Міністерство освіти і науки України
Національний університет «Києво-Могилянська академія»
Факультет інформатики
Кафедра мультимедійних систем

Кваліфікаційна робота

освітній ступінь — бакалавр

на тему:

«СИСТЕМА КООРДИНАЦІЇ ТРАНЗАКЦІЙ МІЖ МІКРОСЕРВІСАМИ НА ОСНОВІ ПАТЕРНУ SAGA»

Виконав: студент 4-го року навчання,
спеціальності 121
«Інженерія програмного забезпечення»
Шандрик Андрій

Керівник Бабич Т. А.,
старший викладач

Рецензент _____
(прізвище та ініціали)

Кваліфікаційна робота захищена
з оцінкою _____

Секретар ЕК _____
«___» _____ 20___ р.

Київ — 2026

Національний університет «Києво-Могилянська академія»

Факультет інформатики
Кафедра мультимедійних систем
Освітній ступінь бакалавр
Спеціальність 121 «Інженерія програмного забезпечення»
Освітня/освітньо-наукова програма бакалавр

ЗАТВЕРДЖУЮ
Зав. кафедри мультимедійних систем,
доц., канд. наук
Жежерун О. П.

_____
(підпис)
«13» жовтня 2025 р.

З А В Д А Н Н Я
ДЛЯ КВАЛІФІКАЦІЙНОЇ РОБОТИ СТУДЕНТУ

Шандрику Андрію
(прізвище, ім'я, по батькові)

1. Тема роботи **«Система координації транзакцій між мікросервісами на основі патерну Saga»**, керівник роботи Бабич Трохим Анатолійович, аспірант комп'ютерних наук, старший викладач.

(прізвище, ім'я, по батькові, науковий ступінь, вчене звання)

затверджені наказом вищого навчального закладу від «___» _____ 20___ року № ___

2. Строк подання студентом роботи 15 травня 2026

3. План роботи:

Анотація
Вступ
Розділ 1. Дослідження та аналіз предметної області
1.1 Архітектура мікросервісних систем
1.2 Проблеми узгодженості даних у розподілених середовищах
1.3 Підходи до керування транзакціями в мікросервісах
1.4 Патерн Saga: концепція, типи (оркестрація та хореографія), переваги й обмеження
Розділ 2. Проєктування та розробка системи

ГРАФІК ПІДГОТОВКИ КВАЛІФІКАЦІЙНОЇ РОБОТИ ДО ЗАХИСТУ

№ з/п	ПЕРЕЛІК РОБІТ	Термін виконання	Дата ознайомлення наукового керівника	Підпис наукового керівника	Примітки
1.	Вибір теми, затвердження її на засіданні кафедри та закріплення наукового керівника. Узгодження календарного графіка підготовки кваліфікаційної роботи. Ознайомлення студента з критеріями оцінювання кваліфікаційної роботи (п. 8.5).	13 жовтня 2025			
2.	Вивчення джерел літератури, матеріалів архівів, періодичних видань, збір та узагальнення фактів, даних	13 жовтня 2025 – 3 листопада 2025			
3.	Складання плану кваліфікаційної роботи та узгодження з науковим керівником	3 листопада 2025			
4.	Написання розділів роботи	листопад 2025 – березень 2026			
5.	Проміжний контроль виконання роботи	01 лютого 2026			
6.	Написання кваліфікаційної роботи в цілому, ознайомлення з її першим варіантом наукового керівника	12 січня 2026 – 29 березня 2026			
	Розділ 1 (постановка проблеми, теоретичні основи, огляд літературних джерел)	25 січня 2026			
	Розділ 2 (аналітично-дослідницька частина)	01 березня 2026			
	Розділ 3 (проєктно-рекомендаційна частина)	29 березня 2026			
7.	Попередній захист кваліфікаційної роботи на засіданні кафедри	30 березня – 1 квітня 2026			
8.	Повне завершення написання кваліфікаційної роботи, оформлення її згідно з вимогами й подання на відгук науковому керівнику	01 квітня – 15 травня 2026			
9.	Подання кваліфікаційної роботи для перевірки на відповідність вимогам академічної доброчесності	До 15 травня 2026			

№ з/п	ПЕРЕЛІК РОБІТ	Термін виконання	Дата ознайомлення наукового керівника	Підпис наукового керівника	Примітки
10.	Публічний захист кваліфікаційної роботи перед екзаменаційною комісією	згідно з розкладом роботи ЕК			

Графік узгоджено «13» жовтня 2025 р.

Науковий керівник _____ Бабич Т. А.

Виконавець кваліфікаційної роботи _____ Шандрик А.

ЗМІСТ

АНОТАЦІЯ

Метою даної роботи є дослідження існуючих підходів до керування розподіленими транзакціями та розробка системи координації транзакцій між мікросервісами на основі патерну оркестрації Saga засобами платформи .NET та мови C#. У роботі розглянуто теоретичні відомості про мікросервісну архітектуру, проблеми узгодженості даних у розподілених середовищах та проведено порівняльний аналіз підходів оркестрації та хореографії. Практичною частиною роботи є проєктування та реалізація власної системи, що підтримує підключення кроків саги через механізм впровадження залежностей (DI), ідемпотентність операцій, кореляцію подій, політики повторів із таймаутами та автоматичне виконання компенсувальних дій. Кінцевим етапом є інтеграційне тестування системи, моделювання збійних сценаріїв та аналіз продуктивності й масштабованості розробленого рішення.

Ключові слова: мікросервіси, розподілені транзакції, Saga, оркестрація, хореографія, ідемпотентність, компенсувальні дії, .NET, C#, EF Core.

ПЕРЕЛІК УМОВНИХ ПОЗНАЧЕНЬ І СКОРОЧЕНЬ

ACID — модель властивостей транзакцій бази даних (Atomicity, Consistency, Isolation, Durability)

AMQP — Advanced Message Queuing Protocol — протокол прикладного рівня обміну повідомленнями

API — Application Programming Interface — інтерфейс прикладного програмування

BASE — модель послаблених гарантій узгодженості (Basically Available, Soft state, Eventually consistent)

CAP — теорема про несумісність трьох властивостей розподіленого сховища (Consistency, Availability, Partition tolerance)

CDC — Change Data Capture — захоплення змін із журналу транзакцій бази даних

CI/CD — Continuous Integration / Continuous Deployment — безперервна інтеграція та доставка

CLR — Common Language Runtime — середовище виконання .NET

CQRS — Command Query Responsibility Segregation — розділення обов’язків команд і запитів

DDD — Domain-Driven Design — предметно-орієнтоване проєктування

DI — Dependency Injection — впровадження залежностей

DSL — Domain-Specific Language — доменно-специфічна мова

DTC — Distributed Transaction Coordinator — координатор розподілених транзакцій (Microsoft)

E2E — End-to-End — наскрізне тестування на повному стеку компонентів

EF Core — Entity Framework Core — об’єктно-реляційний відображувач для платформи .NET

gRPC — фреймворк віддаленого виклику процедур від Google на основі HTTP/2 і Protocol Buffers

HTTP — HyperText Transfer Protocol — протокол передачі гіпертексту

IaC — Infrastructure as Code — інфраструктура як код

JIT — Just-In-Time compilation — компіляція під час виконання

JSON — JavaScript Object Notation — текстовий формат обміну даними

MVCC — Multi-Version Concurrency Control — багатоверсійний контроль конкурентного доступу

mTLS — mutual Transport Layer Security — взаємна автентифікація сторін засобами TLS

NFR — Non-Functional Requirement — нефункціональна вимога

ORM — Object-Relational Mapping — об’єктно-реляційне відображення

PACELC — розширення CAP-теореми, що враховує компроміс між затримкою і узгодженістю за відсутності розділення

REST — Representational State Transfer — архітектурний стиль вебсервісів

SDK — Software Development Kit — комплект розробника програмного забезпечення

SHA — Secure Hash Algorithm — алгоритм криптографічного хешування

SOA — Service-Oriented Architecture — сервіс-орієнтована архітектура

SQL — Structured Query Language — структурована мова запитів

TCC — Try-Confirm-Cancel — протокол керування розподіленими транзакціями

TLS — Transport Layer Security — протокол захисту транспортного рівня

UUID — Universally Unique Identifier — універсальний унікальний ідентифікатор

WAL — Write-Ahead Log — журнал випереджувального запису СКБД

XA — eXtended Architecture — стандарт двофазного фіксування транзакцій (X/Open)

2PC — Two-Phase Commit — протокол двофазної фіксації

3PC — Three-Phase Commit — протокол трифазної фіксації

СКБД — система керування базами даних

ВСТУП

Поширення мікросервісної архітектури поставило перед розробниками конкретну проблему: як забезпечити узгодженість даних, коли кілька сервісів зі своїми базами даних мають виконати пов’язані операції. У монолітному застосунку це вирішується на рівні однієї транзакції бази даних. У мікросервісному середовищі такий підхід не працює.

Актуальність теми зумовлена тим, що традиційні розподілені транзакції на основі двофазної фіксації погано поєднуються з мікросервісами: вони вимагають тривалого блокування ресурсів і знижують доступність системи. Патерн Saga вирішує це інакше — довга операція розбивається на локальні транзакції, кожна з яких у разі збою може бути відкочена через компенсувальну дію. Концепція відома, але її повноцінна реалізація вимагає вирішення низки технічних питань: ідемпотентності обробників, кореляції подій між сервісами, управління повторами та таймаутами.

Мета роботи — проєктування та реалізація системи координації розподілених транзакцій між мікросервісами на основі патерну оркестрації Saga із підтримкою компенсувальних дій, ідемпотентності та керованих політик повторів.

Для досягнення поставленої мети в роботі необхідно виконати такі завдання:

- розглянути проблему узгодженості даних у мікросервісних архітектурах та обмеження класичних підходів до розподілених транзакцій;
- проаналізувати патерн Saga та порівняти підходи хореографії й оркестрації;
- дослідити наявні бібліотеки для реалізації Saga в екосистемі .NET;
- спроєктувати архітектуру системи, що охоплює ідемпотентність, кореляцію подій, політики повторів і таймаути;

- реалізувати систему у вигляді набору пакетів SagaFlow із підтримкою різних транспортних і сховищних адаптерів;
- перевірити коректність роботи системи та узагальнити результати.

Об’єктом дослідження є розподілені транзакції в мікросервісних системах.

Предметом дослідження є методи та засоби їх координації на основі патерну Saga у платформі .NET.

У роботі використано методи аналізу та порівняння наявних рішень, проєктування програмного забезпечення і тестування. Практична частина реалізована мовою C# із застосуванням атрибутного підключення оброблювачів через механізм впровадження залежностей, бібліотеки Polly для керування повторами та EF Core для зберігання стану саг і outbox-таблиці.

Практичне значення роботи полягає в тому, що розроблена система може застосовуватись як готова бібліотека для .NET-проєктів, де потрібна координація операцій між сервісами, а також у навчальному процесі — при вивченні архітектурних патернів і підходів до розробки мікросервісних систем.

Структурно робота складається зі вступу, трьох розділів, висновків та списку використаних джерел. У першому розділі розглянуто теоретичні засади розподілених транзакцій і патерну Saga. Другий розділ присвячено проєктуванню системи SagaFlow — її архітектурі, вибору технологій та механізмів керування помилками. У третьому розділі описано реалізацію системи, результати інтеграційного тестування та аналіз продуктивності.

РОЗДІЛ 1. ДОСЛІДЖЕННЯ ТА АНАЛІЗ ПРЕДМЕТНОЇ ОБЛАСТІ

1.1 Архітектура мікросервісних систем

Мікросервісна архітектура — це архітектурний стиль побудови програмних систем, за якого застосунок формується як сукупність невеликих, автономних і слабо пов’язаних сервісів, що взаємодіють між собою через полегшені мережеві протоколи. Кожен такий сервіс реалізує певну бізнес-спроможність (англ. *business capability*), має власну кодову базу, розгортається самостійно і, як правило, володіє окремим сховищем даних.

Єдиного загальновизнаного визначення мікросервісів не існує. Формулювання М. Фаулера, С. Ньюмана, К. Річардсона, Е. Вольфа, М. Річардса та Н. Форда [1–5] відрізняються акцентами, однак усі зводяться до спільного набору ознак: модульність, побудова навколо бізнес-функції, слабка зв’язаність, незалежне розгортання, децентралізоване управління даними, технологічна гнучкість. Систематизовану добірку таких ознак і відповідних інженерних патернів подає каталог К. Річардсона [6].

В основі підходу лежить організаційний принцип: одна невелика команда відповідає за сервіс упродовж усього життєвого циклу — від проєктування до експлуатації в бойовому середовищі. Цю модель часто називають «ти написав — ти підтримуєш» (*you build it, you run it*). Вона скорочує час доставки змін і дає змогу масштабувати організаційну структуру паралельно з кодовою базою, але водночас вимагає від команд значно ширшого набору компетенцій, ніж у класичному поділі «розробка — експлуатація».

1.1.1 Історичні передумови виникнення

Ідея побудови застосунків із набору малих сервісів не є принципово новою. Її коріння сягають сервіс-орієнтованої архітектури (SOA), філософії Unix (ланцюжки простих інструментів, кожен з яких «робить одну річ добре») та гексагональної архітектури А. Кокберна, описаної у 2005 році [7].

Термін «мікросервіс» у сучасному розумінні почав поширюватися приблизно з 2011–2014 років. На його популяризацію вплинули публікації Дж. Льюїса та М. Фаулера [1], а також практичний досвід компаній Netflix, Amazon та eBay, які перейшли від монолітних застосунків до сервіс-орієнтованих архітектур із сотнями сервісів. Вирішальним технологічним фактором стала поява Docker (2013) та Kubernetes (2014): без контейнеризації та оркестрації керування десятками сервісів у продакшені лишалося практично нездійсненним.

Перехід від моноліту до мікросервісів у літературі часто описується як еволюційний, а не одномоментний процес. Він зазвичай є реакцією на конкретні проблеми: надмірний розмір кодової бази, ускладнену координацію команд, неможливість масштабувати окремі функції незалежно, повільний цикл випуску змін.

1.1.2 Основні принципи та характеристики

До фундаментальних принципів мікросервісної архітектури належать такі.

1. *Побудова навколо бізнес-функцій.* Сервіси проєктуються відповідно до обмежених контекстів (*bounded contexts*) предметної області — концепції, запропонованої Е. Евансом у рамках предметно-орієнтованого проєктування (DDD) [8]. Кожен сервіс інкапсулює певну частину бізнес-логіки разом з її моделлю даних.
2. *Слабка зв'язаність.* Сервіси мають мінімальну кількість спільних точок: не поділяють кодову базу, не використовують спільну базу даних, не залежать від внутрішньої реалізації одне одного. Взаємодія здійснюється виключно через чітко визначені API.
3. *Незалежне розгортання.* Кожен сервіс розгортається окремо від інших, що дає змогу вносити зміни в одну частину системи без потреби переставляти весь застосунок і знижує ризики під час випуску оновлень.
4. *Децентралізоване управління даними.* На відміну від монолітів зі спільною базою даних, кожен мікросервіс володіє власним сховищем.

Підхід «database per service» дає змогу обрати найбільш придатну для конкретних задач технологію зберігання — реляційну, документну, графову чи ключ-значення — і отримав назву поліглотної персистентності.

5. *Автоматизація інфраструктури.* Велика кількість сервісів унеможливлює ручне керування. CI/CD, інфраструктура як код (IaC), автомасштабування та оркестрація контейнерів стають обов'язковою умовою, а не приємним доповненням.
6. *Проєктування з урахуванням відмов.* Розподілена система за визначенням ненадійна: мережа не завжди доступна, вузли виходять з ладу, сервіси можуть бути тимчасово недоступні. Тому широко застосовуються патерни стійкості — розмикач кола (*Circuit Breaker*), повторні спроби з експоненційною затримкою, таймаути, перегородки (*Bulkhead*), резервні відповіді (*Fallback*).
7. *Спостережуваність.* У системі, де один бізнес-сценарій виконується через ланцюжок з десятка сервісів, без спеціальних засобів просто неможливо зрозуміти, що саме відбувається під час обробки запиту. Цю функцію виконують централізоване логування, розподілене трасування (OpenTelemetry, Jaeger, Zipkin), збір метрик (Prometheus) та засоби візуалізації (Grafana).

1.1.3 Компоненти мікросервісної архітектури

Типова мікросервісна система, окрім самих сервісів, містить низку допоміжних компонентів, без яких її повноцінна експлуатація неможлива.

API-шлюз (API Gateway) виступає єдиною точкою входу для зовнішніх клієнтів. Замість того щоб звертатися безпосередньо до десятків внутрішніх сервісів, клієнт взаємодіє зі шлюзом, який виконує маршрутизацію запитів, агрегацію відповідей, автентифікацію, обмеження швидкості, термінацію TLS, логування та інші крізні функції (*cross-cutting concerns*). Шлюз ізолює клієнтів від деталей внутрішньої організації сервісів і спрощує їх подальшу еволюцію.

Водночас потрібно стежити, щоб у шлюзі не накопичувалася бізнес-логіка, інакше він перетворюється на прихований моноліт.

Реєстр сервісів і механізми виявлення (service discovery) потрібні, щоб сервіси могли знаходити одне одного в динамічному середовищі. Адреси й порти екземплярів постійно змінюються під час автомасштабування, перезапусків, розгортання нових версій, тому статична конфігурація не підходить. Виявлення здійснюється або на клієнтській стороні (*Client-side Discovery*), або на серверній — через балансувальник навантаження.

Брокери повідомлень і системи подій (RabbitMQ, Apache Kafka, Azure Service Bus, NATS) забезпечують асинхронну взаємодію. Вони лежать в основі подієво-керованих архітектур, у яких сервіси реагують на події, опубліковані іншими компонентами, замість того щоб викликати одне одного напряму. Ця інфраструктура також є фундаментом для патернів координації розподілених операцій, розгляду яких присвячено подальші підрозділи роботи.

Платформа оркестрації контейнерів (Kubernetes, Docker Swarm, Azure Container Apps) відповідає за розгортання сервісів, їх планування на вузлах кластера, моніторинг стану, автоматичне відновлення після відмов та масштабування за навантаженням.

Засоби спостережуваності об’єднують три категорії телеметрії: логи, метрики та трейси. Агент або SDK у кожному сервісі надсилає ці дані до централізованого сховища, де вони корелюються за ідентифікатором запиту (*correlation ID*). Без коректної кореляції діагностика розподілених збоїв стає практично неможливою.

Сервісна сітка (service mesh) — інфраструктурний рівень, що виносить функції міжсервісного зв’язку (mTLS, повтори, балансування, спостережуваність) зі застосункового коду в спеціальні проксі-агенти (*sidecars*), встановлені поруч із кожним екземпляром сервісу. Поширеними реалізаціями є Istio, Linkerd, Consul Connect.

1.1.4 Моделі взаємодії між сервісами

Спосіб, у який сервіси обмінюються даними, значною мірою визначає властивості системи: її продуктивність, стійкість, складність відлагодження. Прийнято виділяти дві основні моделі.

Синхронна взаємодія типу «запит-відповідь» реалізується через HTTP/REST, gRPC або GraphQL. Сервіс-ініціатор надсилає запит і блокується до отримання відповіді. Модель проста в розумінні, однак утворює часове сполучення: якщо сервіс B недоступний, сервіс A, що його викликає, також зазнає проблем. Довгі ланцюжки синхронних викликів підвищують латентність і помножують ризики: відмова будь-якої ланки ланцюжка означає відмову всієї операції.

Асинхронна взаємодія через обмін повідомленнями здійснюється через брокера. Вона буває двох типів: команда — адресне повідомлення конкретному отримувачу; подія — повідомлення за моделлю публікація/підписка, без знання про конкретних одержувачів. Асинхронна модель кращою мірою переносить тимчасову недоступність окремих сервісів, природно підтримує кількох споживачів однієї події та простіше масштабується. Її ціна — складніше відлагодження, додаткова інфраструктура (брокер зі своїми режимами відмов) і необхідність окремо забезпечувати ідемпотентність оброблювачів.

