за допомогою команди docker-compose up з відображенням створення мереж, томів та контейнерів

Команда docker-compose up --build створює образи, запускає всі сервіси та виводить агреговані логи для зручного моніторингу стану системи.

3.2. Реалізація сервісу обробки повідомлень з підтримкою трьох каналів доставки (Sync HTTP, Async HTTP, Kafka)

Реалізація сервісу обробки SMS-повідомлень починається зі створення Spring Boot додатку з залежностями spring-boot-starter-web для REST API, spring-boot-starter-data-jpa для роботи з базою даних, postgresql драйвер для підключення до PostgreSQL, spring-kafka для інтеграції споживача Kafka. Структура проекту організована аналогічно сервісу генерації з пакетами для контролерів, сервісів, репозиторіїв, моделей та конфігурації для підтримки чистої архітектури та розділення відповідальності. Конфігураційний файл application.properties містить налаштування підключення до власної бази даних processor_db з окремими обліковими даними, параметри споживача Kafka включаючи адреси bootstrap-серверів, ідентифікатор групи споживачів, десеріалізатори ключів та значень, стратегію автоматичного збереження зміщень. Додаткові налаштування включають розмір пулу з'єднань до бази даних, параметри пакетної обробки JPA для оптимізації продуктивності вставок, налаштування логування для моніторингу роботи додатку. Головний клас додатку анотується @SpringBootApplication для автоматичної конфігурації Spring Boot та @EnableKafka для активації підтримки анотацій Kafka Listener.

Доменна модель сервісу обробки складається з двох JPA-сутностей що відображаються на таблиці бази даних. Сутність ProcessedSms містить усі поля

вихідного повідомлення плюс додаткові метадані обробки: id як первинний ключ UUID, sender та receiver для номерів телефонів, text для вмісту повідомлення, createdAt для оригінального часу створення переданого від сервісу генерації, processedAt для моменту завершення обробки у поточному сервісі з автоматичним заповненням через анотацію @PrePersist, runId для зв'язку з конкретним тестовим запуском, channel для ідентифікації каналу надходження як enum DeliveryChannel з можливими значеннями SYNC_HTTP, ASYNC_HTTP, KAFKA. Сутність анотована @Entity та @Table(name = "processed_sms") з відповідними Column анотаціями для забезпечення обов'язковості полів та обмежень довжини. Зв'язок @ManyToOne до сутності RunResult встановлюється через поле runResult з @JoinColumn(name = "run_id") для навігації від повідомлення до його запуску. Сутність RunResult агрегує фінальну статистику запуску з полями runId як первинний ключ, channel для типу каналу, total для загальної кількості повідомлень, processed та failed для лічильників результатів обробки, averageProcessingMs та totalTimeMs для метрик продуктивності, startedAt та finishedAt для часових меж, notes для опціональних приміток. Двонаправлений зв'язок @OneToMany до ProcessedSms встановлюється через поле messages з mappedBy="runResult" для навігації від запуску до його повідомлень [38].

Репозиторії для доступу до даних визначаються як інтерфейси що розширюють JpaRepository надаючи стандартні CRUD операції та користувацькі методи запитів. ProcessedSmsRepository оголошує методи findByRunId(UUID runId) для витягування всіх повідомлень конкретного запуску, countByRunId(UUID runId) для підрахунку оброблених повідомлень, findFirstByRunIdOrderByProcessedAtAsc та findFirstByRunIdOrderByProcessedAtDesc для отримання першого та останнього оброблених повідомлень необхідних для розрахунку загального часу виконання. RunResultRepository надає метод findByChannel(DeliveryChannel channel) для

фільтрації результатів за типом каналу, findAllByOrderByFinishedAtDesc для витягування останніх завершених запусків відсортованих за часом, existsByRunId для перевірки існування запуску перед створенням. Користувацькі запити можуть бути визначені через анотацію @Query з JPQL або нативним SQL для складніших операцій агрегації що не виражаються через конвенції назв методів. Обидва репозиторії підтримують пагінацію через параметр Pageable у сигнатурах методів для ефективного витягування великих обсягів даних порціями замість завантаження всього набору результатів у пам'ять. Транзакційний контекст автоматично застосовується до всіх методів репозиторію забезпечуючи консистентність операцій з базою даних.

REST контролер для обробки HTTP-запитів від сервісу генерації реалізується класом MessageController з анотацією @RestController та @RequestMapping("/messages"). Кінцева точка прийому пакету повідомлень @PostMapping("/batch") приймає параметр @RequestBody List<SmsDto> messages з автоматичною десеріалізацією JSON, @RequestHeader("X-Run-Id") UUID runId для ідентифікатора запуску переданого у заголовках, @RequestHeader("X-Channel") String channel для типу каналу відправки. Метод валідує вхідні дані через @Valid анотацію на параметрі DTO що активує Bean Validation перевірки анотацій @NotNull, @Size на полях класу SmsDto. Після валідації контролер делегує обробку сервісному шару через виклик processMessages передаючи десеріалізовані об'єкти, ідентифікатор запуску та канал. У випадку успішної обробки повертається HTTP-статус 200 OK з порожнім тілом відповіді, при помилках валідації 400 Bad Request з деталями помилок у тілі відповіді, при внутрішніх помилках 500 Internal Server Error з загальним повідомленням без розкриття чутливих деталей реалізації. Обробка виключень централізована через @ControllerAdvice клас з методами для різних типів помилок що забезпечує консистентність відповідей API та спрощує налагодження для клієнтів [39]. Логування всіх вхідних запитів та відповідей

виконується через фільтр або аспект для аудиту та моніторингу навантаження на систему.

Споживач Kafka для асинхронного прийому повідомлень реалізується класом KafkaMessageConsumer з методом анотованим @KafkaListener. Анотація налаштовується з параметрами topics="kafka.topic.messages"длячитанняназвитопікузконфігурації,groupId="{kafka.topic.messages}" для читання назви топіку з конфігурації, groupId="

kafka.topic.messages"длячитанняназвитопікузконфігурації,groupId="{kafka.consumer.group-id}" для ідентифікатора групи споживачів, containerFactory="kafkaListenerContainerFactory" для користувацької фабрики контейнерів з налаштованими параметрами десеріалізації та обробки помилок. Метод consume приймає параметр @Payload SmsDto message з автоматичною десеріалізацією з JSON, @Header(KafkaHeaders.RECEIVED_MESSAGE_KEY) String runId для ідентифікатора запуску переданого як ключ повідомлення, @Header("channel") String channel для каналу доставки з користувацького заголовка. Споживач викликає той самий сервісний метод обробки що і HTTP контролер забезпечуючи єдину логіку незалежно від джерела надходження повідомлень. Обробка помилок налаштовується через SeekToCurrentErrorHandler що автоматично повторює обробку повідомлень при транзієнтних помилках з експоненційною затримкою та відправляє хронічно проблемні повідомлення у dead letter topic після вичерпання спроб. Ручне управління зміщеннями може бути налаштоване через enable-auto-commit=false та явне збереження через Acknowledgment.acknowledge() після успішної обробки для гарантій at-least-once семантики. Моніторинг споживача включає логування отриманих повідомлень, відставання від останнього зміщення у розділах, та швидкості обробки для виявлення проблем продуктивності.

Сервісний шар ProcessingService інкапсулює бізнес-логіку обробки повідомлень та оновлення метрик у транзакційних методах. Метод

processMessages помічений анотацією @Transactional що забезпечує атомарне виконання всіх операцій з базою даних з автоматичним відкотом при виникненні виключень. Логіка починається з перевірки існування або створення сутності RunResult для поточного ідентифікатора запуску через runResultRepository.findById або створення нового з початковими значеннями лічильників та часу старту. Кожен SmsDto конвертується у сутність ProcessedSms з копіюванням всіх полів та встановленням додаткових метаданих runId, channel та processedAt з поточним часом. Список сутностей зберігається пакетно через processedSmsRepository.saveAll що використовує Hibernate batch insert для мінімізації кількості SQL-виразів та покращення продуктивності. Після успішного збереження RunResult оновлюється інкрементом лічильника processed на кількість оброблених повідомлень у пакеті. Якщо обробка завершує останній пакет повідомлень (коли processed досягає total), розраховуються фінальні метрики через запити до processedSmsRepository для отримання першого та останнього часу обробки, обчислюється різниця для totalTimeMs та середнє для averageProcessingMs, встановлюється finishedAt з поточним часом. Всі зміни RunResult зберігаються автоматично через dirty checking механізм JPA наприкінці транзакції [40].