На практиці мікросервісні системи зазвичай поєднують обидва підходи: синхронний використовується для операцій, що потребують негайної відповіді (наприклад, читання даних для інтерфейсу), асинхронний — для розподілених бізнес-процесів, фонових задач та інтеграцій.

1.1.5 Управління даними та проблема узгодженості

Принцип «база даних на сервіс» є однією з центральних технічних вимог мікросервісної архітектури та водночас джерелом її найсерйозніших викликів.

У моноліті узгодженість операції, що змінює кілька сутностей, забезпечується транзакціями бази даних із властивостями ACID: атомарність (Atomicity), узгодженість (Consistency), ізольованість (Isolation), стійкість

(Durability). У мікросервісній системі еквівалентна операція може охоплювати кілька сервісів з різними сховищами. Застосування традиційних розподілених транзакцій (двофазна фіксація, XA) для її координації у документації Microsoft Azure Architecture Center [9], роботах К. Річардсона [3] та інших джерелах кваліфікується як анти-патерн: такий підхід створює сильне часове сполучення між усіма учасниками, знижує доступність і погано масштабується.

Замість нього в мікросервісних архітектурах поширений підхід поступової узгодженості (*eventual consistency*), що відповідає моделі BASE (*Basically Available, Soft state, Eventually consistent*). Система при цьому може тимчасово перебувати в неузгодженому стані, але гарантує, що за відсутності нових змін зрештою його досягне. Реалізація розподілених бізнес-операцій у таких умовах спирається на ідемпотентні оброблювачі, компенсувальні операції та координацію через обмін повідомленнями.

1.1.6 Переваги та недоліки

Переваги мікросервісного підходу — незалежне масштабування окремих сервісів, ізоляція відмов, технологічна гнучкість, паралельна робота кількох команд, спрощення окремо взятого сервісу, можливість поступової модернізації успадкованих систем — добре відомі й багатократно описані в літературі.

Значно важливіше для даної роботи розуміти вартість цих переваг. Мікросервісна архітектура не усуває складність, а переносить її з рівня коду на рівень розподіленої системи. Загальна складність при цьому, як правило, зростає: окремі сервіси простіші, але їх взаємодія породжує класичні проблеми розподілених систем — часткові відмови, ненадійність мережі, відсутність глобального стану, складність діагностики. Особливо вразливою точкою є цілісність даних: будь-яка бізнес-операція, що зачіпає два або більше сервіси, з технічного погляду стає розподіленою транзакцією з усіма її властивостями.

До інших типових недоліків належать збільшена мережева латентність, складніше тестування (повноцінна перевірка сценарію вимагає підняття кількох сервісів одночасно), ускладнене версіонування API, підвищені вимоги до

зрілості DevOps-процесів. М. Фаулер і С. Ньюман наголошують [1; 2], що мікросервіси слід розглядати не як «за замовчуванням кращий» вибір, а як компроміс. Для невеликих застосунків і команд моноліт нерідко лишається раціональнішим рішенням.

1.2 Проблеми узгодженості даних у розподілених середовищах

Задача узгодженості даних у монолітному застосунку здебільшого зводиться до правильного використання транзакцій бази даних: одна база, один процес, одна відмова — один відкат. У мікросервісному середовищі ці припущення перестають виконуватися, і проблема перетворюється на самостійний клас задач, пов’язаний із фундаментальними обмеженнями розподілених систем.

1.2.1 Джерела неузгодженості в розподілених системах

Узгодженість у моноліті гарантована двома умовами за замовчуванням. Перша: усі зміни йдуть через одне сховище з вбудованими механізмами ізоляції та атомарності. Друга: код виконується в одному процесі, і якщо компонент відмовив, операція припиняється цілком — часткових відмов не буває за визначенням.

У розподіленій системі жодне з цих припущень не виконується. У класичних роботах з розподілених обчислень (Л. Лампорт [10], М. Клеппман [11]) виділено кілька типових джерел неузгодженості, від яких у тій чи іншій формі страждає кожна мікросервісна архітектура.

Ненадійність мережі. Мережеве з’єднання може бути втраченим, повідомлення — затриматися, загубитися або прийти не в порядку відправлення. Важливим наслідком є асиметрія спостережень з погляду відправника: ситуації «запит не дійшов» і «запит дійшов, але відповідь втрачено» принципово не розрізняються. Це означає, що клієнт у загальному випадку не знає фактичного стану операції на віддаленому сервісі, і саме звідси виникає потреба в ідемпотентних оброблювачах та явних механізмах повторного надсилання.

Часткові відмови. На відміну від моноліту, де збій одного компонента зупиняє весь процес, у розподіленій системі частина вузлів може працювати коректно, а інша — бути недоступною. Операція, що охоплює три сервіси, здатна завершитися так: у першому зміна зафіксована, у другому відкочена за тайм-аутом, третій взагалі не отримав запит. Зведення такого часткового результату до узгодженого стану не відбувається автоматично й потребує явної координації.

Конкурентні зміни. Коли кілька екземплярів одного сервісу або кілька різних сервісів одночасно працюють з логічно пов'язаними даними, порядок операцій у глобальному масштабі не визначений. Запровадження глобальних блокувань розв'язує проблему на рівні семантики, однак несумісне з вимогами масштабованості: глобальне блокування перетворюється на вузьке місце, що обмежує пропускну здатність усієї системи.

Затримки реплікації. Значна частина сучасних сховищ застосовує реплікацію для підвищення доступності та локалізації читань. Між фіксацією запису на основному вузлі й появою нового значення на репліках проходить певний час. Клієнт, що читає з репліки одразу після запису, може отримати застаріле значення — явище, відоме як *stale read*. Для прикладного коду така поведінка часто виглядає як помилка, хоча технічно сховище працює коректно в межах своєї моделі узгодженості.

Відсутність глобального часу. Годинники на різних вузлах розсинхронізовані, а їхній дрейф не є сталим. Використання міток часу як критерію упорядкування подій у розподіленій системі неприпустиме, якщо порядок має значення для коректності. Задача встановлення причинно-наслідкового порядку розв'язується окремими засобами — векторними годинниками, логічними годинниками Лампорта [10], консенсусними протоколами на зразок Paxos і Raft.

1.2.2 Теорема CAP та модель BASE

Формальне обмеження на те, наскільки перелічені труднощі можна зняти одним універсальним механізмом, дає теорема CAP. Її сформулював Е. Брюер з

Каліфорнійського університету в Берклі у 1998–2000 рр., а строгий доказ опублікували С. Гілберт і Н. Лінч з MIT у 2002 році [12]. Теорема стверджує, що розподілене сховище здатне одночасно гарантувати не більше двох із трьох властивостей: узгодженість (Consistency), доступність (Availability) і стійкість до мережевого розділення (Partition tolerance).

Практична інтерпретація теореми потребує уточнення. У реальних мережах розділення неминуче: жодна мережа не є абсолютно надійною, і рано чи пізно між вузлами відбудеться тимчасова втрата зв'язку. Тому вибір фактично зводиться до двох пар. Системи класу CP при розділенні зберігають узгодженість ціною тимчасової відмови в доступності (приклади — MongoDB у конфігурації з більшістю, HBase). Системи класу AP зберігають доступність, допускаючи тимчасову розбіжність реплік (Cassandra, DynamoDB, Riak). Абсолютно послідовних CA-систем у строгому розподіленому сенсі не існує, про що сам Брюер неодноразово наголошував у коментарях 2012 року [13], переглядаючи початкове формулювання.

У 2010 році Д. Абаді запропонував розширення — теорему PACELC [14]. Вона додає, що навіть за відсутності розділення (else, E) у системі лишається інший компроміс: між затримкою (latency, L) та узгодженістю (C). Синхронна реплікація з гарантіями лінеаризованості оплачується мережевими раундами, асинхронна — послабленням гарантій на користь затримки. PACELC зручніша для опису реальних хмарних систем тим, що враховує не тільки аварійні, а й штатні режими.

З боку практики індустрія сформувала альтернативний до ACID набір принципів — модель BASE (*Basically Available, Soft state, Eventually consistent*), запропоновану в роботах Е. Брюера [13] та інженерів eBay. BASE не заперечує ACID, а описує інший режим роботи: система в нормі доступна, її стан постійно «м'який» (може змінюватися під впливом фонових процесів), а повна узгодженість досягається лише з часом. Саме цей підхід лежить в основі більшості сучасних розподілених сховищ та мікросервісних архітектур.

1.2.3 Спектр моделей узгодженості

Протиставлення «сильної» та «поступової» узгодженості у широкому вжитку є надто грубим. У літературі з розподілених систем (М. Клеппман [11]) вживається ціла ієрархія моделей, упорядкованих за силою гарантій — від найсильніших до найслабших.

Лінеаризованість (linearizability) є найсильнішою практичною моделлю. Операції виглядають так, ніби виконувалися в єдиному глобальному порядку, узгодженому з реальним часом: одразу після завершення запису будь-який наступний запит повертає саме записане значення. Досягається коштом синхронізації між вузлами й, згідно з результатом Аттії та Велч [15], не може виконуватися швидше за певний поріг, визначений затримкою мережі.

Послідовна узгодженість (sequential consistency) послаблює вимогу до реального часу: усі процеси спостерігають один і той самий порядок операцій, однак він не зобов’язаний збігатися з фізичним.

Причинна узгодженість (causal consistency) гарантує лише збереження причинно-наслідкових залежностей. Якщо подія A могла спричинити подію B, усі процеси побачать їх у цьому порядку; непричинно пов’язані події можуть спостерігатися в різній послідовності. Відповідно до аналізу М. Клеппмана [11], це найсильніша модель, що лишається доступною при розділенні мережі.

Сесійні гарантії описують коректність у межах сесії одного клієнта: *read-your-writes* (клієнт завжди бачить власні щойно виконані записи), *monotonic reads* (послідовні читання не «повертаються назад у часі»), *monotonic writes* (записи одного клієнта спостерігаються у порядку виконання). Ці гарантії дешевші за глобальні, бо обмежуються однією сесією, і водночас достатні для низки типових сценаріїв користувацького інтерфейсу.

Поступова узгодженість (eventual consistency) — найслабша з практично вживаних моделей. Вона гарантує лише те, що за відсутності нових записів усі копії даних з часом зійдуться до одного значення; поведінка в перехідний період формально не специфікована й може включати читання застарілих, невпорядкованих або часткових версій.

Вибір моделі визначається бізнес-вимогами. Сильніші гарантії потребують більшої кількості повідомлень між вузлами, триваліших блокувань і знижують доступність. Слабші — переносять складність на прикладний рівень, вимагаючи ідемпотентних оброблювачів, обробки конфліктів і компенсувальних механізмів. Типова мікросервісна система поєднує кілька моделей одночасно: для фінансових операцій та управління доступом у межах одного сервісу застосовуються сильні гарантії на рівні СКБД, для міжсервісних синхронізацій — поступова узгодженість через обмін подіями.

1.2.4 Проблема подвійного запису

Практична ситуація, що виникає щоразу, коли сервіс у межах обробки одного запиту має змінити власне сховище і надіслати зовнішнє повідомлення — подію в брокер, запит до іншого сервісу, електронного листа — у літературі відома як проблема подвійного запису (dual write problem) і докладно описана, зокрема, у матеріалах Confluent [16] і публікаціях Т. Янсена [17].

Зміст проблеми такий. Оброблювач команди виконує два кроки: записує зміну в локальну базу і публікує подію в брокер повідомлень. Обгортка цих кроків у локальну транзакцію бази даних не усуває проблеми, оскільки транзакція охоплює тільки базу, а брокер лишається поза нею. Якщо збій стається після успішного запису, але до публікації, система потрапляє у стан, коли зміна застосована локально, але інші сервіси про неї не знають. Зворотний порядок — спочатку публікація, потім запис — має симетричну ваду: подія вже розіслана й, можливо, спожита, а її джерело до бази не потрапило.

Інтуїтивні способи розв’язання при уважному розгляді виявляються неспроможними. Публікація у post-commit оброблювачі гарантує надсилання тільки після успішного запису, однак не гарантує самого надсилання, якщо на той момент недоступний брокер. Повторні спроби з локальним журналом «невідправлених» подій зводяться до того самого подвійного запису на іншому рівні. Надійні розв’язки обмежуються кількома патернами. Перший — Transactional Outbox: подія зберігається в окрему таблицю тієї самої бази у межах

однієї локальної транзакції, а окремий процес (релей або інструмент CDC на зразок Debezium) згодом публікує її у брокер. Другий — Event Sourcing, де стан сервісу не зберігається явно, а реконструюється з незмінної послідовності подій. Третій — транзакційні можливості самого брокера у зв'язці з коннекторами (наприклад, транзакції Kafka разом з Kafka Connect). Усі вони ускладнюють архітектуру, але надійно замикають подвійний запис у межах одного сервісу.

1.2.5 Обмеження двофазної фіксації у мікросервісному контексті

Класичним розв'язанням задачі узгодженого оновлення кількох ресурсів є протокол двофазної фіксації (two-phase commit, 2PC), стандартизований у специфікації X/Open XA і реалізований у численних корпоративних менеджерах транзакцій (Narayana, Atomikos, JOTM, Microsoft DTC). Протокол розбиває фіксацію на дві фази. У фазі підготовки координатор опитує учасників, чи готові вони зафіксувати зміни; ті виконують перевірки й відповідають «готовий» або «не готовий». У фазі рішення координатор розсилає команду фіксації, якщо всі відповіді позитивні, або команду відкочення — в іншому випадку.

У монолітному корпоративному застосунку з невеликою кількістю ресурсів 2PC забезпечує властивості ACID для розподіленої операції. У мікросервісному середовищі, як неодноразово вказували С. Ньюман [2], К. Річардсон [3], Б. Ібріям [18], застосування 2PC кваліфікується як анти-патерн. Причини тому кілька.

Блокувальна природа протоколу. Від моменту відповіді «готовий» до отримання команди фіксації учасник зобов'язаний утримувати блокування на змінених ресурсах. Тривалість цього інтервалу визначається найповільнішим учасником та станом мережі. Інші операції, що торкаються тих самих даних, мусять чекати. У системі з десятками сервісів і високою частотою операцій над спільними даними такі блокування стають головним обмеженням пропускної здатності.

Вразливість координатора. Якщо координатор виходить з ладу між фазами підготовки й рішення, учасники залишаються у стані невизначеності: вони вже

пообіцяли зафіксувати зміни, але не отримали підтвердження. Без зовнішнього втручання цей стан не розв’язується — потрібне відновлення з персистентного журналу транзакцій, а в найважчих випадках адміністративне рішення з ручним розблокуванням. Трифазна модифікація протоколу (3PC) частково пом’якшує проблему ціною додаткового раунду повідомлень, проте не усуває її при справжньому розділенні мережі.

Обмежена підтримка з боку сховищ. Специфікація XA вимагає, щоб кожен учасник реалізовував інтерфейс диспетчера ресурсів. Більшість реляційних СКБД цю вимогу виконує, однак значна частина NoSQL-сховищ (Cassandra, MongoDB у базовій конфігурації, Redis, бази типу ключ-значення) та багато брокерів повідомлень її не підтримують або підтримують частково. У поліглотному середовищі, характерному для мікросервісів, 2PC стає просто незастосовним для значної кількості комбінацій технологій.

Сильне часове сполучення. Учасники розподіленої транзакції на основі 2PC стають залежними одне від одного в реальному часі: для успішного завершення всі вони мають бути одночасно доступними. Така залежність суперечить принципу незалежного розгортання та ізоляції відмов, на якому будується мікросервісна архітектура. К. Річардсон прямо зазначає [3], що використання 2PC у таких системах нівелює більшість переваг, заради яких і здійснювалася декомпозиція моноліту.

1.3 Підходи до керування транзакціями в мікросервісах

Обмеження двофазної фіксації, розглянуті у попередньому підрозділі, не означають відмови від поняття транзакції як такої. Непридатною на практиці виявляється лише модель ACID поверх кількох розподілених сховищ. Натомість в індустрії сформувався набір патернів, що зміщують гарантії в бік поступової узгодженості, зберігаючи для прикладного коду корисну абстракцію цілісної бізнес-операції. Систематизацію цих патернів подано, зокрема, у працях К. Річардсона [3] та Б. Ібріяма [18]; нижче розглянуто ключові з них у порядку зростання складності координації.

1.3.1 Класифікація підходів

Механізми координації розподілених транзакцій доцільно порівнювати за кількома критеріями. Перший — модель узгодженості: частина підходів прагне семантики, близької до ACID, інші явно обирають поступову узгодженість. Другий — спосіб взаємодії: синхронна координація через виклики запит-відповідь протиставляється асинхронному обміну повідомленнями через брокер. Третій — локалізація логіки координації: її або інкапсулює окремий компонент-оркестратор, або розподіляє між учасниками через реакцію на події (хореографія). Четвертий — джерело надійності: одні патерни покладаються на властивості сховища, інші — на властивості брокера, ще інші — на явні компенсувальні операції у термінах домену.

Концептуальний орієнтир для роботи з цим простором запропонував П. Гелланд у статті «Life beyond Distributed Transactions» [19]. Його теза полягає в тому, що масштабовна система взагалі не має оперувати категорією розподіленої транзакції. Натомість має бути поняття локально транзакційних бізнес-сутностей і явних повідомлень між ними, які опрацьовуються ідемпотентно. Подальші промислові патерни здебільшого конкретизують цю загальну схему для різних класів задач.

1.3.2 Транзакційний поштовий ящик

Прямий розв’язок проблеми подвійного запису пропонує патерн транзакційного поштового ящика (Transactional Outbox). Його ідея полягає в тому, щоб обидва кроки — зміну стану та планування зовнішнього повідомлення — виконувати в межах однієї локальної транзакції єдиного сховища. Для цього в базі даних сервісу створюється допоміжна таблиця outbox, до якої сервіс одночасно із записом основних змін додає рядок із майбутнім повідомленням. Оскільки обидва записи йдуть до однієї бази, вони фіксуються або скасовуються атомарно. Окремий фоновий процес (релей, або ж message relay) періодично читає нові рядки таблиці та публікує їх у брокер, видаляючи або помічаючи рядок після успішної публікації.

Такий підхід зберігає атомарність локально й переносить питання міжсервісної доставки у площину, де вже існують стандартні розв’язки. Публікатор таблиці забезпечує семантику доставки «щонайменше один раз» (at-least-once): при збої після публікації, але до позначення рядка як обробленого повідомлення буде надіслане повторно. Це покладає на споживачів вимогу ідемпотентності — здатності коректно обробити повторне надходження тієї самої події [3; 16].

Практичні недоліки патерну відомі й обмежені. По-перше, таблиця outbox створює додаткове навантаження на базу й потребує регулярного прибирання оброблених рядків. По-друге, між фіксацією локальної транзакції і фактичною публікацією у брокер виникає затримка, зумовлена періодичністю роботи релея. Обидва недоліки послаблюються налаштуваннями — частотою опитування, обсягом пакета, індексацією outbox — але повністю не усуваються.

1.3.3 Захоплення змін даних

Альтернативний спосіб поширення подій у брокер — читання журналу транзакцій бази даних безпосередньо. У будь-якій СКБД із підтримкою відновлення існує фізичний журнал змін (у PostgreSQL — WAL, у MySQL — binlog, у MongoDB — oplog), що заповнюється при кожній фіксації. Якщо підключити окремий процес, який читає цей журнал і транслює зміни у брокер, то сам факт фіксації автоматично гарантує появу події, без жодного додаткового коду в сервісі.