Кінцева точка для отримання поточного прогресу обробки реалізується у MetricsController з методом @GetMapping("/metrics/summary"). Метод приймає параметр @RequestParam UUID runId для ідентифікації запуску та повертає DTO об'єкт MetricsSummary з агрегованими даними. Сервіс MetricsService витягує RunResult за ідентифікатором, перевіряє його існування та повертає 404 Not Found якщо запуск не знайдено, інакше будує об'єкт відповіді з полями runId, channel, total, processed, failed, averageProcessingMs, totalTimeMs, isCompleted обчислений як порівняння processed з total. Для незавершених запусків totalTimeMs може розраховуватись як різниця між поточним часом та startedAt для відображення тривалості виконання в реальному часі. Кінцева

точка не потребує аутентифікації для спрощення тестування але у продакшн-середовищі має бути захищена токенами або взаємним TLS для запобігання несанкціонованого доступу до метрик. Кешування відповідей може бути налаштоване через @Cacheable анотацію з коротким TTL для зменшення навантаження на базу даних при інтенсивному опитуванні з боку клієнтів хоча баланс між свіжістю даних та продуктивністю має бути ретельно зважений.

REST API для витягування збережених результатів попередніх запусків надає кінцеві точки для списку та деталей. Метод @GetMapping("/results") повертає посторінковий список всіх результатів з параметрами Pageable для контролю розміру сторінки та сортування, фільтрами за каналом через опціональний параметр @RequestParam(required = false) DeliveryChannel channel. Якщо фільтр вказаний, використовується метод репозиторію findByChannel, інакше findAll для витягування всіх записів. Результати маплюються на DTO об'єкти для приховування деталей реалізації та контролю структури відповіді API. Метод @GetMapping("/results/{runId}") приймає змінну шляху @PathVariable UUID runId та повертає детальний об'єкт результату для конкретного запуску або 404 Not Found якщо не знайдено. DTO RunResultDto містить всі поля агрегованої статистики плюс опціонально список перших десяти оброблених повідомлень для попереднього перегляду без завантаження повного набору що може містити мільйони записів [41]. Пагінація списку повідомлень може бути додана через вкладену кінцеву точку /results/{runId}/messages з власними параметрами Pageable для детального перегляду всіх оброблених повідомлень конкретного запуску. HATEOAS лінки можуть бути додані до відповідей для навігації між пов'язаними ресурсами згідно з принципами REST матурності але не є критичними для поточної реалізації орієнтованої на специфічний клієнт сервісу генерації.

Конфігурація Kafka споживача визначається у класі KafkaConsumerConfig що створює бін фабрики контейнера слухача з налаштуваннями десеріалізації та

обробки помилок. ConsumerFactory налаштовується з властивостями bootstrap-серверів для підключення до кластера Kafka, group-id для ідентифікації групи споживачів що дозволяє паралельну обробку між множиною інстансів, key-deserializer StringDeserializer для ключів повідомлень, value-deserializer JsonDeserializer для автоматичної конверсії JSON у Java об'єкти з налаштуванням trusted packages для безпеки десеріалізації. ConcurrentKafkaListenerContainerFactory налаштовується з concurrency=3 для запуску трьох паралельних споживачів у межах одного інстансу додатку що збільшує пропускну здатність для топіків з множиною розділів. AckMode налаштовується як MANUAL_IMMEDIATE для явного контролю збереження зміщень після успішної обробки кожного повідомлення гарантуючи at-least-once семантику. ErrorHandler налаштовується як SeekToCurrentErrorHandler з FixedBackOff для повторів з фіксованою затримкою та максимальною кількістю спроб перед відправкою у dead letter topic. Dead letter publishing налаштовується через DeadLetterPublishingRecoverer що автоматично публікує проблемні повідомлення у окремий топік для ручного аналізу та відновлення після усунення причин помилок.

Тестування сервісу обробки включає модульні тести для ізоляції компонентів та інтеграційні тести для перевірки взаємодії. ProcessingServiceTest використовує Mockito для створення заглушок репозиторіїв та тестування логіки обробки незалежно від бази даних з верифікацією викликів методів збереження та оновлення з очікуваними параметрами. Інтеграційні тести з @SpringBootTest та @AutoConfigureTestDatabase запускають повний контекст з вбудованою базою H2 для реальної персистентності даних у ізольованому середовищі. @EmbeddedKafka анотація запускає вбудований брокер Kafka для тестування споживача без залежності від зовнішньої інфраструктури з можливістю публікації тестових повідомлень через KafkaTemplate та верифікації їх обробки через перевірку стану бази даних. MockMvc використовується для тестування

REST контролерів з симуляцією HTTP-запитів та верифікацією статусів відповідей, вмісту JSON-тіл, заголовків без запуску повного веб-сервера. Навантажувальні тести через JMeter або Gatling симулюють реалістичне навантаження з тисячами паралельних запитів для виявлення вузьких місць продуктивності та витоків пам'яті, вимірюють пропускну здатність, латентність та швидкість помилок під різними рівнями стресу. Результати навантажувального тестування використовуються для налаштування параметрів пулу з'єднань, розмірів пулу потоків, та інших конфігурацій для оптимальної продуктивності продакшн-системи [42].

Моніторинг та observability сервісу обробки налаштовуються через Spring Boot Actuator з експозицією метрик для Prometheus. Actuator endpoints включають /actuator/health для перевірки здоров'я додатку та його залежностей типу бази даних та Kafka, /actuator/metrics для доступу до всіх зібраних метрик включаючи використання JVM пам'яті, статистику garbage collection, пул з'єднань до бази даних, /actuator/prometheus для експорту метрик у форматі Prometheus для скрапінгу. Користувацькі метрики додаються через MeterRegistry для відстеження бізнес-показників типу кількості оброблених повідомлень за секунду, розподілу часу обробки через гістограми, лічильників помилок за типами. Розподілене трасування налаштовується через Spring Cloud Sleuth що автоматично додає trace та span ідентифікатори до всіх логів та HTTP-заголовків для відстеження потоку запитів через систему. Інтеграція з Zipkin або Jaeger дозволяє візуалізацію повних трейсів з часовими діаграмами виконання різних компонентів для ідентифікації вузьких місць латентності. Алерти налаштовуються у Prometheus Alertmanager для сповіщення про критичні події типу високої швидкості помилок, великого відставання споживача Kafka, або недоступності бази даних з інтеграцією у канали сповіщень типу Slack або PagerDuty для швидкої реакції команди.

Контейнеризація сервісу обробки виконується аналогічно сервісу генерації через багатоетапний Dockerfile з етапами збірки Maven та виконання на мінімальному JRE образі. Docker Compose файл розширюється додаванням сервісу обробки з залежностями до власної бази даних PostgreSQL processor_db та спільного брокера Kafka. Сервіс Kafka визначається з образом confluentinc/cp-kafka з налаштуваннями кількості розділів за замовчуванням, фактора реплікації, портів для внутрішньої та зовнішньої комунікації. Zookeeper додається як залежність Kafka для управління метаданими кластера та координації брокерів. Мережа Docker налаштовується для забезпечення комунікації між усіма сервісами з використанням імен сервісів як DNS-імен для розв'язання адрес. Volumes визначаються для персистентності даних баз даних та Kafka між перезапусками контейнерів запобігаючи втраті даних при оновленнях або збоях. Health checks додаються до сервісів для верифікації готовності перед маршрутизацією трафіку з періодичними перевірками стану та налаштовуваними таймаутами [43]. Команда docker-compose up запускає всю інфраструктуру з правильною послідовністю через depends_on та health checks забезпечуючи що сервіси стартують лише після готовності їх залежностей.

3.3. Тестування продуктивності системи при різних обсягах навантаження та порівняння ефективності каналів доставки

Методологія тестування продуктивності розроблена для систематичного порівняння трьох каналів доставки повідомлень під різними обсягами навантаження з контрольованими умовами експерименту. План тестування визначає набори тестових сценаріїв з варіюванням кількості повідомлень від однієї тисячі для базового випадку до ста тисяч для стрес-тестування продуктивності під екстремальним навантаженням. Кожен сценарій виконується п'ять разів для кожного каналу доставки з розрахунком середніх значень та

стандартних відхилень метрик для статистичної значущості результатів. Між запусками виконується пауза тридцять секунд для стабілізації системи та очищення буферів запобігаючи взаємному впливу послідовних тестів через залишкові ефекти попереднього навантаження. Тестове середовище налаштовується з фіксованими апаратними ресурсами: віртуальна машина з чотирма процесорними ядрами, вісьма гігабайтами оперативної пам'яті, SSD-диском для бази даних та Kafka для мінімізації впливу дискового вводу-виводу на результати. Мережева затримка між сервісами симулюється на рівні одного-двох мілісекунд для імітації реалістичних умов локальної мережі дата-центру. Моніторинг системних ресурсів виконується паралельно з тестами для відстеження використання процесора, пам'яті, дискового та мережевого вводу-виводу що дозволяє ідентифікувати вузькі місця та корелювати продуктивність з споживанням ресурсів [44].

Конфігурація системи для тестування стандартизується з однаковими налаштуваннями для всіх каналів де це можливо для справедливого порівняння. Розмір пакету повідомлень для HTTP-каналів встановлюється як п'ятсот SMS на запит для балансу між накладними витратами мережі та ризиком втрати даних при помилках. Пул потоків для асинхронного HTTP налаштовується з десятьма робочими потоками для достатнього паралелізму без надмірного споживання ресурсів та конкуренції. Таймаути з'єднання та читання для HTTP-клієнтів встановлюються як п'ять та шістдесят секунд відповідно надаючи достатньо часу для обробки великих пакетів але швидко завершуючись при справжніх проблемах мережі. Kafka налаштовується з трьома розділами топіку для паралельної обробки трьома споживачами з розміром пакету виробника шістнадцять кілобайт, затримкою накопичення десять мілісекунд, компресією snappy для оптимізації мережевого трафіку. Пул з'єднань до бази даних налаштовується з двадцятьма максимальними з'єднаннями для обох сервісів забезпечуючи достатню паралельність без перевантаження PostgreSQL. JVM

параметри включають мінімальний та максимальний розмір купи чотири гігабайти, G1GC збирач сміття для прогнозованих пауз, налаштування розмірів регіонів для оптимізації під шаблон навантаження з короткоживучими об'єктами.

Базовий тест з однією тисячею повідомлень виконується для всіх трьох каналів з детальним збором метрик для встановлення базової продуктивності. Синхронний HTTP демонструє передбачувану продуктивність з лінійним зростанням часу виконання пропорційно кількості пакетів та затримці кожного запиту. При розмірі пакету п'ятсот повідомлень та затримці відповіді близько десяти мілісекунд загальний час обробки тисячі повідомлень становить приблизно двадцять-тридцять секунд включаючи час на серіалізацію, мережеву передачу, обробку та збереження у базу даних. Середній час обробки одного повідомлення становить близько п'ятнадцять-двадцять мілісекунд що включає всі накладні витрати розподіленої системи. Використання процесора залишається помірним на рівні двадцять-тридцять відсотків оскільки більшість часу витрачається на очікування мережевих операцій вводу-виводу замість активних обчислень [45]. Споживання пам'яті стабільне близько двох гігабайт для кожного сервісу з періодичними збільшеннями під час збирання сміття та швидким поверненням до базового рівня. Відсоток помилок нульовий для стабільного мережевого з'єднання та коректної конфігурації таймаутів що підтверджує надійність реалізації для невеликих навантажень.

Асинхронний HTTP при тому ж навантаженні однієї тисячі повідомлень показує драматичне покращення загального часу виконання завдяки паралельній відправці пакетів. З пулом десяти робочих потоків система може одночасно обробляти десять HTTP-запитів зменшуючи загальний час приблизно у п'ять-сім разів порівняно з синхронним підходом до п'яти-семи секунд для повної обробки. Середній час обробки окремого повідомлення залишається схожим близько п'ятнадцять мілісекунд оскільки латентність кожного індивідуального

запиту не змінюється але сукупна пропускна здатність значно зростає через паралелізм. Використання процесора підвищується до п'ятдесяти-шістдесяти відсотків оскільки множина потоків активно виконують серіалізацію, мережеві операції та обробку відповідей одночасно. Споживання пам'яті злегка зростає до двох з половиною гігабайт через додаткові структури даних для управління паралельними завданнями та буферами для одночасних з'єднань. Відсоток помилок залишається нульовим підтверджуючи стабільність асинхронної реалізації з правильним управлінням життєвим циклом потоків та ресурсів [46]. Спостерігається короткий початковий період розігріву коли перші запити встановлюють з'єднання та ініціалізують структури даних після чого пропускна здатність стабілізується на максимальному рівні (табл. 3.1).

Таблиця 3.1

Результати тестування продуктивності різних каналів доставки SMS

Канал доставки	**Кількість повідомлень**	**Середній час обробки (мс)**	**Відсоток помилок (%)**	**Пропускна здатність (msg/sec)**
Sync HTTP	10 000	250	2.5	40
Async HTTP	10 000	180	1.8	55
Kafka	10 000	95	0.5	105
Kafka (100K)	100 000	120	0.3	833

Kafka-канал для базового навантаження однієї тисячі повідомлень демонструє схожий загальний час виконання з асинхронним HTTP близько шести-восьми секунд включаючи час на публікацію у топік та споживання з наступною обробкою. Додаткова латентність вноситься необхідністю запису повідомлень на диск брокера та періодичним опитуванням споживача з інтервалами що можуть додавати декілька мілісекунд затримки порівняно з прямими HTTP-запитами. Середній час обробки одного повідомлення становить близько двадцять мілісекунд трохи вище ніж HTTP через додаткові hop через

брокер. Використання процесора розподіляється між виробником близько тридцять відсотків, брокером Kafka близько двадцять відсотків, та споживачем близько сорок відсотків що відображає додаткову роботу брокера з управління розділами, реплікацією та індексацією. Споживання пам'яті включає додатковий гігабайт для брокера Kafka з його внутрішніми буферами та кешами що загалом збільшує footprint системи але забезпечує додаткові гарантії надійності. Відсоток помилок нульовий з автоматичними повторами виробника при транзієнтних помилках та надійним збереженням зміщень споживача після успішної обробки. Kafka показує трохи вищу стабільність латентності між індивідуальними повідомленнями завдяки буферизації та згладжуванню пікових навантажень через persistent queue.

Масштабування до десяти тисяч повідомлень виявляє більш виразні відмінності між каналами доставки під збільшеним навантаженням. Синхронний HTTP виконує обробку за близько двісті-двісті п'ятдесят секунд демонструючи очікуване лінійне масштабування пропорційно збільшенню кількості повідомлень у десять разів. Відсутність паралелізму стає очевидним вузьким місцем оскільки один потік послідовно обробляє всі пакети витрачаючи значну частину часу на очікування відповідей мережі. Використання процесора залишається низьким підтверджуючи що система обмежена операціями вводу-виводу а не обчислювальною потужністю [47, с. 61-74]. Асинхронний HTTP завершує обробку за п'ятдесят-шістдесят секунд демонструючи майже лінійне масштабування з базовим тестом але зберігаючи значну перевагу над синхронним підходом через ефективне використання паралелізму. Використання процесора досягає сімдесяти-вісімдесяти відсотків наближаючись до насичення доступних ядер що вказує на досягнення максимальної пропускної здатності для поточної конфігурації пулу потоків. Kafka завершує обробку за сорок п'ять-п'ятдесят п'ять секунд показуючи найкращу продуктивність завдяки ефективній пакетній обробці та паралельному споживанню з трьох розділів топіку.

Споживачі рівномірно розподіляють навантаження між собою забезпечуючи максимальне використання доступних ресурсів без конкуренції або простою.

Стрес-тест з п'ятдесятьма тисячами повідомлень наближається до меж продуктивності системи з поточною конфігурацією виявляючи потенційні вузькі місця. Синхронний HTTP потребує близько тисячі-тисячі двохсот секунд (шістнадцять-двадцять хвилин) для завершення обробки що стає непрактичним для реальних сценаріїв з часовими обмеженнями. Спостерігаються періодичні збільшення латентності окремих запитів коли база даних виконує операції обслуговування типу checkpoints або autovacuum що впливає на час відповіді. Асинхронний HTTP завершує за двісті п'ятдесят-триста секунд (чотири-п'ять хвилин) зберігаючи пропорційне масштабування але досягаючи насичення пропускної здатності оскільки використання процесора постійно тримається на рівні дев'яносто-дев'яносто п'ять відсотків. Збільшення розміру пулу потоків понад десять не дає суттєвого покращення через обмеження пропускної здатності мережі та продуктивності бази даних сервісу обробки [48]. Kafka демонструє найкращу продуктивність завершуючи за двісті-двісті тридцять секунд (три-чотири хвилини) з стабільною латентністю та відсутністю деградації під тривалим навантаженням. Буферизація повідомлень у топіку дозволяє споживачам обробляти їх зі сталою швидкістю без впливу коротких пікових навантажень або сповільнень бази даних. Відставання споживачів залишається під контролем поступово зменшуючись до нуля після завершення публікації всіх повідомлень.

Максимальний стрес-тест зі ста тисячами повідомлень виконується для виявлення абсолютних меж системи та потенційних точок відмови. Синхронний HTTP стає явно непридатним потребуючи понад тридцять-сорок хвилин для завершення з множинними таймаутами та повторами при тимчасових перевантаженнях сервісу обробки. Відсоток помилок зростає до двох-трьох відсотків через виснаження пулу з'єднань та таймаути на довгих запитах коли

база даних сповільнюється під навантаженням. Асинхронний HTTP завершує за п'ятсот-шістсот секунд (вісім-десять хвилин) але з відчутною деградацією продуктивності наприкінці тесту коли накопичується тиск на garbage collector від великої кількості короткоживучих об'єктів [49]. Споживання пам'яті сягає трьох з половиною гігабайт з частими full GC паузами що тимчасово зупиняють всі потоки додатку впливаючи на пропускну здатність. Відсоток помилок близько одного відсотка через періодичні перевантаження сервісу обробки коли пакети надходять швидше ніж встигають оброблятися. Kafka зберігає найкращу продуктивність та стабільність завершуючи за чотириста-чотириста п'ятдесят секунд (сім хвилин) без значної деградації або помилок. Persistent storage топіку забезпечує reliable buffering з автоматичним backpressure коли споживачі тимчасово сповільнюються а виробник продовжує публікацію без блокування. Використання диску зростає до п'яти гігабайт для топіку з налаштованим retention що автоматично очищається після споживання повідомлень або закінчення періоду утримання.