Такий патерн відомий під назвою Change Data Capture (CDC), а найпоширенішим його втіленням є платформа Debezium [20], побудована на Apache Kafka Connect. Debezium підтримує провідні реляційні сховища й кілька нереляційних і подає уніфіковане представлення змін у форматі повідомлень Kafka. Перевагою CDC є цілковите відокремлення прикладного коду від механізму публікації: розробник сервісу може взагалі не знати про існування трансляції.

Зворотним боком цієї непомітності є жорстке зчеплення зі схемою бази даних. Перейменування колонки або зміна типу поширюється на всі сервіси-споживачі; внутрішня модель сховища фактично перетворюється на публічний інтерфейс. М. Клеппман [11] рекомендує пом’якшувати цю залежність явним шаром перетворення, що приводить сирий потік журналу до стабільних доменних подій, однак повністю усунути зв’язок не вдається. З цієї причини CDC частіше обирають для інтеграції успадкованих систем і аналітичних пайплайнів, аніж для первинної реалізації мікросервісної комунікації.

1.3.4 Модель подієвого джерела

Найрадикальніше переосмислення ролі транзакції у розподіленій системі пропонує модель подієвого джерела (Event Sourcing) [21]. Вона відмовляється від зберігання поточного стану як основної одиниці бази: натомість зберігається послідовний журнал подій, що цей стан формують. Поточний стан у будь-який момент обчислюється повторним програванням журналу або, на практиці, пришвидшується періодичними знімками.

Для задачі узгодженості такий підхід має очевидну перевагу: фіксація події в журналі і є тією самою операцією, яку спостерігають інші сервіси ззовні. Подвійного запису просто не виникає, бо іншого запису немає. Публікація у брокер зводиться до пересилання вже зафіксованої події, а відновлення стану після збою — до повторного прочитання журналу з останнього знімка. М. Фаулер [21] звертає увагу також на супутні властивості моделі: природний аудит змін і можливість відтворення історичних станів для аналітики, яка в традиційній архітектурі потребувала б інвазивних змін у коді.

Вартість цих властивостей — інший характер запитів. Отримання поточного стану виконується повільніше, тому Event Sourcing типово поєднується з патерном CQRS (Command Query Responsibility Segregation), що розділяє шлях запису і шлях читання: команди формують події, а окремо побудовані матеріалізовані представлення обслуговують запити. Складність проєктування зростає суттєво, тому модель подієвого джерела раціональна

переважно там, де історія змін сама є частиною предметної області: у фінансовому обліку, бухгалтерії, управлінні резервами, окремих формах аудиту [11; 3].

1.3.5 Ідемпотентність і семантика доставлення

Будь-який патерн на базі асинхронного обміну повідомленнями спирається на припущення щодо кількості фактичних доставок одного й того самого повідомлення. М. Клеппман [11] розрізняє три моделі. Семантика «не більше одного разу» (at-most-once) допускає втрати, але виключає дублікати; вона найпростіша у реалізації й придатна тільки для некритичних операцій. Семантика «щонайменше один раз» (at-least-once) виключає втрати, але припускає дублювання; саме її надають у стандартних режимах переважна більшість практичних брокерів, включно з Apache Kafka та RabbitMQ. Семантика «рівно один раз» (exactly-once) найбажаніша, але в розподілених умовах вимагає додаткових механізмів — транзакційних записів у самому брокері, дедуплікації на боці споживача або комбінації обох.

На практиці системи будують навколо at-least-once у поєднанні з ідемпотентною обробкою. Ідемпотентним називають оброблювач, для якого повторне надходження повідомлення не змінює стан сервісу порівняно з одноразовою обробкою. Технічні прийоми досягнення ідемпотентності добре відомі: перевірка наявності вже обробленого ідентифікатора повідомлення у службовій таблиці, умовне оновлення за ключем із версією, формулювання операцій як присвоєнь замість приростів. П. Гелланд [19] наголошує, що саме ідемпотентність, а не якась форма глобальної транзакції, є центральним механізмом коректності у великих розподілених системах.

1.3.6 Протокол Try-Confirm-Cancel

Окремим класом підходів, що намагається наблизити семантику мікросервісних транзакцій до ACID без блокувальної природи 2PC, є протокол TCC (Try-Confirm-Cancel). Він вимагає від кожного учасника реалізувати не одну

бізнес-операцію, а три. Крок Try попередньо резервує необхідні ресурси без остаточної зміни стану. Крок Confirm остаточно застосовує попередньо зарезервоване. Крок Cancel звільняє резерв у разі невдачі сценарію.

TCC зберігає ідею двофазності, проте реалізує її на прикладному рівні й у термінах домену. Резервом може виступати тимчасовий залишок на рахунку, відкладений складський залишок, попередньо заблоковане місце. На відміну від 2PC, тут блокування керовані самим сервісом і видимі в його моделі даних; тайм-аути, компенсації, відновлення стають частиною бізнес-логіки, а не інфраструктурного шару. Б. Ібріям [18] відносить TCC до «семантично сильних» патернів і зауважує, що його доцільність обмежена випадками, коли природа операції справді допускає осмислене поняття резерву.

Слабкі місця протоколу — значні зусилля на проєктування (кожна операція фактично потроюється), необхідність тайм-аутів і механізму гарантованого виклику Cancel, чутливість до некоректної реалізації Confirm або Cancel. Через ці причини у типових мікросервісних системах TCC застосовується рідко й переважно там, де вимоги до наближеної до ACID поведінки не допускають компромісів, — зокрема, у окремих сценаріях бронювання та платежів.

1.3.7 Саги як інтегрувальна модель

Розглянуті вище патерни розв’язують окремі задачі — атомарність локального запису разом з публікацією, доставку повідомлень без втрат, ідемпотентність споживача, обмежене резервування ресурсів. Проте жоден з них окремо не відповідає на центральне питання мікросервісної координації: як організувати послідовність кроків у кількох сервісах так, щоб у разі часткової невдачі система приходила до узгодженого стану без ручного втручання.

Таку відповідь пропонує модель саги — послідовності локальних транзакцій, кожна з яких публікує подію або команду для наступного кроку, а невдача будь-якого кроку компенсується зворотними операціями над усіма попередніми. Саги активно використовують розглянуті тут механізми: outbox

або CDC для надійної публікації, ідемпотентних споживачів для стійкості до повторів, іноді TCC для окремих кроків, що потребують семантики резервування. Тому сагу доцільно розглядати не як альтернативу описаним патернам, а як рівень вищий за них — інтегрувальну модель розподіленої транзакції, що спирається на попередній інструментарій. Її різновидам, механізмам реалізації та межам застосовності присвячений наступний підрозділ.

1.4 Патерн Saga: концепція, типи (оркестрація та хореографія), переваги й обмеження

Концепцію саги запровадили Г. Гарсія-Моліна і К. Салем у статті 1987 року [22] як розв'язання задачі керування довготривалими транзакціями в монолітних системах керування базами даних. Автори запропонували замінити одну велику транзакцію послідовністю коротких локальних транзакцій, кожна з яких має свій компенсувальний відповідник, здатний семантично скасувати її ефект. У первинному формулюванні сага була внутрішньою конструкцією СКБД і не виходила за межі одного сховища; у розподілений вигляд її адаптували наприкінці 2000-х — у 2010-х років, зокрема в працях К. Річардсона [3], коли проблематика координації мікросервісів стала індустріально значущою.

1.4.1 Формальне визначення

У мікросервісному контексті сагу визначають як послідовність локальних транзакцій $T_1, T_2, \ldots, T_n$, що виконуються в різних сервісах. Кожна транзакція T_i змінює стан одного сервісу в межах його локальної ACID-фіксації і, за наявності, має компенсувальну транзакцію C_i, здатну семантично повернути відповідний сервіс до стану, близького до того, що передував T_i. Якщо всі кроки саги виконано успішно, вона вважається зафіксованою. У разі невдачі на кроці T_i система ініціює послідовне виконання компенсацій $C_{i-1}, C_{i-2}, \ldots, C_1$ у зворотному до основного порядку, після чого сага вважається скасованою.

Ключова відмінність саги від класичної транзакції полягає в понятті компенсації. Компенсувальна операція, на відміну від технічного відкочення

(rollback), не повертає систему точно до попереднього стану: вона є семантично зворотною дією на рівні домену. Скасування броні місця в готелі не стирає з журналу факт звернення клієнта, а лише змінює стан броні на «скасовану». Цей факт може залишитися доступним для аудиту, аналітики, нарахування штрафу за пізню відмову тощо. Саме тому компенсація вимагає проектування в термінах предметної області, а не автоматичного виведення з коду основної операції.

1.4.2 Семантика кроків саги

К. Річардсон [3] пропонує класифікацію кроків саги за їхньою семантичною роллю, що виявляється корисною при побудові сценаріїв відновлення. Компенсувальними (compensatable) називають транзакції, ефект яких може бути скасовано зворотною дією; типовий приклад — резервування позицій на складі, яке може бути знято звільненням резерву. Стрижневою (pivot) є транзакція, після якої сага вже не може відступити — точка незворотності сценарію. Наприклад, фактичне списання коштів з рахунку (на відміну від їх попередньої авторизації) часто виступає стрижневим кроком. Повторюваними (retriable) називають транзакції, що йдуть після стрижневої: вони не мають компенсації і мусять бути виконані, можливо, з повторними спробами, до успішного завершення.

Ця класифікація не є формальною властивістю коду, а відображає рішення проектувальника щодо структури саги. Один і той самий технічний крок у різних сценаріях може належати до різних категорій. Визначення pivot-транзакції — окрема проєктна задача: вона має розміщуватися так, щоб усі незворотні або коштовні для скасування дії виконувалися після неї, а всі дії, які можуть ефективно компенсуватися, — до неї. Добре спроектована сага має не більше однієї стрижневої транзакції; наявність кількох точок незворотності свідчить про те, що сценарій доцільно розбити на кілька незалежних саг.

1.4.3 Оркестрована сага

У реалізації через оркестрацію (orchestration) координацію кроків виконує виділений компонент — оркестратор саги (saga execution coordinator, SEC). Оркестратор зберігає стан саги в персистентному сховищі і керує її прогресом явними командами, які надсилає учасникам. Після виконання кожного кроку учасник повідомляє результат, оркестратор оновлює стан саги і приймає рішення про виконання наступного кроку або про запуск послідовності компенсацій.

З архітектурного погляду оркестратор реалізує скінченний автомат (state machine), де стани відповідають прогресу саги, а переходи — отриманим від учасників відповідям. Логіка координації централізована і явно видима в коді одного компонента, що спрощує розуміння загального потоку, діагностику відмов і модифікацію сценарію. Учасники саги при такій організації залишаються «пасивними»: вони отримують команду, виконують локальну транзакцію, повідомляють результат — і не мають знання про решту сценарію.

Недоліком оркестрації є певна концентрація логіки в одному місці. Оркестратор стає сервісом, який знає про всіх учасників, їхні команди і порядок кроків, що може конфліктувати з принципом незалежного еволюціонування сервісів. К. Річардсон [3] наголошує, що оркестратор має обмежуватися керуванням послідовністю і не перебирати на себе бізнес-правила самих кроків — інакше він перетворюється на централізований компонент, що відтворює на новому рівні проблеми моноліту. На практиці оркестрована сага часто реалізується за допомогою спеціалізованих фреймворків — Axon Framework, Eventuate Tram, Camunda, Temporal, — що надають готовий механізм персистентного скінченного автомата, повторних спроб і таймаутів.

1.4.4 Хореографічна сага

Альтернативний спосіб реалізації — хореографія (choreography) — обходиться без центрального координатора. Кожен учасник саги підписаний на певні події і публікує нові події у відповідь на завершення своїх локальних транзакцій. Стан саги виявляється розподіленим між учасниками і між

повідомленнями, що циркулюють у брокері. Загальна логіка саги виникає як сукупна поведінка з локальних реакцій сервісів, без явного опису сценарію в жодному окремому компоненті.

Хореографія послідовно втілює принцип слабкого зчеплення: сервіси не мають прямих залежностей одне від одного, а взаємодіють лише через спільний канал подій. Нові учасники можуть долучатися до саги без змін у вже існуючих сервісах — достатньо підписатися на відповідну тему в брокері. Для простих саг з двох-трьох кроків такий підхід простіший у реалізації і природно вписується в подієво-орієнтовану архітектуру.

Проте при ускладненні сценарію хореографія виявляє суттєві обмеження. С. Ньюман [2] зазначає, що у сазі з п’яти й більше кроків відстежити фактичний потік виконання стає непросто: логіка «розсіяна» по кількох сервісах, і для розуміння повного сценарію доводиться читати код і підписки в кількох кодових базах одночасно. Діагностика проблемних виконань теж ускладнюється — немає єдиного місця, де відображався б поточний стан конкретного екземпляра саги. Крім того, циклічні залежності між подіями легко виникають непомітно для розробника і можуть призводити до безкінечних сценаріїв або каскадного множення повідомлень. Б. Ібріям [18] підсумовує індустріальну практику так: для саг із трьома й меншою кількістю учасників хореографія прийнятна; при зростанні кількості учасників доцільно переходити до оркестрації.

1.4.5 Переваги патерну

Переваги саги порівняно з двофазною фіксацією очевидні і зумовили її поширення як фактичного стандарту координації розподілених транзакцій у мікросервісах. Сага не вимагає підтримки XA з боку сховищ, що робить її застосовною у поліглотному середовищі з реляційними і нереляційними базами. Вона не блокує ресурси через мережеві межі: кожна локальна транзакція короткочасна і обмежена одним сервісом. Сага толерантна до часткової недоступності учасників — якщо потрібний сервіс тимчасово недоступний, оркестратор або наступний споживач події очікують його відновлення, не

блокуючи інших. Нарешті, сага зберігає принцип незалежного розгортання: учасники можуть оновлюватися окремо, поки стабільним залишається контракт повідомлень.

1.4.6 Обмеження патерну

Обмеження саги не менш суттєві, ніж її переваги, і їх недооцінка є найчастішою причиною невдалих впроваджень. Перше і головне — відсутність ізоляції. Модель ACID містить властивість I (isolation), що гарантує виконання паралельних транзакцій, ніби вони відбуваються послідовно. У саги такої гарантії немає: поки сага виконується, інші операції бачать проміжні стани її учасників. Це призводить до класичних аномалій — брудного читання (читання проміжного стану саги, що згодом буде скасована), втраченого оновлення (одночасної зміни того самого поля двома сагами), нестабільного читання (отримання різних значень при повторних читаннях у межах однієї логічної операції).

К. Річардсон [3] перелічує набір контрзаходів для пом’якшення цих аномалій: семантичний замок (встановлення ознаки «операція в процесі» на рівні бізнес-даних, що сигналізує іншим сагам про необхідність обережного читання), комутативні оновлення (формулювання операцій так, щоб їхній результат не залежав від порядку застосування), песимістичне впорядкування (розташування кроків у послідовності, що мінімізує ймовірність аномалій), повторне читання з перевіркою версії (оптимістичний контроль конкурентності на рівні сутностей), журнал версій і операції у значеннях замість посилань. Усі ці механізми переносять частину задачі ізоляції на прикладний рівень, що вимагає додаткового проєктування.

Друге обмеження — потреба у компенсувальних операціях для кожного компенсувального кроку. Для деяких дій компенсація проектується тривіально: звільнити резерв, видалити створений запис, позначити документ як скасований. Для інших — принципово складна або неможлива. Надіслане повідомлення електронної пошти не відкликається; частково виконаний фізичний процес на

зразок доставки не скасовується програмними засобами. Такі неповоротні кроки мають розміщуватися після стрижневої транзакції, а все, що передує їй, повинне залишатися компенсувальним. Неправильне розташування незворотних дій у сазі є типовою проєктною помилкою, наслідки якої виявляються лише у рідкісних сценаріях відмов, коли виправлення вже коштує значно більше.

Третє обмеження — збільшення обсягу проєктування і тестування. Кожен крок саги потребує не лише основної логіки, а й компенсувальної; кількість сценаріїв, які має покривати тест, приблизно подвоюється. Тестування саг особливо складне, оскільки необхідно моделювати відмови на кожному етапі і перевіряти, що послідовність компенсацій справді приводить систему до узгодженого стану. На практиці повне покриття всіх можливих точок відмови вимагає спеціалізованих інструментів chaos-тестування, які в мікросервісному середовищі застосовуються далеко не всюди.

Четверте обмеження — обов’язковість ідемпотентності. Кроки саги виконуються через асинхронний обмін повідомленнями, що має семантику «щонайменше один раз», тому дублікати повідомлень — норма, а не виняток. Кожна локальна транзакція і кожна компенсація мусять бути ідемпотентними, тобто дозволяти безпечне повторне виконання без спотворення стану. Б. Ібріям [18] окремо наголошує, що неуважність до цієї вимоги є найтиповішим джерелом помилок у реалізаціях саг: у типовому сценарії оборонного тестування сага працює коректно, а при збої мережі з повторним надсиланням повідомлення призводить до подвійного списання, подвійного нарахування або іншого порушення бізнес-інваріантів.

Висновки до розділу 1

Попри перелічені обмеження, патерн саги на сьогодні є основним інструментом координації розподілених транзакцій у мікросервісних системах. Його переваги — відповідність архітектурному принципу незалежних сервісів, відсутність блокувальних протоколів, толерантність до часткових відмов — у більшості практичних випадків переважають витрати на додаткове

проєктування. Вибір між оркестрацією і хореографією визначається передусім складністю сценарію: для коротких і лінійних саг хореографія спрощує реалізацію, для складних і розгалужених — оркестрація забезпечує кращу керованість і діагностованість. У наступному розділі розглядається задача проектування мікросервісної системи, в якій координація транзакцій реалізується через оркестровану сагу, з аналізом вимог, архітектурної моделі і технологічного стека.

РОЗДІЛ 2. ПРОЄКТУВАННЯ ТА РОЗРОБКА СИСТЕМИ

2.1 Постановка задачі та вимоги до системи

У межах практичної частини даної роботи об’єктом проєктування виступає програмна бібліотека *SagaFlow* — SDK для платформи .NET, призначений для декларативного опису й виконання Saga-оркестрацій у розподілених мікросервісних застосунках. Постановка задачі спирається на спостереження, сформульоване в теоретичному розділі роботи: реалізація координації розподілених бізнес-операцій у мікросервісних середовищах стикається з відносно сталим набором інженерних проблем, кожна з яких потребує обережного розв’язання і жодна з яких не є тривіальною у практичному втіленні. З огляду на це, метою проєктування системи є виокремлення тих інфраструктурних турбот у готовий багаторазово придатний компонент, який звільняє розробника прикладного мікросервісного застосунку від необхідності реалізовувати їх повторно в кожному новому проєкті.

Розробник, який самостійно впроваджує Saga-оркестрацію в межах окремого проєкту, одразу опиняється перед задачею керування станом тривалої операції, розтягнутої в часі між кількома повідомленнями: цей стан потрібно надійно зберігати у сховищі, відновлювати після перезапусків процесу й коректно обробляти ситуації, у яких декілька пов’язаних повідомлень надходять майже одночасно й потенційно конкурують за одне й те саме місце в сховищі. Паралельно постає так звана проблема подвійного запису (dual-write problem), описана в літературі з інтеграційних патернів [3; 6]: зміна стану саги й публікація вихідного повідомлення повинні відбуватись як одне атомарне ціле, інакше система може опинитись у внутрішньо неузгодженому стані, з якого не існує очевидного автоматичного виходу. Стандартним способом розв’язання цієї проблеми прийнято вважати патерн Transactional Outbox, за яким вихідне повідомлення спершу записується у спеціальну таблицю всередині тієї ж бази даних, що й стан саги, і лише згодом асинхронно пересилається у зовнішній брокер — уже окремим фоновим процесом.

До названих задач додається проблема ідемпотентності обробки повідомлень. Сучасні брокери (RabbitMQ, Apache Kafka, Azure Service Bus) за замовчуванням гарантують семантику доставки at-least-once, тобто одне й те саме повідомлення може бути доставлене повторно, і обробники зобов’язані бути стійкими до такого повтору, інакше подвійне списання коштів, повторна відправка листа або створення двох ідентичних замовлень залишаються лише питанням часу. Окремо постає задача вибору коректної стратегії повторних спроб — з експоненційною затримкою й випадковим зміщенням (jitter), обмеженням максимального часу виконання й, за потреби, розмиканням кола (circuit breaker), — без якої система не може вважатися готовою до промислової експлуатації. Нарешті, компенсаційні операції, які є зворотним боком будь-якої саги, вимагають коректного впорядкованого виконання у порядку, оберненому до початкової послідовності кроків, з урахуванням окремих ситуацій, коли відкат уже принципово неможливий: типовим прикладом є момент після передавання коштів зовнішньому платіжному провайдеру, повернення яких не є технічно допустимим у рамках саги й потребує окремого бізнес-процесу.

Спроба розв’язати всі ці задачі безпосередньо в коді бізнес-логіки конкретного застосунку зазвичай приводить до одного з двох однаково небажаних результатів. У першому випадку виникає локальне рішення, у якому інфраструктурні турботи нерозривно перемішані з бізнес-правилами, а подальша підтримка такого коду швидко стає джерелом помилок і знижує швидкість розробки. У другому — обирається велика універсальна інтеграційна бібліотека на зразок MassTransit або NServiceBus, яка перелічені задачі справді розв’язує, однак нав’язує власну модель обміну повідомленнями, власні розширення контейнера впровадження залежностей та значний обсяг транзитивних залежностей, і виявляється надлишковою для проєктів, у яких потрібна виключно Saga-оркестрація. Саме таке спостереження обґрунтовує методичну доцільність розробки окремої, свідомо вузької бібліотеки, що розв’язує одну задачу й робить це прозорим для користувача способом.

На підставі наведеного аналізу сформульовано набір функціональних вимог до системи. Бібліотека повинна надавати декларативний Fluent API для опису стан-машини саги — її станів, переходів і обробників повідомлень, — не вимагаючи від розробника написання жодного інфраструктурного коду для керування станом, транзакціями, повторами чи маршрутизацією. Для кожного кроку саги має бути передбачена можливість реєструвати компенсаційне повідомлення, яке у разі збою на подальших кроках автоматично публікується у порядку, оберненому до вже виконаних кроків, із зупинкою у попередньо зазначеній точці неповернення (pivot point), після проходження якої автоматичний відкат вважається непридатним. Атомарність зміни стану саги й публікації вихідного повідомлення повинна забезпечуватись реалізацією Transactional Outbox на рівні самої бібліотеки — у такий спосіб, щоб розробник у прикладному коді мав справу лише з фактом публікації, не переймаючись синхронізацією транзакцій між сховищем і транспортом. Дедуплікація повідомлень має виконуватись автоматично на підставі криптографічного хешу, обчисленого з набору властивостей, позначених спеціальним атрибутом; політики повторних спроб — конфігуруватись декларативно через атрибутний інтерфейс із підтримкою фіксованої, лінійної та експоненційної стратегій затримки, опціонального jitter, обмеження часу виконання та розмикача кола.

Окремо виділено вимоги, що стосуються керування конкурентним доступом і технологічної незалежності. Бібліотека повинна підтримувати обидва режими одночасного доступу до одного екземпляра саги — песимістичний, за якого рядок у сховищі блокується на рівні СКБД через конструкцію SELECT … FOR UPDATE, і оптимістичний, побудований на перевірці версійного токена, — з можливістю вибору режиму для кожного окремого типу саг. Вхідні повідомлення мають автоматично маршрутизуватись або до відповідного екземпляра саги за ідентифікатором кореляції, або до прямого обробника у випадках, коли повідомлення не належить жодній сазі. При цьому ядро бібліотеки не повинно містити жодної залежності від конкретного брокера повідомлень чи типу сховища: підтримка нових транспортів і нових типів

сховищ має додаватись без жодних змін у ядрі — виключно через реалізацію відповідних абстракцій, винесених у це ядро. Така вимога є не лише технічною, а й стратегічною: вона визначає межу, за якою система перестає бути придатною, якщо її порушити навіть у незначному обсязі.

У частині нефункціональних характеристик система повинна забезпечувати мінімальний накладний витрат на обробку одного повідомлення: внутрішній конвеєр активностей будується на делегатах без зайвих алокацій об’єктів у купі, а обчислення ідемпотентного ключа виконується у стилі stackalloc, що цілковито уникає виділення пам’яті. Горизонтальне масштабування досягається конкурентною обробкою повідомлень із контролем зворотного тиску: у внутрішньопроцесному транспорті застосовуються канали обмеженої місткості з режимом Wait, у зовнішньому транспорті — параметр prefetch count з боку брокера. Надійність гарантується семантикою доставки не гіршою за at-least-once завдяки Transactional Outbox, а стійкість до перезапусків процесу — використанням реляційного сховища. Модульність архітектури забезпечується через plug-in модель адаптерів, а тестованість — через внутрішньопроцесні (in-memory) реалізації всіх інфраструктурних абстракцій, які дають змогу запускати інтеграційні тести без будь-яких зовнішніх залежностей, та через атрибут InternalsVisibleTo, що відкриває внутрішні класи ядра тестовому проєкту без оприлюднення їх у публічному API.

Межі застосування системи свідомо окреслено однією задачею. SagaFlow не є повним інтеграційним фреймворком і не претендує на роль сервісної шини: поза межами системи лишаються задачі маршрутизації на основі вмісту повідомлення (content-based routing), трансформації та збагачення повідомлень у дорозі, графічний дизайнер саг, а також окремий інтерфейс моніторингу виконання. Таке обмеження сфери не є обмеженням можливостей — це проєктне рішення, що дає змогу зберегти мінімальне й прозоре ядро, у якому розробник може розібратись за один вечір, не прив’язувати користувача ані до конкретного брокера, ані до конкретного сховища, і уникнути ситуації, коли підключення бібліотеки тягне за собою десятки транзитивних залежностей, частина з яких

ніколи не буде використана. Цільовою платформою обрано .NET 10, що передбачає наявність у прикладному застосунку стандартного контейнера впровадження залежностей (Microsoft.Extensions.DependencyInjection) та моделі фонових служб (IHostedService); реляційний адаптер орієнтовано на системи керування базами даних, які підтримуються Entity Framework Core, причому режим песимістичної конкурентності потребує семантики SELECT … FOR UPDATE, характерної для PostgreSQL та Microsoft SQL Server і відсутньої у деяких вбудованих рушіях.

2.2 Архітектурна модель системи

Архітектуру системи SagaFlow побудовано за принципами гексагональної архітектури (*Hexagonal Architecture*, або *Ports and Adapters*), описаної А. Кокберном у 2005 році [7]. Центральна ідея цього підходу полягає у свідомому розділенні програмного коду на незалежне від інфраструктури ядро (*application core*), у якому зосереджено бізнес-логіку, та набір адаптерів, що підключають ядро до конкретних зовнішніх технологій — брокерів повідомлень, систем зберігання, мережевих протоколів. Ядро описує свої потреби у формі абстрактних інтерфейсів — так званих портів, — а адаптери надають конкретні реалізації цих портів для обраних технологій. Такий поділ природно випливає з функціональної вимоги про технологічну незалежність, сформульованої у попередньому підрозділі: бізнес-логіка саг, написана користувачем бібліотеки, за жодних умов не повинна залежати від того, чи обробка повідомлень відбувається через RabbitMQ, Apache Kafka або внутрішньопроцесний канал, і зміна транспорту не повинна вимагати перекомпіляції бодай одного рядка прикладного коду.

Структурно цей принцип втілений у поділі вихідного коду системи на п’ять незалежних пакетів NuGet, кожен з яких має чітко окреслену роль. Пакет SagaFlow.Core утворює ядро системи й містить реалізацію стан-машини саги, конвеєра обробки повідомлень, патерну Transactional Outbox, механізму керування транзакціями та реєстрації саг у контейнері впровадження

залежностей; саме тут визначено всі абстрактні порти системи і локалізовано єдину нетривіальну бізнес-логіку. Пакет SagaFlow.InMemory пропонує адаптер, розрахований на роботу в межах одного процесу: транспорт у ньому реалізовано на основі стандартної бібліотеки System.Threading.Channels, а сховища саг, outbox і ідемпотентності — на структурі ConcurrentDictionary. Такий адаптер свідомо не призначений для продакшн-середовищ, проте є особливо зручним для розробки й для автоматизованого тестування, оскільки не потребує розгортання жодних зовнішніх залежностей. Пакет SagaFlow.Sql містить реляційний адаптер, побудований на Entity Framework Core, і реалізує збереження стану саги у форматі JSON, таблицю outbox, таблицю ідемпотентності, підтримку транзакцій та обох режимів конкурентності. Пакет SagaFlow.Postgres є надбудовою над SagaFlow.Sql для роботи з PostgreSQL — він автоматично реєструє драйвер Npgsql і налаштовує схему бази даних, звільняючи користувача від необхідності виконувати ці кроки вручну. Нарешті, пакет SagaFlow.Rabbitmq реалізує адаптер транспорту для RabbitMQ: публікація в ньому виконується через Direct Exchange з маршрутизацією за повним ім’ям CLR-типу повідомлення, споживання — через AsyncEventingBasicConsumer з ручним підтвердженням обробки, а окремий компонент керування підключенням враховує ситуації повторної ініціалізації після тимчасового розриву зв’язку з брокером.

Внутрішня структура ядра організована довкола шести основних абстракцій-портів, через які воно звертається до інфраструктури. Інтерфейс ISagaRepository<TSaga> відповідає за сховище екземплярів саг і передбачає операції збереження, пошуку за ідентифікатором кореляції, видалення та, за потреби, блокування рядка на рівні сховища. Інтерфейс IOutboxStore описує сховище повідомлень, що чекають на публікацію, і гарантує атомарність запису в межах однієї транзакції разом зі зміною стану саги; за перевантаження повідомлень зі сховища outbox у реальний транспорт відповідає окремий порт IOutboxDispatcher. Інтерфейс IMessageSubscriber описує підписник на зовнішнє джерело повідомлень — чергу брокера, внутрішньопроцесний канал або зовнішній вебхук, — який запускається у фоновій службі й передає отримані

повідомлення до ядра для подальшої обробки. Додатково передбачено інтерфейс IIdempotencyStore для зберігання вже оброблених ідемпотентних ключів на рівні безпосереднього обробника повідомлення, а також ITransactionManager — абстракцію транзакції, що охоплює одночасно сховище саг і сховище outbox і забезпечує атомарність їхніх спільних змін. Саме навколо цих шести абстракцій вибудовано усю подальшу архітектуру системи, і їхня реалізація у конкретних адаптерах дає змогу без модифікацій ядра підключати нові технології зберігання й обміну повідомленнями.

Шлях, який проходить одне повідомлення від моменту отримання з транспорту до завершення обробки, має чітко визначену послідовність кроків, що умовно поділяється на три частини: отримання й маршрутизацію, виконання бізнес-логіки саги та асинхронну публікацію нових повідомлень. Схематичне представлення цього процесу зображено на рисунку 2.1; нижче наведено його словесний опис. На першому етапі підписник, запущений у межах фонової служби SubscribersBackgroundService, отримує повідомлення з транспорту, виконує його десеріалізацію й для кожного окремого повідомлення створює ізольовану область видимості контейнера впровадження залежностей (DI scope). Така ізоляція є принциповою: саме завдяки scope екземпляри контексту бази даних, репозиторіїв та транзакції, що обслуговують одне повідомлення, не перетинаються з тими, що обслуговують інше, навіть у разі повністю паралельної обробки кількох повідомлень в одному процесі. Десеріалізоване повідомлення передається до центрального компонента MessageProcessor, який визначає маршрут на підставі попередньо побудованого реєстру саг: якщо тип повідомлення зареєстрований у щонайменше одній сазі, обробка передається до SagaProcessor, параметризованого відповідним типом; якщо жодна сага не очікує цього типу, запускається прямий обробник, що реалізує інтерфейс IMessageConsumer<T> і не веде стану між викликами.

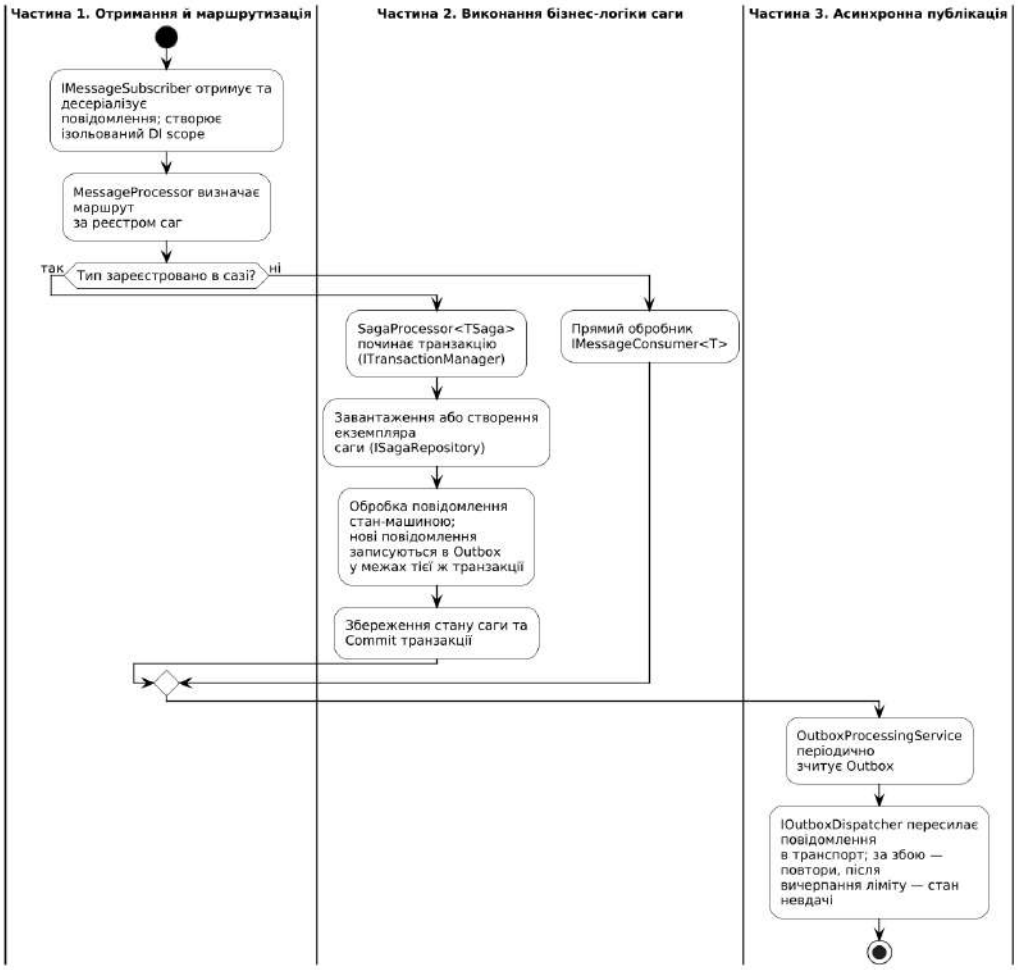

Рисунок 2.1 – Узагальнена схема шляху обробки одного повідомлення в SagaFlow

Компонент SagaProcessor виконує сам цикл обробки повідомлення саги. Він починає транзакцію через ITransactionManager, завантажує або створює екземпляр саги через ISagaRepository, передає його стан-машині для обробки отриманого повідомлення, після чого зберігає новий стан і завершує транзакцію. Принципова особливість полягає в тому, що під час виконання стан-машина може публікувати нові повідомлення — однак ці повідомлення на даному етапі записуються не безпосередньо в зовнішній транспорт, а у сховище outbox, причому в межах тієї ж транзакції, що й зміна стану саги. Лише після успішної фіксації транзакції окрема фонова служба OutboxProcessingService періодично зчитує нові записи з таблиці outbox і пересилає їх у реальний транспорт через IOutboxDispatcher, виконуючи у разі тимчасових збоїв повторні спроби з

інкрементом лічильника помилок і, за потреби, переходом повідомлення в стан постійної невдачі після вичерпання ліміту повторів. Саме така двоетапна схема публікації й реалізує патерн Transactional Outbox, що гарантує атомарність пари «зміна стану саги + публікація вихідного повідомлення» і водночас звільняє основний потік обробки від блокування на операціях мережевого запису в брокер, який може бути повільним або тимчасово недоступним.

Налаштування системи здійснюється у стилі конфігуратора: користувач отримує об’єкт SagaFlowConfigurator, на якому послідовно, через ланцюжок викликів, обирає адаптери інфраструктури й реєструє конкретні саги. Кожна реєстрація технічно є записом у стандартний контейнер Microsoft.Extensions.DependencyInjection, що робить SagaFlow природно сумісним з будь-якими іншими бібліотеками, побудованими на тому самому контейнері, — від EF Core і Serilog до ASP.NET Core, — без потреби в окремих обгортках чи адаптаціях. Життєвий цикл зареєстрованих компонентів обрано з урахуванням природи кожного з них. Визначення стан-машини (SagaStateMachine<T>) реєструється як singleton, оскільки після побудови воно є незмінним, і зберігання його в одному екземплярі економить і пам’ять, і час повторної ініціалізації. Натомість компоненти, що безпосередньо обробляють повідомлення — SagaProcessor, SagaExecutor, конкретні реалізації ISagaRepository та сам MessageProcessor, — реєструються з областю видимості scoped. Саме ця область гарантує, що в межах обробки одного повідомлення буде створено лише один екземпляр контексту бази даних і лише одну транзакцію; без такого обмеження неможливо було б коректно реалізувати Transactional Outbox, оскільки запис у сховище outbox і зміна стану саги обов’язково мусять ділити спільну транзакцію. Якщо користувач не підключив жодного інфраструктурного провайдера, що надавав би реалізацію ITransactionManager — типова ситуація при запуску в повністю in-memory режимі, — система реєструє NoOpTransactionManager як запасний варіант — завдяки цьому вдається уникнути зайвих помилок про відсутність обов’язкового сервісу, не створюючи

водночас неправдивого враження про наявність справжніх розподілених транзакцій.

2.3 Вибір технологій та інструментів для реалізації

Цільовою платформою для реалізації системи обрано .NET 10 — останню на момент написання роботи версію .NET з тривалим циклом підтримки (Long Term Support). Порівняно з попередніми версіями ця платформа дає помітний приріст продуктивності в асинхронній обробці завдяки оптимізаціям у System.Threading.Channels та у реалізації Task, а також містить низку вдосконалень у бібліотеці базових класів, що стосуються, зокрема, генерації унікальних ідентифікаторів і роботи з криптографічними функціями. Мовою програмування обрано C# 14, що дало змогу активно застосувати сучасні мовні конструкції: первинні конструктори, файлові простори імен (file-scoped namespaces), колекційні вирази, вдосконалене зіставлення зі зразком, а також метод Guid.CreateVersion7(), який генерує монотонно зростаючі ідентифікатори UUID сьомої версії. Останнє є принциповим саме для ідентифікації повідомлень у сховищі outbox, оскільки природне впорядкування таких ідентифікаторів збігається з хронологічним порядком їх створення й виключає потребу в окремому часовому індексі під час вибірки непідтверджених повідомлень.

Вибір зовнішніх бібліотек, використаних у реалізації, підпорядкований двом міркуванням — зрілості рішення у межах .NET-екосистеми та мінімальності набору залежностей у ядрі. Ядро SagaFlow.Core залежить лише від двох зовнішніх бібліотек, не враховуючи самого .NET: Polly.Core — для побудови стратегій стійкості (fault-handling), і абстракцій Microsoft.Extensions.DependencyInjection.Abstractions та Microsoft.Extensions.Hosting.Abstractions — для інтеграції зі стандартними моделями впровадження залежностей і фонових служб. Бібліотека Polly у .NET-екосистемі є фактичним стандартом для побудови політик повторів, таймаутів і розмикачів кола; вона входить до переліку рішень, які Microsoft рекомендує у документації Azure Architecture Center [9], і має широку практику промислового

застосування. У SagaFlow саме через її модель resilience pipeline реалізовано атрибутне налаштування повторних спроб обробників повідомлень і обмеження часу виконання одного кроку саги, без потреби у власноруч написаному коді експоненційних затримок чи лічильників помилок.