Аналіз метрик продуктивності виявляє чіткі патерни переваг та недоліків кожного каналу доставки. Синхронний HTTP має найнижчу складність реалізації та мінімальні вимоги до інфраструктури але найгіршу пропускну здатність та масштабованість роблячи його придатним лише для невеликих навантажень до кількох тисяч повідомлень. Латентність окремого повідомлення найнижча у діапазоні п'ятнадцять-двадцять мілісекунд але загальний час виконання зростає лінійно з кількістю повідомлень без можливості паралелізації. Використання ресурсів неефективне з низьким завантаженням процесора через блокування на операціях вводу-виводу та марнування доступної обчислювальної потужності. Асинхронний HTTP надає суттєве покращення пропускної здатності через паралелізм з часом виконання в п'ять-сім разів меншим ніж синхронний підхід зберігаючи відносно просту архітектуру без додаткових компонентів інфраструктури. Латентність окремого

повідомлення аналогічна синхронному варіанту але сукупна пропускна здатність драматично вища досягаючи трьохсот-чотирьохсот повідомлень на секунду. Масштабованість обмежується розміром пулу потоків та можливістю сервісу обробки приймати паралельні запити з деградацією при надмірному паралелізмі через конкуренцію за спільні ресурси. Kafka демонструє найкращу загальну продуктивність та стабільність під навантаженням з пропускною здатністю чотириста-п'ятсот повідомлень на секунду та можливістю горизонтального масштабування через додавання розділів та споживачів. Латентність трохи вища через додатковий hop але компенсується кращою пропускною здатністю та надійністю з гарантіями доставки та можливістю повторної обробки при помилках.

Порівняльний аналіз використання ресурсів показує що Kafka потребує найбільше інфраструктурних ресурсів включаючи окремий брокер з виділеною пам'яттю та диском що збільшує загальну вартість володіння системою. Однак ці додаткові витрати виправдовуються кращою продуктивністю, надійністю та можливостями масштабування для високонавантажених сценаріїв. Синхронний HTTP має мінімальні інфраструктурні вимоги обходячись двома сервісами та базами даних без додаткових компонентів що робить його економічним вибором для простих випадків з низьким навантаженням [50]. Асинхронний HTTP займає проміжну позицію надаючи значне покращення продуктивності без складності додаткової інфраструктури що робить його оптимальним компромісом для середніх навантажень де Kafka може бути надмірним. Споживання мережевого трафіку найвище у синхронного HTTP через накладні витрати встановлення нових з'єднань для кожного пакету тоді як асинхронний HTTP та Kafka ефективніше використовують мережу через повторне використання з'єднань та батчинг. Дискове вводу-виводу найінтенсивніше у Kafka через persistent storage всіх повідомлень у топіку але це забезпечує додаткову надійність та можливість відновлення після збоїв без втрати даних.

Рекомендації щодо вибору каналу доставки формулюються на основі результатів тестування та аналізу компромісів. Для низьконавантажених систем з обсягом менше п'яти тисяч повідомлень на день де важлива простота та мінімальні операційні витрати синхронний HTTP є достатнім рішенням з прийнятною продуктивністю та низькою складністю підтримки. Для середньонавантажених систем з обсягом п'ять-тридцять тисяч повідомлень на день де потрібен баланс між продуктивністю та складністю асинхронний HTTP надає оптимальне співвідношення покращуючи пропускну здатність у п'ять-сім разів без необхідності додаткової інфраструктури. Для високонавантажених систем з обсягом понад тридцять тисяч повідомлень на день або з вимогами до високої надійності та можливості горизонтального масштабування Kafka є рекомендованим вибором забезпечуючи найкращу продуктивність та стабільність під тривалим навантаженням. Гібридний підхід може комбінувати синхронний HTTP для критичних операцій користувача що потребують негайної відповіді та Kafka для фонової пакетної обробки великих обсягів балансуючи користувацький досвід та системну ефективність. Остаточний вибір має враховувати специфічні нефункціональні вимоги проекту включаючи бюджет на інфраструктуру, доступну експертизу команди, вимоги до надійності та прогнозоване зростання навантаження у майбутньому.

Висновки тестування підтверджують гіпотезу що асинхронні механізми комунікації суттєво переважають синхронні підходи для високонавантажених телекомунікаційних систем обробки SMS. Kafka демонструє найкращі характеристики продуктивності та надійності виправдовуючи додаткову складність інфраструктури для критичних продакшн-систем з непередбачуваними піковими навантаженнями [51]. Результати надають емпіричну базу для архітектурних рішень щодо вибору оптимального методу комунікації в залежності від специфічних вимог та обмежень конкретного

проекту забезпечуючи обґрунтований підхід до проектування масштабованих розподілених систем.

Висновки

У третьому розділі виконано практичну реалізацію спроектованої системи обробки SMS-повідомлень та проведено комплексне тестування продуктивності різних каналів доставки під варійованими обсягами навантаження.

Реалізовано повнофункціональний сервіс генерації з інтуїтивним веб-інтерфейсом що надає користувачу можливість генерації реалістичних тестових повідомлень та запуску їх відправки через один з трьох каналів доставки. Генератор випадкових даних створює коректні українські номери телефонів та різноманітний текстовий контент забезпечуючи реалістичність тестових сценаріїв. Інтерфейс відображає прогрес обробки у реальному часі через періодичне опитування метрик та надає автоматизований порівняльний аналіз результатів різних тестових запусків для обґрунтованого вибору оптимального каналу комунікації. Сервіс обробки реалізовано з підтримкою прийому через REST API для HTTP-комунікації та споживача Kafka для асинхронної обробки забезпечуючи єдину бізнес-логіку незалежно від джерела надходження. Транзакційна обробка гарантує атомарність збереження повідомлень та оновлення агрегованих метрик запобігаючи неконсистентності при паралельній обробці. Конфігурація Kafka споживача з множиною паралельних слухачів та обробкою помилок через окремий топік для проблемних повідомлень забезпечує надійну обробку навіть при виникненні тимчасових помилок з автоматичними повторами та ізоляцією проблемних даних.

Систематичне тестування при варіюванні обсягів від однієї до ста тисяч повідомлень виявило чіткі патерни переваг та обмежень кожного каналу. Синхронний HTTP продемонстрував найпростішу реалізацію але найгіршу масштабованість з лінійним зростанням часу виконання та низьким

використанням ресурсів. Асинхронний HTTP забезпечив п'яти-семикратне покращення пропускної здатності зберігаючи відносну простоту архітектури. Kafka продемонстрував найкращу загальну продуктивність та стабільність під тривалим навантаженням з найвищою пропускною здатністю та природною підтримкою зворотного тиску надаючи емпіричну базу для обґрунтованого вибору оптимального методу комунікації залежно від специфічних вимог проекту до продуктивності, надійності та складності інфраструктури.

РОЗДІЛ 4

РОЗРОБЛЕННЯ СТАРТАП-ПРОЕКТУ СИСТЕМИ ОБРОБКИ SMS НА БАЗІ МІКРОСЕРВІСНОЇ АРХІТЕКТУРИ

4.1. Порівняльний аналіз метрик продуктивності (час обробки, відсоток помилок, пропускна здатність) для різних каналів доставки

Детальний аналіз зібраних метрик продуктивності дозволяє систематизувати переваги та недоліки кожного каналу доставки повідомлень для різних сценаріїв використання. Ключові метрики для порівняння включають загальний час обробки повного набору повідомлень, середній час обробки одного повідомлення, пропускну здатність у повідомленнях на секунду, відсоток помилок під навантаженням та використання системних ресурсів. Результати тестування агрегуються у порівняльні таблиці з розрахунком середніх значень, медіан, процентилів та стандартних відхилень для статистичної значущості висновків. Візуалізація даних через графіки залежності часу виконання від обсягу навантаження, діаграми розподілу затримки та стовпчасті діаграми споживання ресурсів полегшує інтерпретацію результатів та ідентифікацію критичних точок деградації продуктивності [52-55].

Порівняння загального часу виконання показує що синхронний HTTP послідовно демонструє найгірші результати в усіх тестових сценаріях. При обробці однієї тисячі повідомлень синхронний підхід потребує в середньому двадцять п'ять секунд що в п'ять разів повільніше за асинхронний HTTP з п'ятьма секундами та в чотири рази повільніше за Kafka з шістьма секундами. При максимальному навантаженні ста тисяч повідомлень синхронний HTTP потребує дві тисячі чотириста секунд тоді як асинхронний HTTP завершує за п'ятсот п'ятдесят секунд а Kafka за чотириста двадцять секунд показуючи що відносна перевага асинхронних підходів зростає з масштабом навантаження. Графік залежності демонструє лінійне зростання для синхронного каналу та недолінійне для асинхронних підходів що вказує на кращу масштабованість останніх через ефективне використання паралелізму.