Решта залежностей локалізована в конкретних адаптерах і жодним чином не проникає до ядра системи. Реляційний адаптер SagaFlow.Sql використовує Entity Framework Core для об'єктно-реляційного зіставлення стану саги й таблиць outbox та ідемпотентності: оптимістична конкурентність реалізована через стандартний механізм IsConcurrencyToken, ефективне оновлення статусу повідомлень у таблиці outbox — через метод ExecuteUpdateAsync, що виконує оновлення одним SQL-оператором, без завантаження сутностей у пам'ять процесу. Адаптер SagaFlow.Postgres підключає драйвер Npgsql і використовує характерну для PostgreSQL семантику SELECT … FOR UPDATE для реалізації режиму песимістичної конкурентності — рядок саги блокується на рівні СКБД до завершення поточної транзакції, що виключає можливість одночасних змін з боку кількох конкурентних повідомлень. Адаптер транспорту SagaFlow.Rabbitmq побудований на бібліотеці RabbitMQ.Client: публікація виконується через Direct Exchange з маршрутизацією за повним іменем CLR-типу повідомлення, споживання — через AsyncEventingBasicConsumer з ручним підтвердженням (acknowledgement) обробки, а керування кількістю непідтверджених повідомлень здійснюється параметром BasicQos, який обмежує пам'ять, зайняту ще не обробленими повідомленнями на стороні обробника. Для внутрішньопроцесного транспорту в пакеті SagaFlow.InMemory жодних зовнішніх залежностей не потребується: бібліотека System.Threading.Channels, що входить до самого .NET, надає канали обмеженої місткості з режимом Wait, який природно реалізує зворотний тиск — у разі заповнення каналу виробники блокуються, замість того щоб безмежно накопичувати повідомлення в пам'яті процесу.

Серіалізація повідомлень і стану саги виконується через System.Text.Json — стандартну бібліотеку серіалізації .NET, вибір якої дозволяє уникнути

залежності від сторонніх рішень на зразок Newtonsoft.Json. Для повторного завантаження екземпляра саги з бази даних використовується режим PreferredObjectCreationHandling.Populate, який при десеріалізації зберігає вже створені колекції всередині об’єкта, замість того щоб створювати їх заново; це є принциповим для списку завершених кроків (completed steps), у якому індекси виконаних кроків накопичуються поступово, і втрата цих значень між збереженнями зробила б механізм компенсації непридатним до роботи. Обчислення ідемпотентного ключа виконується за алгоритмом SHA-256 у стилі без виділення пам’яті на купі: метод SHA256.TryHashData із простору імен System.Security.Cryptography отримує вхідні дані у вигляді стеково розміщеного буфера (stackalloc), що цілковито уникає алокацій для коротких ключів — а саме такі ключі типово використовуються в повідомленнях бізнес-рівня.

Для тестування системи застосовано стандартний у .NET-екосистемі набір інструментів: фреймворк xUnit версії 3, бібліотеку перевірок Shouldly з fluent-стилем викликів та засіб створення заглушок NSubstitute. Тестова піраміда складається з трьох рівнів — модульних тестів для ізольованих компонентів ядра, інтеграційних тестів із in-memory-адаптерами та тестів наскрізного рівня з реальними RabbitMQ і PostgreSQL, запущеними через контейнери Testcontainers на час виконання тестового прогону. Окремого коментаря заслуговує використання атрибута InternalsVisibleTo. Внутрішні класи ядра, не призначені для прямого використання зі сторони клієнтського коду — зокрема, SagaExecutor, OutboxProcessor, ConsumerInvoker, — оголошено з модифікатором internal sealed, що дозволяє зберегти компактний публічний API й водночас забезпечити інкапсуляцію внутрішньої реалізації, вільної від неявних контрактів з користувачем. Для повноцінного тестування цих класів тестовий проєкт оголошено «другом» основного складання через атрибут InternalsVisibleTo — стандартну у .NET практику, яка відкриває можливість перевіряти внутрішню логіку, не оприлюднюючи її та не роздуваючи публічний інтерфейс бібліотеки.

Серед архітектурних патернів, застосованих у реалізації, найвагоміше місце посідає патерн оркестрованої саги (Saga Orchestration) за класифікацією К.

Річардсона [3]: центральний координатор — клас SagaStateMachine<T> — керує усіма переходами між кроками й усіма публікаціями вихідних повідомлень, на відміну від хореографічного варіанта, за якого учасники обмінюються подіями безпосередньо і загальний потік процесу ніде в коді не представлений явно. Оркестровану модель вибрано свідомо, оскільки вона суттєво спрощує спостережуваність і відлагодження системи: увесь потік виконання конкретного бізнес-сценарію читається в одному місці, в описі стан-машини, і не вимагає відновлення логіки процесу з розрізнених подій у різних сервісах. Разом з цим патерном застосовано Transactional Outbox для розв'язання проблеми подвійного запису, Ports and Adapters на рівні структурного поділу коду, Ланцюжок відповідальності (Chain of Responsibility) [10] — для побудови конвеєра активностей усередині стан-машини, де кожна активність (перехід стану, публікація повідомлення, реєстрація компенсації) обгорнута в окрему об'єкт-ланку, яка делегує виклик наступній, а термінатор ланцюжка (LastBehavior) завершує його роботу. Нарешті, патерн одиниці роботи (Unit of Work) реалізовано імпліцитно через scoped-область видимості DI: в межах обробки одного повідомлення всі компоненти, що працюють з базою даних — контекст EF Core, репозиторій саги, сховище outbox, — ділять одну область видимості й одну транзакцію, завдяки чому їхні зміни фіксуються або відкочуються як одне нероздільне ціле.

2.4 Проєктування механізмів Saga-оркестрації

2.4.1 Формальна модель саги як теоретичний фундамент проєктування

Сага в її оригінальному трактуванні Г. Гарсіа-Моліни та К. Салема [22] є послідовністю локальних ACID-транзакцій $S = (T_1, T_2, ..., T_n)$, доповнених частковим відображенням у множину компенсаційних транзакцій $(C_1, C_2, ..., C_{n-1})$. Для кожної пари $\langle T_i, C_i \rangle$ автори постулюють семантичну інваріанту: виконання C_i після вже завершеного T_i відновлює систему в стан, еквівалентний за бізнес-змістом передумовному. Саме ця інваріанта відрізняє компенсацію від

фізичного відкату: C_1 не є оберненою функцією T_1, а радше окремою транзакцією, що нейтралізує бізнес-ефект T_1 доступними в домені засобами.

Ключовим наслідком такого означення є відсутність властивості ізоляції (Isolation), що є однією з базових у ACID-моделі одиночної транзакції. Проміжні стани саги є видимими для інших процесів протягом усього її виконання, що порушує серіалізовність і вимагає від проєктувальника окремого осмислення того, як паралельні саги можуть спостерігати часткові результати одна одної. Такий компроміс є свідомим: саги доцільно застосовувати саме в тих доменах, де тривалість транзакції або її розподілена природа роблять глобальну ізоляцію економічно невиправданою, а бізнес-вимоги припускають асинхронну узгодженість у сенсі М. Клеппмана [11].

Для потреб проєктування бібліотеки ця формальна послідовність доповнена двома атрибутами. Перший — функція переходу $\delta: S \times M \to S'$, що для поточного стану саги та вхідного повідомлення визначає наступний стан та побічні ефекти (публікації повідомлень, реєстрацію компенсацій). Такий апарат перетворює сагу з лінійної послідовності на скінченний автомат, у якому побічні ефекти прив’язано до переходів. Другий атрибут — прапорець незворотності (pivot), що ділить послідовність на дві підпослідовності: передпівотну, у якій компенсація залишається семантично можливою, і постпівотну, у якій бізнес-процес уже не може бути відкочений технічними засобами.

Обрана формалізація має прямі наслідки для проєктування інтерфейсу бібліотеки. Оскільки сага є автоматом, її опис природно розкладається на оголошення станів, оголошення тригерів (типів повідомлень, що ініціюють переходи) та оголошення ефектів, що виконуються під час переходу. Ця декомпозиція стає основою доменно-специфічної мови, що обговорюється нижче.

2.4.2 Порівняння підходів до координації: хореографія проти оркестрації

Проєктувальник розподіленого бізнес-процесу стикається з архітектурним вибором між двома основними підходами, які в літературі позначають термінами «хореографія» та «оркестрація» [2, 3]. У хореографічній моделі координаційна логіка розподілена між учасниками процесу: кожен сервіс реагує на події інших сервісів власним локальним рішенням, а глобальна поведінка виникає як емерджентна властивість їхніх взаємодій. У оркестраційній моделі, навпаки, координація зосереджена в окремому компоненті-координаторі, який приймає рішення про наступний крок процесу та розсилає відповідні команди учасникам.

Компаративний аналіз цих підходів охоплює кілька вимірів. У вимірі еволюційності хореографія ускладнює модифікацію бізнес-процесу: додавання нового кроку часто вимагає одночасної зміни кількох сервісів, що створює координаційний оверхед у команді розробників і є джерелом типових помилок «часткового розгортання», коли одна частина системи вже знає про новий крок, а інша — ще ні. Оркестрація локалізує зміну в одному компоненті — координаторі — і дозволяє одноразове атомарне оновлення процесу.

У вимірі спостережуваності хореографічна сага має принципово обмежені можливості моніторингу: поточний стан процесу не зберігається в жодному окремому місці, а реконструюється за слідом подій у журналі повідомлень. У промислових системах така реконструкція є дорогим і нетривіальним завданням, що особливо гостро виявляється під час розслідування інцидентів. Оркестраційна сага, навпаки, має явне матеріалізоване представлення стану в координаторі, що робить діагностичні запити прямолінійними та уможливлює побудову операційних дашбордів без архітектурних ухилів.

У вимірі зв’язаності хореографія мінімізує технічне зв’язування (сервіси не знають один про одного напряму), але підвищує семантичне — кожен учасник повинен знати структуру глобального процесу, щоб правильно реагувати на події. Оркестрація обмежує семантичне знання координатором, але створює технічну залежність усіх учасників від нього. Вибір між цими компромісами

визначається природою домену: для стабільних і простих процесів хореографія може бути економнішою, для складних і часто змінюваних — оркестрація.

На підставі цього аналізу архітектура SagaFlow зорієнтована на оркестраційний підхід. Такий вибір відповідає типовим характеристикам бізнес-процесів у доменах, для яких бібліотека проєктується: багатокрокові сценарії з розгалуженою умовною логікою, необхідність інтроспекції стану для регуляторних вимог і відносно швидка еволюція процесу під впливом продуктових рішень. У сценаріях, де хореографія є адекватнішим вибором, SagaFlow не заважає її реалізації на прикладному рівні — прямі споживачі повідомлень можуть обмінюватися подіями без участі центрального координатора.

2.4.3 Декларативна модель опису саги як дизайн-рішення

Другим принциповим проєктним вибором є спосіб, яким розробник-користувач бібліотеки описує свій бізнес-процес. У програмуванні існують два контрастних підходи до побудови подібних інтерфейсів: імперативний, за якого процес описується послідовністю команд у керуючому коді, і декларативний, за якого процес задається як дані — набір правил, тригерів і переходів, — що далі інтерпретуються рушієм [11].

Декларативне представлення має кілька переваг, важливих саме для саги. Читаність специфікації процесу наближається до природно-мовного опису, що полегшує перевірку бізнес-логіки предметними експертами, які не є розробниками. Статичний опис доступний для інтроспекції: рушій може побудувати граф станів, перевірити його на досяжність усіх станів із початкового, виявити непокриті переходи або нескінченні цикли. Зміна процесу локалізується у специфікації без необхідності зміни самого рушія. Нарешті, декларативний опис природно серіалізується, що відкриває можливість конфігурування саги з зовнішніх джерел (файлів, баз даних) без рекомпіляції коду. Остання можливість не використовується в поточній версії SagaFlow, але

залишається відкритою для майбутніх розширень, оскільки сам опис уже відокремлено від рушія.

Імперативне представлення, своєю чергою, забезпечує більшу гнучкість: розробник може вводити довільні керуючі конструкції, обчислювати умови переходів під час виконання і реалізовувати патерни, що виходять за межі наперед визначеного набору операцій. Платою за цю гнучкість є складність інтроспекції: імперативний опис важко проаналізувати статично, і розробник змушений реалізовувати діагностичну інфраструктуру самостійно.

Для SagaFlow обрано декларативний підхід через внутрішній DSL, вбудований у синтаксис C#. Такий дизайн дозволяє користуватися перевагами декларативного опису, зберігаючи при цьому типову безпеку та інструментарій мови-носія. Ескізний опис саги бронювання поїздки має такий вигляд:

```
Initially(
    When<BookTripCommand>()
        .Then(ctx => ctx.Instance.TripId = ctx.Message.TripId)
        .Publish(ctx => new ReserveHotelCommand(...))
        .TransitionTo(ReservingHotel));

During(ReservingHotel,
    When<HotelReservedEvent>()
        .WithCompensation(i => new
CancelHotelCommand(i.CorrelationId))
        .Publish(ctx => new BookFlightCommand(...))
        .TransitionTo(BookingFlight));
```

Наведений фрагмент фактично є специфікацією бізнес-процесу: його можна читати як блок-схему, у якій кожен виклик During(state, ...) відповідає вузлу автомата, кожен When<T>() — вхідною стрілкою, а послідовність викликів — виконуваним ефектам переходу. Така форма відповідає принципу «моделі як коду» [8] і скорочує когнітивну дистанцію між зображенням процесу на дошці бізнес-аналітика та його програмною реалізацією.

2.4.4 Ланцюг відповідальності як архітектурне рішення для виконання переходів

Декларативний опис саги є статичними метаданими, які рушій повинен інтерпретувати під час виконання. Проєктувальник стикається з вибором способу, яким набір активностей, оголошених для одного переходу (модифікація стану, публікація команди, реєстрація компенсації, зміна поточного стану саги), має виконуватися. У літературі патернів проєктування цю задачу традиційно розглядають у термінах ланцюга відповідальності (Chain of Responsibility) [10]: послідовність обробників, кожен з яких виконує власну частину роботи і делегує подальше виконання наступному ланці.

Альтернативні підходи теж можливі. Лінійний інтерпретатор міг би просто ітерувати список активностей і викликати кожну з них — такий варіант простіший у реалізації, але обмежує можливості введення крос-кутових поведінок (cross-cutting concerns): логування, трасування, метрик, обробки винятків. Якщо кожна з цих поведінок повинна бути присутньою в одних переходах і відсутньою в інших, лінійний інтерпретатор змушений вводити прапорці конфігурації та умовні галузки, що поступово розмиває чистоту коду.

Вибір ланцюга відповідальності розв’язує цю проблему на архітектурному рівні. Крос-кутова поведінка стає окремим елементом ланцюга, що вставляється між активностями без зміни їхньої реалізації. Логування виклику активності, наприклад, реалізується як обгорткова поведінка, що перехоплює виклик наступного елемента, логує момент початку, передає керування далі, логує момент завершення. Додавання такої поведінки до одних переходів і відсутність в інших виконується на рівні побудови ланцюга, а не на рівні коду активностей.

Другою перевагою патерна є природність композиції. Ланцюг, побудований для одного переходу, не залежить від інших ланцюгів, і їх можна компонувати паралельно, не турбуючись про спільний стан. Для системи з декларативним описом це означає, що кожна комбінація ⟨стан, тип повідомлення⟩ отримує власний незалежний ланцюг, побудований один раз під час ініціалізації і далі повторно використовуваний без додаткових витрат на

побудову. Така архітектура природно узгоджується з іммутабельністю стан-машини на рівні типу саги — побудовані ланцюги разом зі стан-машиною стають іммутабельним артефактом, що може безпечно спільно використовуватися множиною екземплярів саги.

2.4.5 Pivot Point як формалізація семантичної незворотності

Оригінальна модель саги Г. Гарсіа-Моліни та К. Салема [22] розглядає всі кроки як потенційно компенсовувані. На практиці, однак, бізнес-процеси часто містять кроки, після виконання яких повернення до попереднього стану технічними засобами неможливе: фізичне відвантаження товару з логістичного центру, надсилання електронної пошти зовнішньому отримувачу, передача коштів у платіжну систему третьої сторони з обмеженим часом відкликання. Для таких ситуацій модель компенсації потребує розширення поняттям точки семантичної незворотності.

Такий концепт присутній, у різних формулюваннях, у сучасній літературі з мікросервісних патернів. К. Річардсон [3] описує його під назвою pivot transaction і вводить поділ саги на три послідовних блоки: компенсовувані транзакції, pivot-транзакцію, повторно-виконувані транзакції. Компенсовувані транзакції можуть бути відкочені у разі збою пізнішого кроку. Pivot-транзакція є точкою переходу, після якої сага більше не відкочується. Повторно-виконувані транзакції, розташовані після pivot-точки, не мають явних компенсацій — їхня семантика полягає в тому, що сага має бути доведена до завершення, навіть якщо це потребує багаторазових повторних спроб.

SagaFlow приймає цю модель і формалізує її через окрему активність, що встановлює бінарний прапорець на екземплярі саги. Архітектурне рішення про зберігання pivot-стану саме на екземплярі, а не в зовнішньому сховищі чи як частина поточного стану автомата, має кілька обґрунтувань. По-перше, pivot-стан є логічно приналежним до конкретного екземпляра — різні екземпляри однієї саги можуть перебувати в різних pivot-фазах залежно від того, наскільки далеко кожен із них просунувся. По-друге, такий дизайн забезпечує, що будь-яка

перевірка pivot-стану є локальною і не потребує додаткових IO-операцій поза межами транзакції саги. По-третє, такий прапорець природно серіалізується разом із рештою стану екземпляра, що гарантує його збереження між перезапусками процесу.

Семантика pivot-точки вимагає уваги до її розміщення в послідовності активностей переходу. Типове місце для такої активності — між реєстрацією компенсації кроку та публікацією повідомлення, що ініціює незворотну дію. До pivot-активності компенсації залишаються зареєстрованими; після неї попередні компенсації автоматично блокуються, а подальша відповідальність за цілісність покладається або на підвищену надійність повторних спроб, або на окремий бізнес-процес ручної ескалації. Таке розташування відповідає semantic lock-принципу [3], коли pivot-точка одночасно блокує автоматичну компенсацію і сигналізує про перехід у фазу «гарантованого завершення».

2.4.6 Розділення стан-машини та екземпляра саги

Наступне проєктне рішення, що має вплив на всю архітектуру бібліотеки, стосується розрізнення двох рівнів абстракції: рівня типу саги (що описує структуру бізнес-процесу) і рівня екземпляра саги (що утримує стан конкретної активної транзакції). Такий поділ відповідає класичному розрізненню Type vs Instance у об’єктно-орієнтованому моделюванні та Entity vs Service у тактичних патернах Domain-Driven Design [8].

Тип саги описує правила: множину станів, тригери переходів, ефекти кожного переходу, кроки компенсації. Ці дані є незмінними протягом виконання програми і поділяються між усіма активними екземплярами. Екземпляр саги, навпаки, утримує лише те, що є специфічним для конкретної транзакції: поточний стан, кореляційний ідентифікатор, накопичені поля домену, список уже виконаних кроків із зареєстрованими компенсаціями та pivot-прапорець.

Такий дизайн дає кілька переваг. Споживання пам’яті мінімізується — одна стан-машина обслуговує необмежену кількість екземплярів без дублювання метаданих. Іммутабельність типу забезпечує потокобезпечність без

синхронізаційних примітивів: паралельна обробка різних екземплярів однієї саги не потребує взаємного виключення на рівні опису процесу. Нарешті, екземпляр саги стає простим даним-об’єктом, що природно серіалізується для зберігання — на відміну від імперативного підходу, коли «стан процесу» є фрагментом виконання, який важко зафіксувати у стійкому сховищі без залучення continuations або аналогічних мовних засобів.

Принципове значення має також те, що екземпляр саги ідентифікується складеним ключем: парою ⟨кореляційний ідентифікатор, тип саги⟩, а не лише кореляційним ідентифікатором. Цей вибір дозволяє одному бізнес-об’єкту одночасно брати участь у кількох паралельних сагах різних типів (наприклад, одне замовлення може мати окрему сагу для платежу й окрему для доставки), без ризику взаємного переплутування їхніх станів. Такий поділ узгоджується з принципом єдиної відповідальності на рівні агрегатів у DDD [8].

2.4.7 Маршрутизація повідомлень як задача відображення множин

На рівні архітектури системи важливою є задача маршрутизації: як для вхідного повідомлення визначити, хто повинен його обробити — одна конкретна сага, кілька паралельних саг, прямий споживач або жоден з них. Формально задача полягає у побудові відображення $\mu: T \times ID \to 2^n$, де T — тип повідомлення, ID — його кореляційний ідентифікатор, а множина значень — підмножина всіх обробників, зареєстрованих у системі.

Проєктне рішення для цього відображення має враховувати два типи поліморфізму. Перший — один тип повідомлення може оброблятися кількома сагами одночасно. Наприклад, подія PaymentProcessedEvent може бути цікавою і сазі замовлення, і сазі лояльності, і сазі нотифікацій. Жорстка прив’язка «один тип повідомлення — одна сага» унеможливила б такі сценарії. Другий — не всі повідомлення є частиною саг; частина з них адресована звичайним споживачам, що виконують атомарну локальну обробку без складного життєвого циклу.

На підставі цих міркувань у SagaFlow обрано дворівневу модель маршрутизації. На першому рівні резолвер повертає множину всіх дескрипторів

саг, які оголосили обробку заданого типу повідомлення. Якщо ця множина непорожня, кожна сага обробляє повідомлення в ізольованому контексті: окремий діапазон DI, окрема транзакція, окреме сховище даних. Така ізоляція є принциповою — збій обробки в одній сазі не повинен впливати на інші, навіть якщо вони реагують на те саме повідомлення.

На другому рівні, якщо множина саг порожня, маршрутизатор звертається до контейнера DI за реалізацією IMessageConsumer<TMessage>. Наявність такої реалізації означає, що повідомлення адресоване прямому споживачеві. Якщо й її немає, повідомлення вважається нерозпізнаним і передається у механізм дослідження помилок (зазвичай — у dead-letter-чергу транспорту).

Побудова цього відображення відбувається один раз під час ініціалізації системи, шляхом інтроспекції всіх зареєстрованих саг і читання їхніх метаданих. Такий статичний підхід виключає витрати на маршрутизацію під час обробки повідомлення, які звелися б до одного пошуку у хеш-словнику. Для систем із високим потоком повідомлень ця економія є істотною.

2.5 Управління помилками, компенсаційними операціями та відновленням цілісності

2.5.1 Теоретичні межі узгодженості в розподілених системах

Проєктування механізмів відновлення цілісності в розподіленій системі неможливе без урахування фундаментальних обмежень, сформульованих у CAP-теоремі Е. Брюера [13] та її формальному доведенні С. Гілберта і Н. Лінч [12]. Теорема стверджує, що жодна розподілена система зі спільним станом не може одночасно гарантувати узгодженість (Consistency), доступність (Availability) і стійкість до мережевих поділів (Partition Tolerance). Оскільки в реальних розподілених середовищах мережеві поділи є неминучими, вибір практично зводиться до компромісу між узгодженістю й доступністю.

Для систем, орієнтованих на бізнес-операції через кілька мікросервісів, вибір сильної узгодженості через протокол двофазного фіксування (2PC) є неприйнятним. Як детально обґрунтував П. Гелланд [19], 2PC вимагає

синхронного блокування всіх учасників на час виконання транзакції, що призводить до каскадних затримок, зменшення загальної доступності системи і неможливості горизонтального масштабування. Більше того, протокол має класичну уразливість: якщо координатор відмовляє після фази підготовки, учасники залишаються в невизначеному стані, і відновлення потребує ручного втручання [17].

Альтернативою сильній узгодженості є модель BASE (Basically Available, Soft state, Eventually consistent) як антитеза ACID [11]. У моделі BASE система гарантує базову доступність кожного сервісу незалежно, припускає тимчасову неузгодженість стану (soft state) і забезпечує узгодженість у кінцевому підсумку (eventual consistency). Сага є одним із провідних патернів реалізації BASE-семантики: кожен крок є локально-ACID, але глобальна узгодженість досягається лише після виконання всієї послідовності кроків або, у разі збою, — після виконання всіх компенсацій.

Це ставить перед проєктувальником чітке обмеження: система не може гарантувати миттєву глобальну узгодженість стану. Замість цього вона повинна забезпечити прогрес — властивість, згідно з якою будь-яка започаткована сага рано чи пізно дійде до термінального стану (успіху або компенсованого збою). Саме гарантування прогресу і є центральною задачею механізмів, що розглядаються далі.

2.5.2 Таксономія збоїв як орієнтир проєктних рішень

Раціональне проєктування механізмів відновлення вимагає чіткої класифікації сценаріїв збою, які система повинна витримувати. У літературі розподілених систем традиційно виділяють три основні категорії збоїв: припинення роботи (fail-stop) — процес повністю припиняє виконання; перехідні збої (transient) — тимчасові аномалії, які зникають самі собою при повторній спробі; та візантійські збої (Byzantine) — довільна неконсистентна поведінка, зокрема зловмисна [11].

SagaFlow проєктувався для першої й другої категорій; захист від візантійських збоїв виходить за межі області застосування бібліотеки та є предметом окремих протоколів консенсусу. Для категорії припинення роботи критичною властивістю є стійкість стану: після перезапуску процесу виконання саги повинно продовжитися з точки, на якій воно було перервано. Для категорії перехідних збоїв критичною є здатність повторити операцію без зміни її кінцевого результату, що вимагає ідемпотентності операцій.

Окрему проблему становить класифікація винятків, що виникають під час обробки повідомлення. Не всі з них є перехідними: помилка валідації вказує на принципово некоректне повідомлення, і повторна спроба її обробки не змінить результату. Помилка порушення обмеження цілісності бази даних зазвичай означає, що операція вже була виконана раніше. Натомість помилка мережевого тайм-ауту чи тимчасової недоступності зовнішнього сервісу є кандидатом на повторну спробу. Тому стратегія resilience не може бути уніфікованою — вона повинна розрізняти відновлювані й невідновлювані збої на підставі типу винятку й контексту.

На підставі цього розрізнення формується таксономія механізмів відновлення. Для fail-stop-збоїв застосовується персистування стану у стійкому сховищі та атомарне оновлення через транзакції. Для перехідних збоїв відновлюваних типів — автоматичні повторні спроби з обмеженою кількістю разів і експоненційним зростанням затримки. Для перехідних збоїв невідновлюваних типів — негайне пробросування винятку без повторів. Для бізнес-збоїв, що виникають після успішного виконання попередніх кроків саги, — компенсаційні транзакції у зворотному порядку.

2.5.3 Семантична компенсація проти синтаксичного відкату

Фундаментальною точкою розходження між розподіленими транзакціями та сагами є трактування «відкату». У ACID-моделі одиночної транзакції відкат є синтаксичним: система просто скасовує неприйняті зміни, повертаючи базу даних до стану, тотожного до стану перед початком транзакції. Така можливість

існує лише тому, що транзакційний журнал містить точну інверсію кожної зміни, і всі зміни є ізольованими до моменту фіксації.

У сазі кожен крок є окремою ACID-транзакцією, що вже зафіксувалася та стала видимою іншим процесам. Скасування такої транзакції синтаксичним відкатом неможливе — вона вже є частиною публічного стану системи. Відповідно, компенсація є семантичною: окремою транзакцією, що виконує нову дію, метою якої є нейтралізація бізнес-ефекту попередньої. Якщо перший крок зарезервував готельний номер, компенсаційним кроком є не фізичне «скасування брону», а видача команди «скасувати раніше підтверджений брон», що є повноцінною бізнес-операцією з власними побічними ефектами — наприклад, нарахуванням штрафу клієнтові за пізнє скасування.

Проєктувальник саги повинен чітко розуміти цю різницю, оскільки вона впливає на вимоги до компенсаційних операцій. По-перше, компенсація не зобов’язана відновити стан, ідентичний до стану перед кроком; достатньо, щоб результат був семантично прийнятним для домену. По-друге, компенсація сама може зазнавати збоїв, і тому вимагає власних механізмів повторних спроб. По-третє, компенсація повинна бути комутативною з подальшими діями того ж домену — тобто результат «виконати T_i, виконати C_i, виконати T_j» має бути еквівалентним результату «виконати T_j», інакше можуть виникнути патологічні стани.

У моделі SagaFlow кожна компенсація представлена не як виконавча функція, а як повідомлення, що надсилається сервісу-учаснику, який спочатку виконав основний крок. Такий дизайн зберігає автономність сервісів: сервіс готелю сам вирішує, як саме обробити команду скасування — чи це буде негайне скасування, чи відкладене підтвердження, чи ескалація до людини-оператора. Компенсаційне повідомлення містить кореляційний ідентифікатор та додаткові контекстні поля, що дозволяють сервісу ідентифікувати ресурс, який слід скомпенсувати.

2.5.4 Класифікація гарантій доставки та її практичні наслідки

У літературі систем обміну повідомленнями традиційно виділяють три рівні гарантії доставки: at-most-once, at-least-once і exactly-once [11]. Перша категорія передбачає, що повідомлення може бути доставлене нуль або один раз, але ніколи не більше. Друга — що повідомлення буде доставлене щонайменше один раз, але може дублюватися. Третя — що повідомлення буде доставлене рівно один раз.

Часте неправильне очікування з боку початківців-проєктувальників полягає в тому, що exactly-once є «найкращою» гарантією, до якої слід прагнути. Насправді, як обґрунтовано в літературі з розподілених систем, абсолютний exactly-once недосяжний у загальному випадку: для його гарантування потрібна атомарна координація між брокером повідомлень, обробником та сховищем даних, — що в розподіленій системі еквівалентно розв’язанню задачі розподіленого консенсусу і має відповідну ціну у вигляді затримок і зменшеної доступності [11].

Тому реалістичною ціллю для систем класу SagaFlow є гарантія, що в літературі позначається терміном effectively-once: повідомлення може бути фізично доставлене кілька разів, але кожна повторна доставка не має жодного ефекту на кінцевий стан системи. Ця гарантія є композицією двох: at-least-once на рівні транспорту та ідемпотентності на рівні обробника. Жодна зі складових окремо не дає потрібної властивості — at-least-once допускає повторні ефекти, а ідемпотентність без at-least-once допускає втрату повідомлень.

На цій основі проєктується комплексна модель гарантій SagaFlow. Транспортний рівень забезпечує at-least-once через механізми підтверджень доставки й повторних спроб. Рівень обробки — ідемпотентність через дедуплікацію повідомлень на підставі ключів, що обчислюються з бізнес-змісту повідомлення. Шар між ними — патерн Transactional Outbox — забезпечує атомарність публікації повідомлень відносно змін стану саги. Таким чином, система формально досягає effectively-once на рівні бізнес-семантики, визнаючи при цьому неминучість фізичних дублікатів на рівні транспорту.

2.5.5 Проблема подвійного запису та її концептуальний розв’язок

Однією з найпоширеніших архітектурних пасток у мікросервісних системах є так звана проблема подвійного запису (dual write problem) [16]. Вона виникає щоразу, коли одна бізнес-операція повинна одночасно змінити стан у власній базі даних та опублікувати подію у брокер повідомлень. За відсутності спільної транзакції між цими двома сховищами проєктувальник стикається з вибором порядку виконання, кожен із яких має власні уразливості.

Якщо спочатку фіксувати транзакцію бази, а потім публікувати повідомлення, існує ненульова ймовірність, що процес припинить роботу між двома операціями. Транзакція вже прийнята, стан змінений, але повідомлення так і не було надіслане — відповідний бізнес-процес втрачає зв’язок із рештою системи, а проблему виявляють через розбіжність стану, що може набувати тижнів. Якщо ж спочатку публікувати повідомлення, а потім фіксувати транзакцію, виникає симетрична проблема: отримувачі можуть почати реагувати на подію про зміну, яка насправді не відбулася, якщо транзакція була відкочена.

Традиційні розв’язки цієї проблеми — розподілені транзакції через 2PC або XA — виходять за межі прийнятного з міркувань продуктивності й доступності. Альтернативний концептуальний розв’язок запропоновано в літературі під назвою Transactional Outbox [16]. Сутність патерна полягає у тому, що повідомлення не публікуються безпосередньо в брокер, а записуються у спеціальну таблицю (outbox) тієї самої бази даних, у тій самій транзакції, що й основні зміни стану. Оскільки обидва записи належать до однієї транзакції, вони або обидва фіксуються, або обидва відкочуються — атомарність забезпечується локальним ACID-двигуном бази, без потреби в розподіленій координації.

Подальша доставка повідомлень з outbox у брокер виконується асинхронним фоновим процесом, який періодично зчитує непроцесовані записи і передає їх до транспортного рівня. У такій конструкції виникає тонкий момент: процес доставки може зазнати збою після успішної доставки повідомлення, але до оновлення його статусу в outbox, що призведе до повторної доставки під час наступного проходу. Такий сценарій є концептуально прийнятним: система

свідомо переходить від гарантії exactly-once до at-least-once на рівні транспорту, покладаючись на ідемпотентність обробки для нейтралізації дублікатів.

Таке проєктне рішення відображає характерний для розподілених систем компроміс: замість того, щоб намагатися досягти ідеальної гарантії на одному рівні, архітектура розкладає задачу на кілька рівнів із різними гарантіями, композиція яких дає прийнятний результат. Такий стилістичний підхід С. Ньюман [2] називає «мисленням у гарантіях» і рекомендує як базовий спосіб аналізу будь-якого мікросервісного рішення.

2.5.6 Моделі ідемпотентності та вибір ключа

Ідемпотентність як математичну властивість визначають як f(f(x)) = f(x) — повторне застосування функції не змінює результату. У контексті обробки повідомлень це означає, що повторне отримання того самого повідомлення не повинно призводити до повторного виконання побічних ефектів.

Проєктувальник має кілька варіантів досягнення цієї властивості. Перший — природна ідемпотентність операції: деякі операції ідемпотентні за побудовою, наприклад, встановлення значення поля в заданий стан. Повторне виконання такої операції не змінює результату. Другий — умовна ідемпотентність: операція стає ідемпотентною за умови додаткової перевірки стану перед її виконанням, як-от «створити запис, якщо його ще не існує». Третій — штучна ідемпотентність через дедуплікацію: система явно запам’ятовує, які повідомлення вже оброблено, і ігнорує повторні надходження.

У загальному випадку не можна покладатися на природну або умовну ідемпотентність, оскільки вона є властивістю конкретної операції й часто порушується вже при дрібних змінах бізнес-логіки. Тому SagaFlow обирає третій варіант — штучну дедуплікацію на основі ідентифікаційного ключа. Таке рішення вимагає розв’язання двох підзадач: вибору способу обчислення ключа й вибору способу зберігання множини вже оброблених ключів.

Для способу обчислення ключа теоретично можливі кілька підходів. Найпростіший — використати власний ідентифікатор повідомлення. Такий

підхід працює лише за умови, що ідентифікатори є стабільними між повторними доставками того самого повідомлення — що не завжди гарантовано, особливо якщо повідомлення генерується клієнтом з обмеженими можливостями. Альтернатива — обчислення ключа як детермінованої функції від бізнес-змісту повідомлення, що забезпечує однаковий ключ для двох повідомлень із тотожним бізнес-змістом, незалежно від обставин їх генерації.

SagaFlow обирає другий підхід, реалізований через криптографічну хеш-функцію, застосовану до властивостей повідомлення, оголошених як частина ключа. Вибір криптографічного хешу замість простіших функцій (таких як MurmurHash чи CRC-32) обґрунтований вимогою до низької ймовірності колізій: у фінансових і транзакційних доменах колізія ключа може призвести до некоректного замовчування справжньої операції як дубліката. Хоча криптографічний хеш є обчислювально дорожчим за некриптографічні, його продуктивність залишається на рівні сотень мегабайт на секунду на сучасному обладнанні, що на кілька порядків перевищує типову частоту повідомлень у бізнес-системах [11].

На концептуальному рівні ідемпотентність у SagaFlow діє у двох точках: під час публікації повідомлення (перевірка, чи не було таке саме повідомлення вже опубліковане) і під час прийняття повідомлення споживачем (перевірка, чи не було воно вже оброблене). Така дворівнева модель є свідомою надмірністю: вона захищає від дублікатів, що виникають як на стороні відправника (через повторний запуск обробника), так і на стороні отримувача (через повторну доставку від брокера).

2.5.7 Моделі конкурентного доступу та їхні межі застосування

Одночасне надходження кількох повідомлень для одного й того самого екземпляра саги створює класичну проблему конкурентного доступу: втрати оновлень (lost update), брудного читання (dirty read), несеріалізовних переплетень. У літературі баз даних розрізняють два фундаментально відмінних підходи до її розв’язання: песимістичний і оптимістичний [11].

Песимістична модель припускає, що конфлікти є частими, і проактивно блокує ресурс на час виконання операції. У реляційних базах даних це реалізується через явне блокування рядка. Конкурентні транзакції, що намагаються отримати доступ до того самого рядка, блокуються до моменту звільнення блокування. Перевагою такої моделі є детерміністичність: конфліктів не виникає за побудовою, а затримки на блокуванні є передбачуваними. Недоліком є зниження пропускної здатності при паралельному навантаженні та ризик дедлоку при неуважному проєктуванні порядку блокування ресурсів.

Оптимістична модель, навпаки, припускає, що конфлікти є рідкісними, і не блокує ресурс під час читання. Натомість вона фіксує на момент читання версію ресурсу (зазвичай як цілочисельний токен або позначку часу) і перевіряє під час запису, чи не змінилася ця версія. Якщо змінилася, операція скасовується з виявленим конфліктом, який далі обробляється на прикладному рівні — типово через повторну спробу з оновленим станом. Перевагою моделі є висока пропускна здатність за низької конкуренції; недоліком — необхідність зайвої роботи у випадку частих конфліктів, коли кожна операція має шанс потребувати повторного виконання.

Вибір між моделями в реальній системі не є універсальним і залежить від профілю навантаження. Для саги з низькою частотою повідомлень на один екземпляр оптимістична модель дає мінімум затримок і достатню коректність. Для саги з високою частотою (наприклад, коли кілька паралельних джерел одночасно надсилають події для того самого бізнес-об’єкта) песимістична модель уникає лавинного повторення та гарантує прогрес.

Проєктне рішення SagaFlow полягає у наданні обох моделей як альтернатив, що декларативно обираються на рівні типу саги. За замовчуванням обрано оптимістичну модель як менш обмежувальну, з чітким очікуванням, що переведення саги в песимістичний режим є свідомим рішенням, яке приймається на підставі вимірюваного профілю навантаження. Така диференціація узгоджується з принципом відсутності передчасної оптимізації: зайві

блокування не виникають у звичайних сценаріях, але залишаються доступними для специфічних випадків.