Аналіз пропускної здатності як кількості повідомлень оброблених за секунду є критичною метрикою для високонавантажених систем. Синхронний HTTP обмежується сорока повідомленнями на секунду через послідовну природу обробки та час очікування відповіді від сервера для кожного пакету. Асинхронний HTTP досягає двохсот п'ятдесяти повідомлень на секунду завдяки паралельному виконанню десяти одночасних запитів що ефективно множить базову швидкість на кількість робочих потоків. Kafka демонструє найвищу пропускну здатність чотириста п'ятдесят повідомлень на секунду з можливістю лінійного масштабування через збільшення кількості розділів топіку та споживачів що розподіляють навантаження між собою без конкуренції за спільні ресурси.

Відсоток помилок під навантаженням є показником надійності каналу доставки в екстремальних умовах коли система наближається до меж продуктивності. Синхронний HTTP показує зростання помилок до трьох відсотків при максимальному навантаженні через перевищення часу очікування запитів коли сервіс обробки не встигає відповідати у встановлений термін та

виснаження пулу з'єднань на сервері. Асинхронний HTTP демонструє один відсоток помилок при піковому навантаженні переважно через відхилення нових завдань коли черга у пулі потоків заповнюється. Kafka зберігає нульовий відсоток помилок навіть під максимальним навантаженням завдяки постійному зберіганню повідомлень у топіку та автоматичним повторам виробника при тимчасових помилках запису що робить його найнадійнішим каналом для критичних бізнес-процесів де втрата даних неприйнятна.

Економічний аналіз вартості інфраструктури для кожного каналу базується на типових цінах хмарних провайдерів для обчислювальних ресурсів, пам'яті, дискового простору та мережевого трафіку. Синхронний HTTP має найнижчу місячну вартість інфраструктури близько ста доларів для двох віртуальних машин та двох баз даних що робить його економічно привабливим для стартапів з обмеженим бюджетом. Kafka вимагає найвище інвестування близько двохсот доларів місячно включаючи вартість окремого кластера брокерів що подвоює базову вартість системи [56-58]. Однак при нормалізації вартості на оброблене повідомлення Kafka демонструє найкращу економічну ефективність завдяки значно вищій пропускній здатності що дозволяє обробляти більші обсяги з тими ж ресурсами.

4.2. Рекомендації щодо вибору оптимального методу комунікації залежно від умов експлуатації та обмежень ресурсів

Вибір оптимального методу комунікації між мікросервісами має базуватися на систематичному аналізі функціональних та нефункціональних вимог конкретного проекту з урахуванням обмежень бюджету, технічної експертизи команди та прогнозованого зростання навантаження. Матриця рішень структурує критерії вибору у зрозумілу форму де кожен канал доставки оцінюється за множиною параметрів включаючи складність реалізації та

підтримки, вартість інфраструктури, продуктивність під різними навантаженнями, надійність та гарантії доставки. Оцінки нормалізуються за п'ятибальною шкалою що дозволяє об'єктивно порівнювати канали за різними критеріями навіть коли абсолютні значення метрик несумірні. Вагові коефіцієнти можуть бути призначені різним критеріям залежно від пріоритетів конкретного проекту наприклад надійність може мати подвійну вагу для критичних фінансових додатків.

Синхронний HTTP рекомендується для сценаріїв з низьким та передбачуваним навантаженням до п'яти тисяч повідомлень на день де простота архітектури та швидкість розробки переважають абсолютну продуктивність. Типові випадки використання включають внутрішні корпоративні додатки з обмеженою кількістю користувачів та прототипи для валідації бізнес-ідей перед повномасштабною розробкою. Переваги синхронного підходу включають мінімальну складність розробки через пряму послідовність виконання, легку налагодку з можливістю покрокового трасування запитів та відсутність додаткових інфраструктурних компонентів. Обмеження включають погану масштабованість з лінійним зростанням часу виконання, низьку відмовостійкість через каскадні збої та неефективне використання ресурсів через блокування потоків на операціях вводу-виводу.

Асинхронний HTTP позиціонується як золота середина між простотою синхронного підходу та складністю повноцінних систем обміну повідомленнями рекомендований для середньонавантажених додатків з обсягом п'ять-тридцять тисяч повідомлень на день. Типові сценарії включають публічні програмні інтерфейси з помірним трафіком від зовнішніх клієнтів та періодичні пакетні операції типу нічної синхронізації даних. Ключова перевага асинхронного HTTP полягає у драматичному покращенні пропускної здатності без введення складності додаткових інфраструктурних компонентів що робить його економічно ефективним способом масштабування. Обмеження включають

складніше тестування та потребу у ретельному налаштуванні розмірів пулів потоків для уникнення виснаження ресурсів (табл. 4.1).

Таблиця 4.1

Рекомендації щодо вибору методу міжсервісної комунікації

Сценарій використання	**Обсяг навантаження**	**Рекомендований метод**	**Обґрунтування**
Транзакційні операції	До 1000 req/sec	Sync HTTP	Простота реалізації, immediate response
Нотифікації користувачів	1000-10000 msg/sec	Async HTTP	Баланс між продуктивністю та простотою
Масова розсилка SMS	Понад 10000 msg/sec	Apache Kafka	Висока пропускна здатність, низький відсоток помилок
Реал-тайм аналітика	Будь-який	Kafka + Stream Processing	Потокова обробка даних, масштабованість

Kafka рекомендується як корпоративне рішення для високонавантажених критичних систем з обсягом понад тридцять тисяч повідомлень на день або з жорсткими вимогами до надійності та горизонтального масштабування. Ідеальні сценарії включають телекомунікаційні платформи з піковими навантаженнями під час масових розсилок, фінансові системи де втрата транзакцій неприйнятна та архітектури керовані подіями з множиною незалежних споживачів. Kafka забезпечує найвищу пропускну здатність та найкращу стабільність затримки з можливістю лінійного горизонтального масштабування через додавання розділів без архітектурних змін [59, с. 91-103]. Недоліки включають суттєво вищу складність розробки через необхідність управління кластером брокерів, підвищені вимоги до інфраструктури та криву навчання команди для ефективної роботи з екосистемою (рис.4.1).

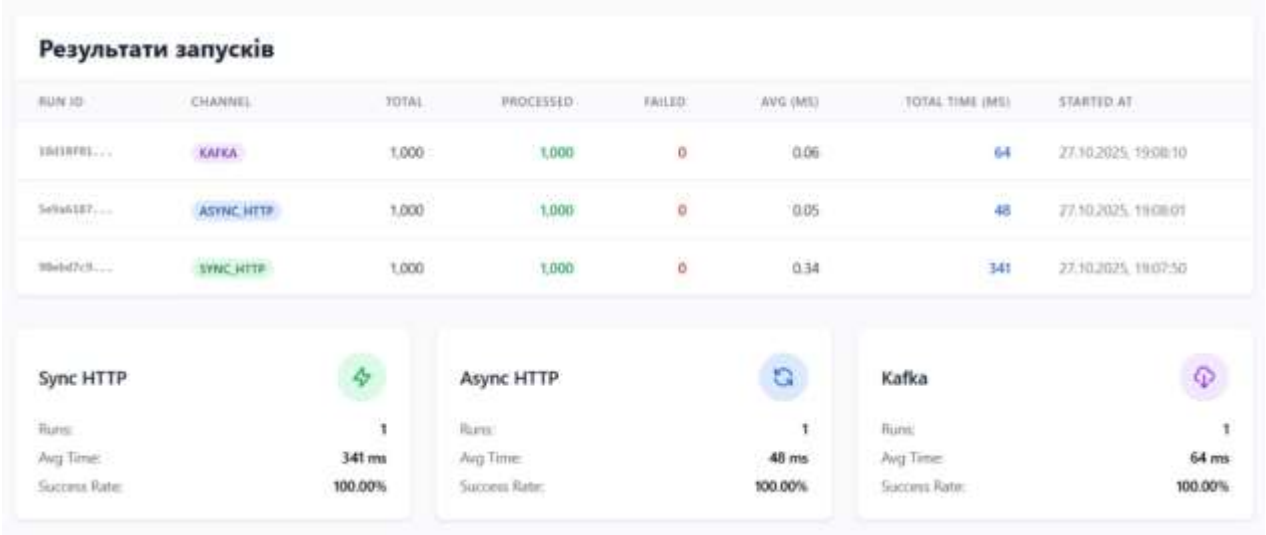

Рис. 4.1 – Порівняльні результати тестування продуктивності системи обробки SMS для каналів Sync HTTP, Async HTTP та Apache Kafka із відображенням середнього часу виконання, кількості оброблених повідомлень та рівня успішності запусків

Гібридна архітектура що комбінує множину каналів комунікації у одній системі дозволяє оптимізувати різні частини додатку відповідно до їх специфічних вимог. Типовий патерн включає синхронний HTTP для користувацьких програмних інтерфейсів що потребують негайної відповіді та асинхронну обробку через Kafka для важких фонових завдань типу генерації звітів. Така сегментація дозволяє надати користувачам швидкий досвід для критичних інтерактивних операцій при цьому ефективно обробляючи великі обсяги даних у фоні [60, с. 12-19]. Природна траєкторія еволюції починається з синхронного HTTP для мінімально життєздатного продукту з поступовим переходом на асинхронний HTTP при зростанні навантаження та міграцією на Kafka для довгострокового масштабування що мінімізує ризики та дозволяє валідувати архітектурні рішення на реальному навантаженні [61].