Окремою тонкою є вибір рівня ізоляції транзакції, у межах якої виконуються операції саги. Поширеною спокусою є використання рівня RepeatableRead або вищого, що, здавалося б, природно гарантує стабільність читання. Однак у PostgreSQL, який ґрунтується на моделі MVCC, рівень RepeatableRead реалізується через снепшоти, і при конкурентній зміні запису активна транзакція отримує не блокування, а виняток серіалізації. Така поведінка перетворює очікуване блокування на явну помилку, що вимагає окремої обробки на рівні застосунку. Обраний рівень ReadCommitted поводиться відповідно до інтуїтивних очікувань: явне блокування справді призводить до чекання, що узгоджується з семантикою песимістичного режиму.

2.5.8 Резилентність як системна властивість

Крім відновлення від окремих збоїв, проєктувальник повинен враховувати резилентність як системну властивість — здатність системи зберігати працездатність при частковій відмові її компонентів. У літературі інженерії надійних систем цю властивість розкладають на кілька складових патернів: повторні спроби (retry), таймаути (timeout), обмеження навантаження (bulkhead) і розмикання ланцюга (circuit breaker) [9].

Патерн повторних спроб є найочевиднішим, але його некоректна реалізація може погіршити ситуацію замість її поліпшення. Наївна реалізація — негайна повторна спроба без затримки — при збої зовнішнього сервісу призводить до лавиноподібного зростання навантаження на вже перенавантажений сервіс, поглиблюючи його проблеми. Для запобігання цьому ефектові патерн доповнюється експоненційним зростанням затримки між спробами (exponential backoff), при якому кожна наступна спроба відкладається на вдвічі більший інтервал.

Експоненційний backoff сам по собі вразливий до ефекту синхронізованого повторного виклику, коли кілька екземплярів одночасно намагаються здійснити

повторну спробу через однаковий час після спільного збою. Розв’язанням цієї проблеми є додавання джиттера: випадкового відхилення затримки в межах певного проценту від розрахованого значення. Джиттер десинхронізує повторні виклики і розподіляє навантаження в часі, згладжуючи його профіль.

Патерн таймауту обмежує максимальний час очікування відповіді від операції. Без таймауту повільна або заблокована операція може утримувати ресурси (потоки, з’єднання, пам’ять) невизначено довго, поширюючи свій вплив на інші компоненти системи. Ключовим проєктним питанням є вибір адекватного значення таймауту: надто мале призводить до хибно-позитивних переривань швидких, але затриманих операцій; надто велике — нівелює саму мету патерну.

Патерн розмикання ланцюга (circuit breaker) додає ще один рівень захисту. Якщо кількість послідовних збоїв у виклику певного зовнішнього ресурсу перевищує встановлений поріг, патерн переводиться у стан «розімкнутого», у якому всі подальші виклики цього ресурсу відхиляються одразу, без фактичної спроби. Такий стан утримується протягом заданого періоду, після чого система переходить у стан «напіврозімкнутого» — пропускає один пробний виклик, на підставі якого вирішує повертатися до нормального режиму чи залишатися в режимі відмови. Патерн запобігає каскадним збоям, коли відмова одного компоненту провокує послідовний збій усіх, що від нього залежать.

SagaFlow застосовує комбінацію цих патернів через декларативну конфігурацію на рівні типу споживача. Такий дизайн забезпечує, що резилентність є частиною контракту обробника, а не окремою наскрізною темою, яку доводиться реалізовувати поза бізнес-логікою.

2.5.9 Інтегрована модель гарантій системи

Розглянуті механізми — компенсації, pivot-точки, ідемпотентність, outbox, транзакційне управління, політики повторів — не є незалежними. Вони утворюють багатошарову модель, у якій кожен шар розв’язує власний клас задач і покладається на гарантії нижчих шарів для побудови власних.

На нижньому рівні транспорт забезпечує гарантію at-least-once доставки. На середньому рівні outbox забезпечує атомарність публікації відносно змін стану, а ідемпотентність — нейтралізацію дублікатів, що виникають від at-least-once. На верхньому рівні автомат саги забезпечує узгодженість багатокрокового бізнес-процесу через послідовні переходи та компенсації.

Принциповим для такої багатошарової моделі є властивість неперетинання зон відповідальності. Транспорт не турбується про бізнес-семантику; він лише гарантує, що повідомлення рано чи пізно буде доставлене. Шар outbox не турбується про повторні доставки; він лише гарантує, що те, що було обіцяне до публікації, справді буде опубліковане. Шар ідемпотентності не турбується про порядок публікацій; він лише гарантує, що кожне повідомлення буде оброблене щонайбільше один раз. Шар саги не турбується про ідемпотентність окремих повідомлень; він оперує у припущенні, що кожне отримане повідомлення є унікальною подією.

Таке розділення відповідальностей відповідає принципу розділення інтересів і дає змогу змінювати або замінювати окремі шари без впливу на інші. Заміна транспорту з RabbitMQ на Kafka, наприклад, потребує лише нової реалізації адаптера транспортного шару; решта механізмів продовжує працювати без змін, оскільки їхня поведінка не залежить від конкретного транспорту, а лише від властивості at-least-once, яку обидва транспорти гарантують.

Спільним результатом композиції цих шарів є властивість, що може бути охарактеризована як «прагматична цілісність». Вона не дорівнює сильній узгодженості розподілених транзакцій, але наближається до неї за практичними характеристиками: будь-яка зафіксована бізнес-операція або доходить до повного завершення, або відкочується через компенсації до стану, семантично еквівалентного початковому. Проміжні неузгодженості тривають обмежений час і не призводять до стійких втрат даних. Для більшості доменів такі характеристики є прийнятним компромісом між досяжною цілісністю та реалістичними витратами на координацію в розподіленому середовищі.

Проєктне рішення є свідомим відображенням філософії, сформульованої П. Гелландом [19]: у розподіленій системі не буває магії — є лише чіткі гарантії, зручно скомпоновані в зрозумілі людині абстракції. Архітектура SagaFlow слідує цій філософії, пропонуючи не одну «срібну кулю», а набір скоординованих примітивів, композиція яких дає інженеру передбачувану поведінку системи у всьому діапазоні очікуваних та несподіваних сценаріїв.

Висновки до розділу 2

У розділі система SagaFlow окреслена на проєктному рівні. Сформульовано функціональні та нефункціональні вимоги, обрано архітектуру «портів і адаптерів» з виокремленим ядром координації, описано стек технологій (.NET, EF Core, RabbitMQ через MassTransit). Принципові проєктні рішення — декларативний DSL поверх Fluent API, ланцюг відповідальності для виконання переходів, поділ типу й екземпляра саги, явна формалізація точки незворотності — зведено в єдину модель.

Окрему увагу приділено механізмам відновлення цілісності. Семантичну компенсацію відмежовано від синтаксичного відкату, ідемпотентність винесено на рівень обов’язкового контракту обробників, розглянуто компроміси оптимістичного та песимістичного контролю конкурентного доступу. Спільний результат — набір примітивів, з яких користувач бібліотеки збирає процес із передбачуваною поведінкою у штатних і збійних сценаріях.

У третьому розділі ці рішення реалізовано в коді й експериментально перевірено їхню коректність та продуктивність.

РОЗДІЛ 3. РЕАЛІЗАЦІЯ ТА ТЕСТУВАННЯ СИСТЕМИ

3.1 Опис компонентів та їх взаємодії

У підрозділі 2.2 запропоновано архітектурну модель системи SagaFlow, побудовану за принципом «портів і адаптерів» А. Кокберна [7]. У цьому підрозділі описано конкретне втілення цієї моделі: фізичний поділ кодової бази, склад центральних компонентів, що беруть участь в обробці повідомлення, та механізм їхньої координації в межах одного циклу обробки. Виклад побудовано від фізичної структури проєкту до логіки взаємодії в часі — тобто від статичного зрізу коду до динамічної моделі його виконання.

3.1.1 Фізична структура коду й композиція пакетів

Кодову базу SagaFlow організовано як solution із п’яти NuGet-пакетів, доповнений набором тестових проєктів. Розподіл коду між пакетами підпорядкований принципу мінімальних транзитивних залежностей, сформульованому у вимогах: ядро системи не повинно тягнути за собою залежностей від конкретної інфраструктури, а кожен адаптер — лише ту бібліотеку, яка є необхідною для роботи з відповідною технологією. Композицію пакетів та їхні взаємні залежності зображено на рисунку 3.1.

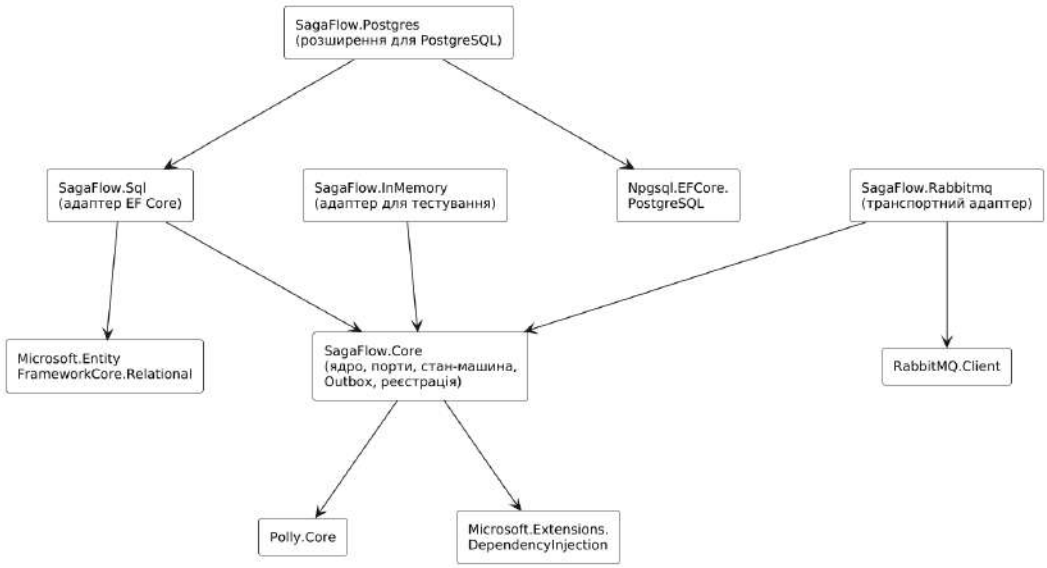

Рисунок 3.1 – Структура NuGet-пакетів SagaFlow та їхні залежності

Як видно з рисунка 3.1, пакет SagaFlow.Core залежить лише від платформних бібліотек Polly.Core та Microsoft.Extensions.DependencyInjection, не маючи у своєму графі ні драйверів СКБД, ні клієнтських бібліотек брокерів повідомлень. Адаптерні пакети формують відокремлені гілки графа залежностей, кожна з яких підключається користувачем за потреби. Така організація відповідає принципу інверсії залежностей, що формулюється як вимога будувати програмні модулі високого рівня незалежно від модулів низького рівня — у мікросервісному контексті це означає, що бізнес-логіка саг не залежить від драйверів і транспортів, а обидві категорії компонентів залежать від абстракцій, винесених у ядро.

Усередині кожного пакета застосовано однотипну функціональну декомпозицію за директоріями. Зокрема, у SagaFlow.Core виокремлено простори імен StateMachine, Messages, Outbox, Consumers, Transactions та Registration; кожен із них містить лише ті типи, що належать до відповідної інженерної турботи. Аналогічна структура повторюється в адаптерних пакетах. Така регулярність організації коду полегшує орієнтацію в проєкті: розробникові, обізнаному зі структурою ядра, відомо, де шукати конкретний клас в адаптері.

3.1.2 Конвеєр обробки одного повідомлення

Динамічна модель функціонування SagaFlow зводиться до обробки одного повідомлення в межах окремої області видимості (scope) контейнера впровадження залежностей. Послідовність взаємодії компонентів зображено на рисунку 3.2.

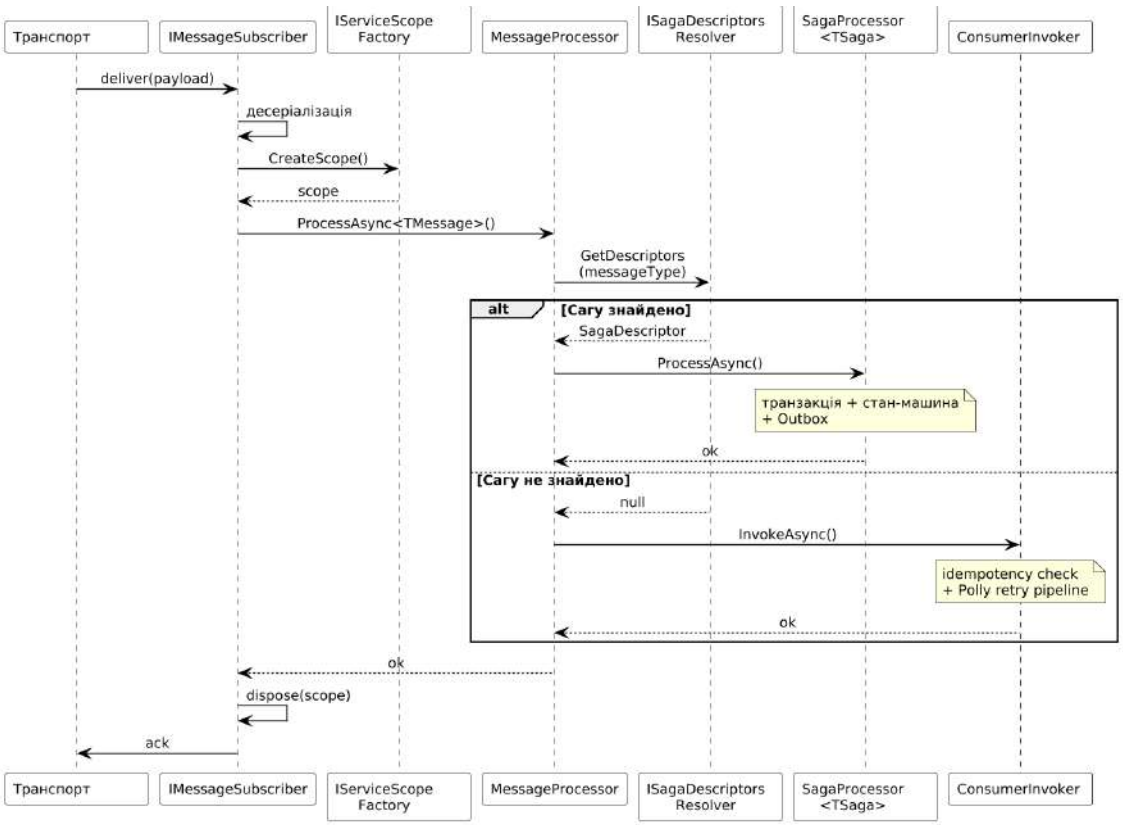

Рисунок 3.2 – Послідовність обробки одного повідомлення в SagaFlow

Точкою входу є реалізація порту IMessageSubscriber (RabbitMqSubscriber для зовнішнього транспорту або InMemorySubscriber для внутрішньопроцесного), що отримує сире повідомлення з транспорту та виконує його десеріалізацію у CLR-об’єкт. Перед делегуванням обробки підписник створює нову область видимості через виклик IServiceScopeFactory.CreateScope(). Це проєктне рішення є визначальним для коректності всієї подальшої обробки: воно гарантує, що всі залежності, задіяні при обробці одного повідомлення (контекст бази даних, репозиторій саги, сховище outbox, менеджер транзакцій), створюються в межах цієї області видимості та знищуються разом із нею. За відсутності такого обмеження запис у сховище outbox і зміна стану саги не змогли б ділити одну транзакцію — а отже, один екземпляр DbContext, — і атомарність патерну Transactional Outbox [3; 16] була б порушена.

Усередині нової області видимості підписник звертається до IMessageProcessor — основного контракту маршрутизації, реалізованого класом

MessageProcessor. Логіка методу ProcessAsync<TMessage>() зводиться до короткого діалогу з ISagaDescriptorsResolver: процесор запитує перелік дескрипторів саг, здатних обробити повідомлення цього типу. У разі знайденого дескриптора керування передається відповідному SagaProcessor<TSaga> — компоненту, що інкапсулює повний життєвий цикл обробки саги. У разі відсутності дескрипторів MessageProcessor шукає в DI-контейнері прямого споживача IMessageConsumer<TMessage> і делегує обробку в IConsumerInvoker. Один і той самий тип повідомлення може бути зареєстрований одночасно і як такий, що обробляється сагою, і як такий, що споживається прямим обробником; у такому разі обидва шляхи виконуються незалежно й послідовно.

Клас ConsumerInvoker відповідає за прозоре застосування політик стійкості до прямих споживачів. Перед викликом IMessageConsumer.Consume() ConsumerInvoker звертається до IIdempotencyStore для перевірки факту попередньої обробки повідомлення з обчисленим ідемпотентним ключем; у разі повторного надходження обробку припиняють без побічних ефектів. Сам виклик обгорнуто Polly-конвеєром ResiliencePipeline, побудованим з атрибута [HandlerRetryPolicy], оголошеного на класі споживача. Якщо споживач реалізує також інтерфейс ICompensableMessageConsumer<TMessage>, ConsumerInvoker викликає метод Compensate() перед повторним пробросом винятку, чим уможливлюється локальна логіка відкату для тих ефектів, що були встигнуті до моменту збою.

3.1.3 Сховище стану саг

Стан кожного активного екземпляра саги зберігається в єдиній таблиці SagaFlowSagaInstances, відображеній на сутність SagaInstanceEntity. Її структуру свідомо обрано мінімалістичною. Поля CorrelationId і SagaType утворюють складений первинний ключ; поле Version слугує токеном оптимістичного контролю конкурентності в EF Core (атрибут IsConcurrencyToken); саме тіло екземпляра саги серіалізується в полі Data як JSON-документ. Такий вибір зумовлений двома міркуваннями. По-перше, схема саги визначається

користувачем бібліотеки і не може бути відомою наперед на рівні ядра — нав’язування конкретного реляційного розкладу полів обмежило б застосовність SagaFlow. По-друге, JSON-серіалізація з опцією PreferredObjectCreationHandling.Populate уможливлює безпечну еволюцію схеми саги між версіями застосунку: додавання нового поля з типовим значенням не призводить до помилки десеріалізації старих записів. Цей підхід відповідає поширеній у літературі рекомендації щодо schema-on-read для довгоживучих даних [11].

Доступ до сутностей опосередкований інтерфейсом ISagaRepository<TSaga>, реалізованим у двох варіантах. Реалізація InMemorySagaRepository<TSaga> зберігає екземпляри в ConcurrentDictionary і призначена для модульного й інтеграційного тестування без зовнішніх залежностей. Реалізація SqlSagaRepository<TSaga> побудована над SagaFlowDbContext і виконує пошук, збереження та видалення екземплярів саги засобами EF Core. Між двома реалізаціями є одна суттєва різниця — підтримка SQL-варіантом обох режимів конкурентного доступу. У песимістичному режимі метод FindByCorrelationIdAsync додатково виконує запит SELECT 1 FROM SagaFlowSagaInstances WHERE … FOR UPDATE, що накладає блокування на рівні рядка СКБД на весь час транзакції. В оптимістичному режимі такого блокування немає; натомість при виклику SaveAsync EF Core інкрементує Version, і у разі попередньої зміни рядка іншою транзакцією виникає DbUpdateConcurrencyException, який ядро перетворює на SagaConcurrencyException для подальшої обробки.

3.1.4 Підсистема Outbox

Підсистема Outbox реалізує патерн Transactional Outbox у формулюванні К. Річардсона [3] і Б. Ібріяма [18] та розв’язує проблему подвійного запису, висвітлену в підрозділі 1.3. Її функціонування зведено до двох сценаріїв: атомарного запису вихідного повідомлення в outbox-таблицю одночасно зі

зміною стану саги та асинхронної доставки цих повідомлень у транспортний рівень. Структуру підсистеми та потоки даних зображено на рисунку 3.3.