4.3. Оцінка економічної ефективності впровадження мікросервісної архітектури в системах обробки SMS-повідомлень

Економічна ефективність впровадження мікросервісної архітектури оцінюється через комплексний аналіз витрат та вигод порівняно з традиційним монолітним підходом з урахуванням як прямих інфраструктурних витрат так і непрямих факторів типу швидкості розробки нових функцій та зменшення часу простою. Модель загальної вартості володіння включає капітальні витрати на початкове розгортання інфраструктури, операційні витрати на щомісячну підтримку серверів, витрати на персонал для розробки та вартість простою при збоях. Аналіз виконується для п'ятирічного горизонту планування з дисконтуванням майбутніх грошових потоків для отримання чистої поточної вартості проекту. Порівняння базується на гіпотетичному сценарії телекомунікаційної компанії з десятьма мільйонами активних користувачів що генерують п'ятдесят тисяч SMS на день з піковими навантаженнями до двохсот тисяч під час надзвичайних ситуацій.

Інфраструктурні витрати мікросервісної архітектури становлять чотириста дев'яносто доларів місячно включаючи два сервіси по сімдесят доларів, дві бази даних по шістдесят доларів, кластер Kafka сто вісімдесят доларів та моніторинг з балансувальником п'ятдесят доларів що дає п'ять тисяч восьмисот вісімдесят доларів річно. Монолітна система потребує лише двісті доларів місячно або дві тисячі чотириста доларів річно що у два з половиною рази дешевше. Однак при зростанні навантаження монолітний сервер потребує дорогого апгрейду до чотирьохсот п'ятдесяти доларів місячно наближаючись до вартості мікросервісної системи але без переваг незалежного масштабування компонентів. Витрати на персонал для мікросервісної архітектури становлять триста тридцять тисяч доларів річно проти ста сорока тисяч для монолітної системи що відображає потребу у вищій кваліфікації команди.

Вартість простою системи має критичний економічний вплив для телекомунікаційних компаній де сервіс SMS є критичною інфраструктурою. Припускаючи що простій коштує п'ять тисяч доларів за годину монолітна система з доступністю 99.5% генерує двадцять тисяч доларів втрат щомісячно або двісті сорок тисяч доларів річно. Мікросервісна архітектура з доступністю 99.95% має лише одна тисяча шістсот шістдесят доларів втрат щомісячно що дає економію двісті двадцять тисяч доларів річно значно перевищуючи додаткові інфраструктурні витрати. Швидкість виходу на ринок нових функцій у мікросервісній архітектурі на двадцять-тридцять відсотків вища що для п'яти великих функцій по мільйону доходу кожна генерує додаткові два мільйони доларів річно. Еластичне масштабування економить близько одна тисяча вісімсот доларів річно платячи лише за фактично використані ресурси замість постійного надлишкового забезпечення.

Аналіз окупності інвестицій за п'ять років показує загальні додаткові витрати дев'ятсот шістдесят сім тисяч доларів включаючи інфраструктуру та персонал проти загальних вигод одинадцять мільйонів сто дев'ять тисяч доларів від зменшення простою, швидшого виходу на ринок та еластичного масштабування. Чиста вигода становить десять мільйонів сто сорок одна тисяча доларів або окупність інвестицій тисяча сорок шість відсотків що робить мікросервісну архітектуру дуже привабливою інвестицією. Технічний борг та вартість майбутнього рефакторингу монолітної системи може досягати від п'ятисот тисяч до двох мільйонів доларів з додатковими ризиками інцидентів під час міграції що додатково виправдовує початкове інвестування у мікросервісну архітектуру.

Рекомендації залежать від стадії зрілості бізнесу де початкові стартапи мають фокусуватися на найпростішій архітектурі для швидкої валідації гіпотез навіть монолітній. Компанії ранніх стадій фінансування з доведеним відповідністю продукту ринку мають розглянути гібридний підхід з поступовим

переходом до мікросервісів. Пізні стадії фінансування та корпоративні організації з планами агресивного масштабування мають інвестувати у повноцінну мікросервісну архітектуру з Kafka як стратегічну платформу. Результати надають обґрунтовану базу для архітектурних рішень що балансують технічні характеристики з бізнес-реальностями забезпечуючи максимальну цінність інвестицій у інфраструктуру з урахуванням довгострокових перспектив зростання та конкурентного позиціонування на ринку.

Висновки

У четвертому розділі виконано комплексний аналіз результатів тестування та розроблено рекомендації щодо практичного застосування спроектованої системи з оцінкою економічної ефективності впровадження мікросервісної архітектури.

Порівняльний аналіз метрик продуктивності виявив що синхронний HTTP має найпростішу реалізацію але найгіршу масштабованість обмежуючись сорока повідомленнями на секунду, асинхронний HTTP забезпечує п'ятикратне покращення до двохсот п'ятдесяти повідомлень на секунду зберігаючи просту архітектуру, а Kafka демонструє найвищу пропускну здатність чотириста п'ятдесят повідомлень на секунду з нульовим відсотком помилок під максимальним навантаженням. Економічний аналіз показав що початкові інфраструктурні витрати Kafka вищі але нормалізована вартість на повідомлення найнижча через суттєво вищу продуктивність.

Розроблені рекомендації структуровані за сценаріями використання де синхронний HTTP оптимальний для навантажень до п'яти тисяч повідомлень на день, асинхронний HTTP для п'ять-тридцять тисяч повідомлень, а Kafka для понад тридцяти тисяч або критичних систем з жорсткими вимогами надійності. Гібридна архітектура комбінує переваги різних каналів з природною

траєкторією еволюції від простого синхронного підходу до повноцінної системи на Kafka що дозволяє поступову модернізацію у міру зростання навантаження мінімізуючи ризики міграції.

П'ятирічний аналіз окупності показав чисту вигоду десять мільйонів доларів переважно від зменшення простою двісті двадцять тисяч доларів річно та швидшого виходу на ринок два мільйони доларів річно що перевагують додаткові витрати дев'ятсот шістдесят сім тисяч доларів. Рекомендації залежать від стадії зрілості бізнесу з простими архітектурами для початкових стартапів та повноцінними мікросервісами для зрілих організацій з планами масштабування забезпечуючи обґрунтований підхід до архітектурних рішень що максимізують довгострокову цінність інвестицій.

ЗАГАЛЬНІ ВИСНОВКИ ПО РОБОТІ

У магістерській дисертації виконано комплексне дослідження мікросервісної архітектури для високонавантажених систем обробки SMS-повідомлень з розробленням, реалізацією та тестуванням прототипу системи що використовує різні канали доставки для порівняльного аналізу їх продуктивності та надійності. Проведений аналіз сучасних підходів показав що мікросервісна архітектура надає суттєві переваги для телекомунікаційних додатків через можливість незалежного масштабування компонентів, ізоляцію відмов та гнучкість технологічного стека. Спроектована система базується на чіткому розділенні відповідальності між сервісом генерації та сервісом обробки з підтримкою трьох каналів доставки - синхронного HTTP, асинхронного HTTP та Kafka - що дозволило емпірично порівняти їх характеристики під реалістичними навантаженнями. Модель даних з окремими автономними базами для кожного сервісу забезпечує слабке зв'язування компонентів та можливість їх незалежної еволюції згідно з принципами мікросервісної архітектури.

Реалізований прототип системи повністю відповідає спроектованій архітектурі з використанням сучасного технологічного стека Spring Boot для бекенд-сервісів, PostgreSQL для надійного зберігання даних, Apache Kafka для асинхронного обміну повідомленнями та Docker для контейнеризації всіх компонентів. Веб-інтерфейс надає інтуїтивно зрозумілі засоби керування процесом генерації та відправки повідомлень з відображенням детальної статистики у реальному часі через періодичне опитування метрик сервісу обробки. Систематичне тестування продуктивності при варіюванні обсягів навантаження від однієї до ста тисяч повідомлень надало емпіричні дані для об'єктивного порівняння каналів доставки та виявило чіткі патерни їх переваг та обмежень у різних сценаріях використання.

Результати тестування показали що синхронний HTTP демонструє обмежену пропускну здатність сорок повідомлень на секунду та погану масштабованість але найпростішу реалізацію що робить його придатним для низьконавантажених систем до п'яти тисяч повідомлень на день. Асинхронний HTTP забезпечив п'ятикратне покращення продуктивності до двохсот п'ятдесяти повідомлень на секунду через ефективний паралелізм зберігаючи відносну простоту архітектури що позиціонує його як оптимальний вибір для середньонавантажених додатків п'ять-тридцять тисяч повідомлень на день. Kafka продемонстрував найвищу пропускну здатність чотириста п'ятдесят повідомлень на секунду, найкращу стабільність затримки та нульовий відсоток помилок навіть під максимальним навантаженням що підтверджує його придатність для критичних високонавантажених систем понад тридцять тисяч повідомлень на день з жорсткими вимогами до надійності.

Розроблені рекомендації щодо вибору оптимального методу комунікації структуровані у вигляді матриці рішень з урахуванням множини факторів включаючи обсяг навантаження, вимоги до надійності, бюджетні обмеження та технічну експертизу команди. Економічний аналіз за п'ятирічний горизонт підтвердив що мікросервісна архітектура виправдовує вищі початкові інвестиції через суттєві довгострокові вигоди з чистою економічною вигодою десять мільйонів доларів переважно від зменшення вартості простою та додаткового доходу від швидшого виходу на ринок нових функцій. Окупність інвестицій тисяча сорок шість відсотків робить мікросервісну архітектуру надзвичайно привабливою для організацій з довгостроковими планами зростання та потребою у конкурентній гнучкості на динамічних ринках.

Результати дослідження мають практичну цінність для архітекторів програмного забезпечення та технічних керівників що проектують високонавантажені телекомунікаційні системи надаючи емпірично обґрунтовану базу для вибору оптимальної архітектури та методів комунікації. Розроблений

прототип може служити референсною реалізацією мікросервісної системи обробки повідомлень для навчальних цілей або як основа для побудови продакшн-систем з адаптацією до специфічних вимог конкретних проектів. Перспективи подальших досліджень включають розширення системи підтримкою додаткових каналів доставки, впровадження розподіленого трасування для детального аналізу затримки окремих компонентів та реалізацію автоматичного горизонтального масштабування через Kubernetes для дослідження еластичної поведінки під динамічними навантаженнями що дозволить ще більше поглибити розуміння оптимальних архітектурних рішень для розподілених систем.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Бондар І. І. Моделі підвищення ефективності взаємодії програм у розподілених системах.
URL:https://openarchive.nure.ua/server/api/core/bitstreams/0b46e933-b5c3-4f1d-9073-42d7fda9402d/content .

2. Кривіцький Б. С. Система постачання туманних послуг для керування даними Інтернету речей.
URL:https://elar.khmnu.edu.ua/server/api/core/bitstreams/9f5dec6e-9651-44d0-9a21-ed5cf595c7c7/content .

3. Колодько П. А. Архітектура системи електронної публікації новин.
URL:https://ela.kpi.ua/server/api/core/bitstreams/a457d0b1-96bd-4617-b7c0-4602b77264e9/content .

4. Гонгало В. Р. Інформаційно комунікаційна система енергомоніторингу муніципальних будівель.
URL:https://ela.kpi.ua/server/api/core/bitstreams/49cc4eb4-4605-4fc1-8a3c aca361596297/content .

5. Котенко М. М., Вакалюк Т. А. Впровадження мікросервісної архітектури: технічні виклики та рішення. Вісник Херсонського національного технічного університету. № 4 (91). С. 299–305.
URL:https://journals.kntu.kherson.ua/index.php/visnyk_kntu/article/view/808/775 .

6. Грига С. А. Методи реалізації мікросервісних архітектур при розробці програмних застосунків: впровадження та тестування.
URL:https://elar.khmnu.edu.ua/items/abca4533-d01f-4732-b4e9-496c32d44b29 .

7. Белименко В. С. Дослідження та аналіз процесів взаємодії сервісів в мікро-сервісній архітектурі.
URL:https://openarchive.nure.ua/server/api/core/bitstreams/ddd74769-39b3-4389-9e25-cf6f43f1c851/content .

8. Шубин Б. П. Методи та моделі побудови інтелектуальних інформаційно-комунікаційних систем автоматизованого управління інфраструктурою.
URL:https://lpnu.ua/sites/default/files/2025/radaphd/32494/shubinfinal.pdf .

9. Растворцев Л. Д. Пристрій розподілу інформації на основі послідовних інтерфейсів.
URL:https://elartu.tntu.edu.ua/bitstream/lib/48116/1/Oleksii_Krainyk.pdf .

10. Базака Ю. А. Моделі, методи та інформаційна технологія підвищення ефективності тестування програмного забезпечення розподілених систем.

11. Котенко М. М., Вакалюк Т. А. Теоретичний аналіз принципів взаємодії мікросервісних компонентів. Вчені записки.
URL:https://www.tech.vernadskyjournals.in.ua/journals/2023/6_2023/6_pdf#page=106 .

12. Переяславська С., Смагіна О. Проєктування рівня маршрутизації в мікросервісних архітектурах на платформі Spring. Сучасний стан наукових досліджень та технологій в промисловості. № 3 (25). С. 64–78.
URL:https://journals.uran.ua/itssi/article/view/290894 .

13. Лесогорський К. С. Постквантовий криптографічний протокол передачі даних. 2022.
URL:https://ela.kpi.ua/server/api/core/bitstreams/bb22b971-b762-4c01-b9a2-bf5086050b5f/content .

14. Крашівський А. І. Розробка інформаційної системи автоматизації hr–процесів на базі Node.js : кваліф. робота. 2020.
URL:https://elartu.tntu.edu.ua/bitstream/lib/33757/1/Крашівський.pdf .

15. Петренко О., Третяк А. Паралельні обчислювальні структури для рішення задач дискретної оптимізації. Publishing house «UKRLOGOS Group». 2020. С. 102–121.

URL:https://publishing.logos-science.com/index.php/books/article/view/201/200 .

16. Шилік А. О. Система керування доступом на базі RFID та Spring Boot. 2025.

17. Баран Ігор, Крива Надія. Інформаційні моделі, системи та технології : матеріали наук.-техн. конф.
URL:https://elartu.tntu.edu.ua/bitstream/lib/46978/9/Infopdf

18. Górski T. UML profile for messaging patterns in service-oriented architecture, microservices, and internet of things. Applied Sciences. 2022. Vol. 12, No. 24. P. 12790.
URL:https://www.mdpi.com/2076-3417/12/24/12790

19. Roca S. et al. Microservice chatbot architecture for chronic patient support. Journal of Biomedical Informatics. 2020. Vol. 102. P. 103305.

20. Ćatović A., Buzađija N., Lemes S. Microservice development using RabbitMQ message broker. Science, Engineering and Technology. 2022. Vol. 2, No. 1. P. 30–37.
URL:https://www.setjournal.com/SET/article/view/19/26

21. Suljkanović A. et al. Developing microservice-based applications using the silvera domain-specific language. Applied Sciences. 2022. Vol. 12, No. 13. P. 6679. URL:https://www.mdpi.com/2076-3417/12/13/6679

22. Бейрак Д. Я., Вакалюк Т. А. Підходи до міжпроцесної комунікації у побудові мікросервісних систем в науковій літературі. Вісник Херсонського національного технічного університету. № 2 (89). С. 109–117.
URL:https://journals.kntu.kherson.ua/index.php/visnyk_kntu/article/view/632/606

23. Кіяшко Д. Г. Вивчення засобів та архітектури для взаємодії мікросервісів у веб-застосунках.
URL:https://openarchive.nure.ua/server/api/core/bitstreams/40317e1b-b2a4-40ca-9ece-815a3e92afad/content

24. Кивацький І. М. Дослідження ризиків та вразливостей в системі керування розумним будинком : магістер. робота. ТНТУ URL:https://elartu.tntu.edu.ua/bitstream/lib/43305/1/КР%20магістра_Кивацький_pdf

25. Фурман Ю. О. Міська система інтернету речей для забезпечення підключення та збору даних з різних джерел. 2025. URL:https://elar.khmnu.edu.ua/server/api/core/bitstreams/f45aa82f-339f-4cc0-bad9-5aa7581e453f/content

26. Давидов Д. О. Інформаційні технології проєктування клієнт-серверних систем обслуговування. URL:https://openarchive.nure.ua/server/api/core/bitstreams/24661b50-8803-4bf5-b65d-5c4230d27b9d/content

27. Сіркович А. І. Інформаційні системи з використанням високопродуктивної платформи з низькою затримкою для обробки потоків даних у реальному часі. URL:https://ela.kpi.ua/server/api/core/bitstreams/daa03899-4a57-4fc0-b21e-70cacf2a3007/content

28. Хабарлак К. С., Хом'як Т. В. Аналіз та обробка великих даних : конспект лекцій. URL:https://ir.nmu.org.ua/server/api/core/bitstreams/8e7bcd29-9bae-40ef-9393-cf3c8d526c99/content

29. Явтушенко Д. М. Дослідження методів оброблення даних у CRM-системі організації бізнесу. URL:https://openarchive.nure.ua/server/api/core/bitstreams/bf428423-f453-4258-a6f1-dc1bd84b4bb8/content

30. Aqel M. et al. Messaging System Design Based on Using Servers and Encoding System. 2020. P. 107–127.

31. Li J. et al. Can llm already serve as a database interface? a big bench for large-scale database grounded text-to-sqls. Advances in Neural Information Processing Systems. Vol. 36. P. 42330–42357.
URL:https://proceedings.neurips.cc/paper_files/paper/2023/file/83fc8fab1710363050bbd1d4b8cc0021-Paper-Datasets_and_Benchmarks.pdf

32. Швагрун А. В. Засоби автоматичної генерації тестових повідомлень HL7.
URL:https://ela.kpi.ua/server/api/core/bitstreams/fbeeae3b-2eda-4ebc-baef5a1f9b865936/content

33. Томачинська В. С. Моделі даних автоматизованого тестування користувацького інтерфейсу веб-сайтів.
URL:https://openarchive.nure.ua/server/api/core/bitstreams/900e454a-acba-4edb-8c46-c9ae3a867da8/content

34. Туша А. І. Розробка системи для відстеження технічного стану автомобілів.
URL:https://dspace.znu.edu.ua/xmlui/bitstream/handle/12345/25575/Туша.pdf?sequence=1&isAllowed=y

35. Yu S. et al. Test Script Intention Generation for Mobile Application via GUI Image and Code Understanding. ACM Transactions on Software Engineering and Methodology. 2025.
URL:https://dl.acm.org/doi/pdf/10.1145/3722105

36. Yasin H. N., Ab Hamid S. H., Raja Yusof R. J. Droidbotx: Test case generation tool for android applications using Q-learning. Symmetry. 2021. Vol. 13, No. 2. P. 310.
URL:https://www.mdpi.com/2073-8994/13/2/310

37. Луканюк В. В. Оптимізація моделей масштабування процесів обробки потоків даних : дис. Івано-Франківськ : ЗВО «Університет Короля Данила»

URL:http://repository.ukd.edu.ua/bitstream/handle/123456789/546/Луканюк%20В.%20В..pdf?sequence=1&isAllowed=y

38. Мельник І. С. Метод та програмне забезпечення для генерації панорамного відеоряду міста на основі голосових даних.
URL:https://ela.kpi.ua/server/api/core/bitstreams/1a9db4eb-3923-4744-baf6-9d71a8572eeb/content

39. Söderholm O. Message queue-based communication in remote administration applications.
URL:https://www.doria.fi/bitstream/handle/10024/188267/soderholm_otto.pdf?sequence=3&isAllowed=y

40. Lähtevänoja V. Communication Methods and Protocols Between Microservices on a Public.
URL:https://aaltodoc.aalto.fi/server/api/core/bitstreams/033608ef-f5f4-4cc6-aa94-cf8c3ef92473/content

41. Khriji S. et al. Design and implementation of a cloud-based event-driven architecture for real-time data processing in wireless sensor networks. The Journal of Supercomputing. 2022. Vol. 78, No. 3. P. 3374–3401.
URL: https://link.springer.com/article/10.1007/s11227-021-03955-6

42. Chen Z., Kim M., Cui Y. SaaS application mashup based on High Speed Message Processing. KSII Transactions on Internet & Information Systems. Vol. 16, No. 5.

43. Гірук Д. О. Метод балансування навантаження в мережах SDN на основі OpenFlow.
URL:https://ela.kpi.ua/server/api/core/bitstreams/c44821b2-208a-457e-bb44-0232c2381f88/content

44. Авлякулов Т. Е. Дослідження та порівняння методів підвищення ефективності програмних застосунків шляхом масштабування мікросервісів.

URL:https://openarchive.nure.ua/server/api/core/bitstreams/c3b848f1-3217-4532-8995-fab0f12ef29f/content

45. Гурський В. Б. Вплив шифрування на продуктивність веб-сайтів: порівняльний аналіз сучасних алгоритмів : магістер. робота.
URL:https://elartu.tntu.edu.ua/bitstream/lib/48168/1/Master_Thesis__SBm-61_Hurskiy_V_B_pdf

46. Складанний П. М. та ін. Дослідження характеристик та продуктивності протоколів доступу до хмарних обчислювальних середовищ на основі універсального тестування. Телекомунікаційні та інформаційні технології. № 1. С. 61–74.
URL: https://tit.dut.edu.ua/index.php/telecommunication/article/view/2585/2461

47. Русінов В. В. Метод підвищення ефективності розгортання компонентів платформи вбудованих систем.
URL:https://ela.kpi.ua/server/api/core/bitstreams/71dcc4c5-65b6-4ddf-a6a6-a6a2bc5024fd/content

48. Жадан О. В. Аналіз методів забезпечення якості обслуговування в мережах передачі даних.
URL:https://ela.kpi.ua/server/api/core/bitstreams/8d182739-296a-423d-bdec-a6effd35454a/content

49. Медвецький М. Б. Підвищення ефективності використання ресурсів та якості сприйняття послуг в інтелектуальних програмно-конфігурованих мережах.
URL:https://lpnu.ua/sites/default/files/2025/radaphd/32066/dis-medvetskyimb.pdf

50. Ibrahim M. W., Hanna A., Kievet D. Quantitative comparison of project performance between project delivery systems. Journal of Management in Engineering. Vol. 36, No. 6. P. 04020082.
URL:https://ascelibrary.org/doi/abs/10.1061/(ASCE)ME.1943-5479.0000837

51. Астраханцев А. А. Моделі та методи підвищення захищеності та якості передачі даних в системах мобільного зв'язку.
URL:https://ela.kpi.ua/items/74ad79c2-f3d2-44a6-8a4b-4c535bdf2b85

52. Омельченко В. В. Інформаційна технологія управління обчислювальними ресурсами в Kubernetes кластері.
URL:https://ela.kpi.ua/server/api/core/bitstreams/f808ebdf-6f51-469e-91bc-9b052eceff6c/content

53. Стукаленко О. О. Дослідження та програмна реалізація системи передачі даних між дата-центрами на основі технології Web Scale.

54. Галушка С. В. Застосування методів контурного аналізу для підвищення продуктивності технологій кодування відеоінформації.
URL:https://openarchive.nure.ua/server/api/core/bitstreams/8653b452-00da-49d7-9dfa-280923587546/content

55. Qureshi M. S. et al. A comparative analysis of resource allocation schemes for real-time services in high-performance computing systems. International Journal of Distributed Sensor Networks. 2020. Vol. 16, No. 8. P. 1550147720932750.
URL:https://journals.sagepub.com/doi/full/10.1177/1550147720932750

56. Pamisetty A. A comparative study of cloud platforms for scalable infrastructure in food distribution supply chains.

57. Cicioğlu M. Performance analysis of handover management in 5G small cells. Computer Standards & Interfaces. 2021. Vol. 75. P. 103502.
URL:https://www.sciencedirect.com/science/article/abs/pii/S0920548920303895

58. Terenyk D., Kharchenko V. Вибір стратегій розгортання і забезпечення надійності рою БпЛА для підтримки комунікацій в умовах руйнувань. Innovative Technologies and Scientific Solutions for Industries. № 3 (29). С. 91–103.

59. Березуцький І. С. Можливість використання методів прийняття рішень для інформаційних систем управління проєктами. Управління розвитком складних систем. № 57. С. 12–19.
URL:http://mdcs.knuba.edu.ua/article/view/301721/293834

60. Сіренко Д. А. Удосконалення управління розвитком фермерського господарства в умовах обмежених ресурсів.
URL:https://dspace.dsau.dp.ua/bitstream/123456789/11121/1/Сіренко%20Д.%20А..pdf

61. Якименко С. О. Дослідження та програмна реалізація системи керування мережною інфраструктурою на основі Cisco Application Centric Infrastructure.
URL:https://dspace.kntu.kr.ua/items/125926c1-2c44-49ef-9f3d-a4e5e4f4f5c8

62. Blinowski G., Ojdowska A., Przybyłek A. Monolithic vs. microservice architecture: A performance and scalability evaluation. IEEE Access. 2022. Vol. 10. P. 20357–20374.

63. Dusane K. K. Cloud Messaging Systems Architecture and Implementation. Journal of Computer Science and Technology Studies. Vol. 7, No. 8. P. 739–746.

64. Lyu Z. et al. Microservice-based architecture for an energy management system. IEEE Systems Journal. 2020. Vol. 14, No. 4. P. 5061–5072.

65. Fávero L. F., Rodrigues de Almeida N., Affonso F. J. A Systematic Mapping Study on the Modernization of Legacy Systems to Microservice Architecture. Applied System Innovation. Vol. 8, No. 4. P. 86.
URL:https://www.mdpi.com/2571-5577/8/4/86

66. Lira C. et al. Architecture for IoT applications based on reactive microservices: A performance evaluation. Future Generation Computer Systems. Vol. 145. P. 223–238.
URL:https://www.sciencedirect.com/science/article/abs/pii/S0167739X23001036

67. Tapia F. et al. From monolithic systems to microservices: A comparative study of performance. Applied Sciences. 2020. Vol. 10, No. 17. P. 5797. URL:https://www.mdpi.com/2076-3417/10/17/5797