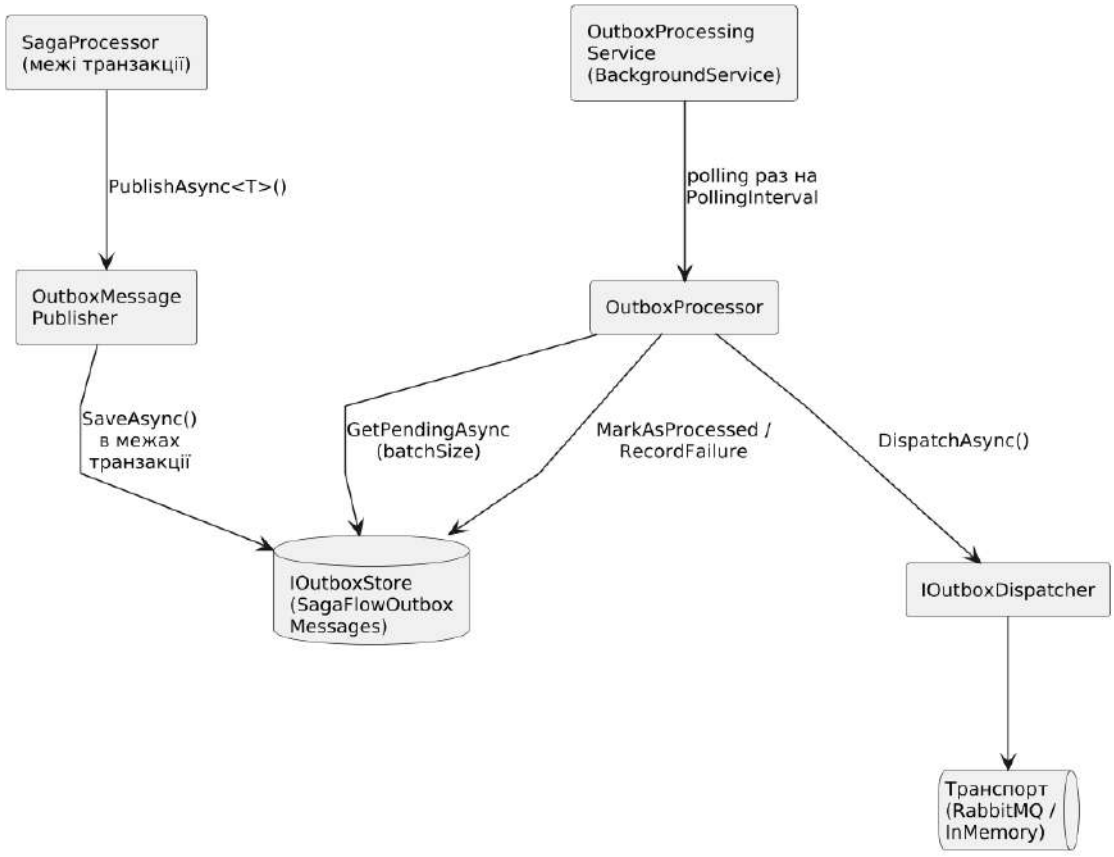

Рисунок 3.3 – Структура підсистеми Transactional Outbox

Запис вихідного повідомлення виконується класом OutboxMessagePublisher, що реалізує порт IMessagePublisher. Виклик методу PublishAsync<TMessage>() з поведінкового конвеєра саги формує сутність OutboxMessageEntity з такими полями. Поле Id отримує значення викликом Guid.CreateVersion7(), що гарантує монотонне зростання ідентифікаторів і впорядкований обхід outbox під час доставки. Поле MessageType зберігається у формі AssemblyQualifiedName для коректного відновлення CLR-типу під час десеріалізації. Поле Payload містить серіалізоване тіло повідомлення у форматі JSON. Полю Status присвоюється значення Pending. У разі наявності у повідомленні властивостей з атрибутом [IdempotencyKeyPart] OutboxMessagePublisher обчислює ідемпотентний ключ як SHA-256 хеш від конкатенації цих властивостей і перевіряє його наявність у

IOutboxStore.ExistsByIdempotencyKeyAsync(); за збігу повторне записування пропускається. Виклик методу IOutboxStore.SaveAsync() виконується в межах тієї самої транзакції, що й зміна стану саги, оскільки обидва компоненти спільно використовують один екземпляр DbContext через scoped DI.

Доставку забезпечує пара класів — OutboxProcessingService та OutboxProcessor. Перший зареєстрований як BackgroundService у середовищі Microsoft.Extensions.Hosting і періодично, з інтервалом, заданим у конфігурації, викликає процесор. Другий зчитує батч pending-повідомлень через IOutboxStore.GetPendingAsync(batchSize), відновлює CLR-тип за MessageType, десеріалізує тіло повідомлення і передає його до IOutboxDispatcher.DispatchAsync() — порту, реалізованого RabbitMqOutboxDispatcher для зовнішнього транспорту або InMemoryOutboxDispatcher для внутрішньопроцесного. У разі успіху повідомленню встановлюється Status = Processed; у разі винятку інкрементується RetryCount, а після перевищення MaxRetries запис переходить у термінальний стан Failed. Описана схема свідомо переходить від exactly-once на рівні фізичної доставки до at-least-once: у разі збою процесу після успішної доставки, але до оновлення статусу в outbox, повідомлення буде надіслано повторно під час наступного проходу. Цей дублікат поглинається ідемпотентністю споживача — другим рівнем дедуплікації, реалізованим у ConsumerInvoker. Така композиція рівнів відповідає принципу «мислення в гарантіях», описаному С. Ньюманом [2]: замість того щоб намагатися досягти ідеальної гарантії на одному рівні, архітектура розкладає задачу на кілька рівнів з різними гарантіями, композиція яких дає прийнятний результат.

3.1.5 Інтеграція з контейнером впровадження залежностей

Конфігурація системи здійснюється через клас SagaFlowConfigurator, який під час виклику методу AddSagaFlow(...) приймає делегат користувача та послідовно реєструє у стандартному IServiceCollection усі необхідні сервіси. Лайфтайми зареєстрованих компонентів обрано з урахуванням природи кожного

з них. Тип SagaStateMachine<TSaga> реєструється як singleton, оскільки після виклику конструктора стан-машина залишається незмінною, і зберігання її в одному екземплярі економить пам’ять і час повторної ініціалізації. Натомість типи SagaProcessor<TSaga>, SagaExecutor<TSaga>, MessageProcessor, ISagaRepository<TSaga> та IOutboxStore оголошені як scoped — лише в такому режимі виконується інваріант про спільний DbContext і спільну транзакцію в межах однієї обробки повідомлення. У разі непідключення жодного інфраструктурного провайдера з реалізацією ITransactionManager (типова ситуація при роботі лише в режимі InMemory) система реєструє NoOpTransactionManager як запасний варіант через TryAddSingleton. Завдяки цьому ядро не падає з помилкою про відсутність обов’язкового сервісу, проте водночас не створює у користувача неправдивого враження про наявність справжніх розподілених транзакцій.

Кожна сага реєструється викликом configurator.AddSaga<TStateMachine, TInstance>(). Цей метод не лише додає тип до DI, а й виконує побічну дію — ініціалізує дескриптор саги через SagaDescriptorsResolver. У ході побудови дескриптора відбувається рефлексивний обхід стан-машини, у результаті якого збираються типи повідомлень-ініціаторів, типи повідомлень, що обробляються в проміжних станах, і типи компенсаційних повідомлень. На підставі цих переліків MessageProcessor згодом приймає рішення про маршрутизацію.

3.2 Реалізація координації транзакцій за допомогою Saga

3.2.1 Обґрунтування проєктних рішень

У підрозділі 1.4 розглянуто концепцію саги як послідовності локальних транзакцій T_1, T_2, …, T_n із компенсаційними транзакціями C_1, C_2, …, C_n, та сформульовано вибір на користь оркестраційної моделі координації за К. Річардсоном [3]. У цьому підрозділі описано конкретний механізм реалізації такої моделі в SagaFlow. Реалізація стан-машини спирається на три проєктні рішення, що працюють спільно: рефлексивне виявлення станів у конструкторі базового класу, Fluent API для декларативного опису переходів та поведінковий

конвеєр на основі патерну «Ланцюжок відповідальності» (Chain of Responsibility), у якому кожен крок саги реалізовано як окрему ланку. Названі рішення взаємопов’язані: кожне з них окремо було б недостатнім, але їхня композиція забезпечує одночасно компактність прикладного коду, розширюваність ядра та можливість незалежного тестування кожного кроку саги.

3.2.2 Базовий клас і виявлення станів

Конструктор SagaStateMachine<TSaga> виконує дві дії, що готують стан-машину до роботи. По-перше, ініціалізуються чотири службові стани з семантично визначеним призначенням: стан Initial маркує екземпляр саги, що ще не отримав жодного повідомлення; стан Completed слугує сигналом до видалення екземпляра з репозиторію після успішного завершення; стан Failed зберігає сагу для діагностики після виконання компенсацій; стан Final є допоміжним для нестандартних сценаріїв, що не вписуються в класичний шаблон. По-друге, конструктор виконує рефлексивний обхід усіх публічних властивостей похідного класу типу State і автоматично створює для кожної з них об’єкт-стан з іменем, що збігається з іменем властивості. Така розв’язка позбавляє користувача обов’язку явно ініціалізовувати кожне поле — достатньо оголосити властивість, і базовий клас сам присвоїть їй коректний об’єкт під час побудови. Імена станів зберігаються у внутрішній таблиці й використовуються при серіалізації екземпляра саги: текстове ім’я стану записується в JSON разом з іншими полями, що робить дамп збережених саг придатним для людського читання й полегшує діагностику в продакшні.

3.2.3 Fluent API і побудова переходів

Опис переходів між станами виконується в конструкторі похідного класу через ланцюжок викликів. Точкою входу є метод Initially(...) (для повідомлень, що ініціюють нову сагу) або During(state, ...) (для повідомлень, що обробляються у проміжному стані). Усередину передається конфігурація When<TMessage>(),

яка асоціює конкретний CLR-тип повідомлення з послідовністю activity-операцій. Приклад опису одного переходу наведено нижче:

```
During(ReservingHotel,
    When<HotelReservedEvent>()
        .Then(ctx => ctx.Instance.HotelConfirmationCode =
            ctx.MessageContext.Message.ConfirmationCode)
        .WithCompensation(i => new
CancelHotelCommand(i.CorrelationId))
        .Publish(ctx => new BookFlightCommand(
            ctx.Instance.CorrelationId,
            ctx.Instance.TripId))
        .TransitionTo(BookingFlight));
```

Описана конструкція не виконується щоразу при отриманні повідомлення. Виклик When<T> створює об’єкт-білдер, на якому ланцюжкові методи накопичують перелік activity, а завершальний метод (TransitionTo або просто кінець ланцюжка) реєструє побудований опис переходу у внутрішній таблиці стан-машини. Уся ця робота відбувається одноразово у конструкторі; її результати стають доступними всім обробкам повідомлень, що далі проходять через цю стан-машину. Завдяки singleton-режиму DI-реєстрації стан-машини один і той самий побудований конвеєр використовується для тисяч паралельних обробок без повторної ініціалізації, що позитивно впливає на продуктивність системи.

Така конструкція є реалізацією патерну плавного інтерфейсу (Fluent Interface): кожен метод повертає той самий або сумісний за типом об’єкт-білдер, утворюючи синтаксично читабельне DSL. Перевагою такого підходу є те, що повний потік виконання конкретного бізнес-сценарію читається в одному місці — в описі стан-машини, — і не потребує реконструкції з розрізнених подій у різних сервісах [3].

3.2.4 Activities як ланки поведінкового конвеєра

Кожна activity, оголошена в декларативному описі переходу, під час побудови переходу обгортається в поведінкову ланку — реалізацію інтерфейсу IBehavior<TSaga, TMessage>. Інтерфейс має один метод

ExecuteAsync(BehaviorContext<TSaga, TMessage> context, IBehavior<TSaga, TMessage> next), який приймає контекст обробки та посилання на наступну ланку конвеєра. Класична реалізація патерну «Ланцюжок відповідальності» передбачає, що ланка виконує власну дію, після чого делегує керування виклику next.ExecuteAsync(...). Завершує ланцюжок термінатор LastBehavior, що не виконує жодної дії та не має наступної ланки.

Перелік основних activity, відповідні поведінкові класи та їхній ефект на стан саги наведено у таблиці 3.1.

Таблиця 3.1 – Activity SagaFlow та їхні поведінкові реалізації

Activity у Fluent API	**Поведінковий клас**	**Ефект на стан саги**
Then(action)	ThenActivityBehavior	Виконує синхронну або асинхронну зміну полів екземпляра саги
Publish(factory)	PublishActivityBehavior	Будує вихідне повідомлення та передає його в IMessagePublisher (Outbox)
WithCompensation(factory)	CompensationRecorderActivity	Реєструє фабрику компенсації в стан-машині та додає індекс кроку до Instance.CompletedSteps
PivotPoint()	PivotActivity	Встановлює прапор Instance.IsPivotReached у true
Compensate()	CompensateActivity	Запускає послідовне виконання компенсацій у зворотному порядку
TransitionTo(state)	TransitionToActivityBehavior	Змінює Instance.CurrentState на цільовий стан

Конкретна послідовність ланок визначається порядком, у якому користувач викликав activity у Fluent API. Така архітектура є відкритою для розширення — додавання нової activity не потребує змін у ядрі стан-машини, достатньо реалізувати інтерфейс IBehavior<TSaga, TMessage> і метод-розширення для білдера. Описане рішення відповідає принципу відкритості/закритості з SOLID-принципів: ядро є закритим для модифікації, але відкритим для розширення.

3.2.5 Реалізація компенсаційного механізму

Механізм компенсацій спирається на дві activity, що працюють у парі. Перша — CompensationRecorderActivity — викликається кожного разу при використанні в Fluent API виразу WithCompensation<T>(factory). Activity не виконує жодних побічних дій у транспорті: вона додає до списку Instance.CompletedSteps ціле число — індекс зареєстрованої фабрики компенсації у внутрішньому масиві стан-машини. Самі фабрики зберігаються на рівні стан-машини як singleton-об’єкти; екземпляр саги несе лише індекси тих кроків, що були успішно виконані. Така розв’язка має дві переваги. По-перше, екземпляр саги залишається легким і добре серіалізується. По-друге, не виникає проблеми із серіалізацією делегатів — фабрики ніколи не покидають пам’яті процесу.

Друга activity — CompensateActivity — викликається при отриманні повідомлення про невдачу одного з кроків. У разі рівності Instance.IsPivotReached значенню true activity припиняє роботу й логує попередження: бізнес-логіка вже досягла точки, відкат якої вважається технічно неможливим (наприклад, моменту після авторизації списання коштів зовнішнім платіжним провайдером). У протилежному випадку activity ітерує CompletedSteps у зворотному порядку, для кожного індексу витягує відповідну фабрику зі стан-машини, формує конкретне компенсаційне повідомлення та публікує його через IMessagePublisher. У реалізації цієї логіки є технічна деталь, яку легко прогавити при першому читанні коду: кожне компенсаційне повідомлення публікується через виклик PublishAsync<T>, де параметр T є CLR-типом конкретного повідомлення, а не загальним інтерфейсом IMessage. У разі публікації під типом IMessage серіалізатор System.Text.Json зберіг би в outbox тип IMessage як MessageType, і десеріалізатор не зміг би відновити конкретний клас. Тому всередині CompensateActivity використано рефлексивний виклик через MakeGenericMethod: для кожної фабрики формується конкретний generic-метод, що викликається через MethodInfo.Invoke. Після публікації всіх

компенсацій список CompletedSteps очищується, а сама сага переводиться в стан Failed для подальшої діагностики.

Описана реалізація відповідає семантиці компенсацій, сформульованій Г. Гарсією-Моліною та К. Салемом ще у роботі 1987 року [22]: компенсація не зобов'язана відновити стан, ідентичний до стану перед кроком, — достатньо, щоб результат був семантично прийнятним для бізнес-домену. SagaFlow виконує саме семантичну компенсацію, делегуючи власне бізнес-логіку відкату прикладному коду користувача, а на себе беручи лише гарантоване впорядковане виконання компенсацій у зворотному порядку.

3.2.6 Pivot Point як спеціальний випадок

Activity PivotActivity має мінімалістичну реалізацію — вона встановлює прапор IsPivotReached = true на екземплярі саги, — однак її семантичне навантаження є значним. Точка неповернення (pivot point) розмежовує дві фази в життєвому циклі саги: до її досягнення будь-яка невдача провокує автоматичне виконання компенсацій; після — компенсації пропускаються, а сама сага вважається «прийнятою» в тому сенсі, що зворотний відкат уже не може бути виконаний автоматичними засобами. Прапор IsPivotReached зберігається у складі екземпляра саги та серіалізується разом з ним; це уможливлює коректну поведінку навіть при перезапуску процесу між фазами «до» та «після» pivot point. Концепція точки неповернення є важливим практичним розширенням базової моделі саги, адже у багатьох бізнес-сценаріях не всі кроки є автоматично відкатними — типовим прикладом є відправлення фізичного товару клієнту, повернення якого є окремим бізнес-процесом і не належить до області відповідальності координатора саги.

3.2.7 Повний цикл обробки в SagaProcessor

Усі описані механізми координуються класом SagaProcessor<TSaga>. Метод ProcessAsync цього класу описує канонічний шлях обробки одного повідомлення в межах саги, що складається з семи послідовних кроків.

Крок перший — встановлення ambient-контексту конкурентності: значення SagaConcurrencyContext.Mode присвоюється з метаданих дескриптора саги. Цей контекст пізніше зчитується з SqlSagaRepository, який на його підставі вмикає або не вмикає вираз SELECT … FOR UPDATE.

Крок другий — старт транзакції викликом ITransactionManager.StartTransactionAsync(). У SQL-режимі цей виклик призводить до BeginTransactionAsync на рівні DbContext; у режимі InMemory повертається no-op-транзакція, що задовольняє контракт без реальних блокувань.

Крок третій — пошук екземпляра саги через ISagaRepository.FindByCorrelationIdAsync(correlationId). У песимістичному режимі цей виклик блокує рядок до commit-у. У разі ненайдення екземпляра і належності типу повідомлення до ініціюючих створюється новий екземпляр; інакше повідомлення відхиляється з логуванням.

Крок четвертий — виконання стан-машини через виклик ISagaExecutor.ExecuteAsync(), який, своєю чергою, делегує виконання методу SagaStateMachine.HandleAsync(). Цей виклик і запускає поведінковий конвеєр з відповідних activity. У разі оголошення для саги атрибута [SagaTimeout] виконання обмежується стратегією TimeoutStrategy бібліотеки Polly; перевищення часу породжує TimeoutException з інформативним описом.

Крок п'ятий — оцінка стану екземпляра після завершення конвеєра. У разі рівності CurrentState значенню Completed (для скінченних саг, де явно не потрібно зберігати історію) викликається ISagaRepository.DeleteAsync(); інакше — SaveAsync(), який інкрементує Version для оптимістичного контролю конкурентності.

Крок шостий — фіксація транзакції викликом transaction.CommitAsync() за умови відсутності винятків. Усі повідомлення, додані в outbox під час кроку четвертого, фіксуються одночасно зі змінами стану саги — це і є гарантія атомарності, заради якої запроваджено Transactional Outbox.

Крок сьомий — обробка винятків. У разі виникнення винятку (зокрема SagaConcurrencyException від оптимістичного контролю конкурентності) транзакція відкочується, а виняток пробрасується вище, до MessageProcessor і далі до підписника, де може бути оброблений політикою повторних спроб транспорту.

3.2.8 Стан-машина TripSaga як приклад завершеного сценарію

На базі описаних механізмів сагу бронювання поїздки, що використовує два сервіси-учасники (готельний та авіаційний) та містить точку неповернення після підтвердження польоту, реалізовано у вигляді одного компактного класу. Стан-машину цієї саги зображено на рисунку 3.4.

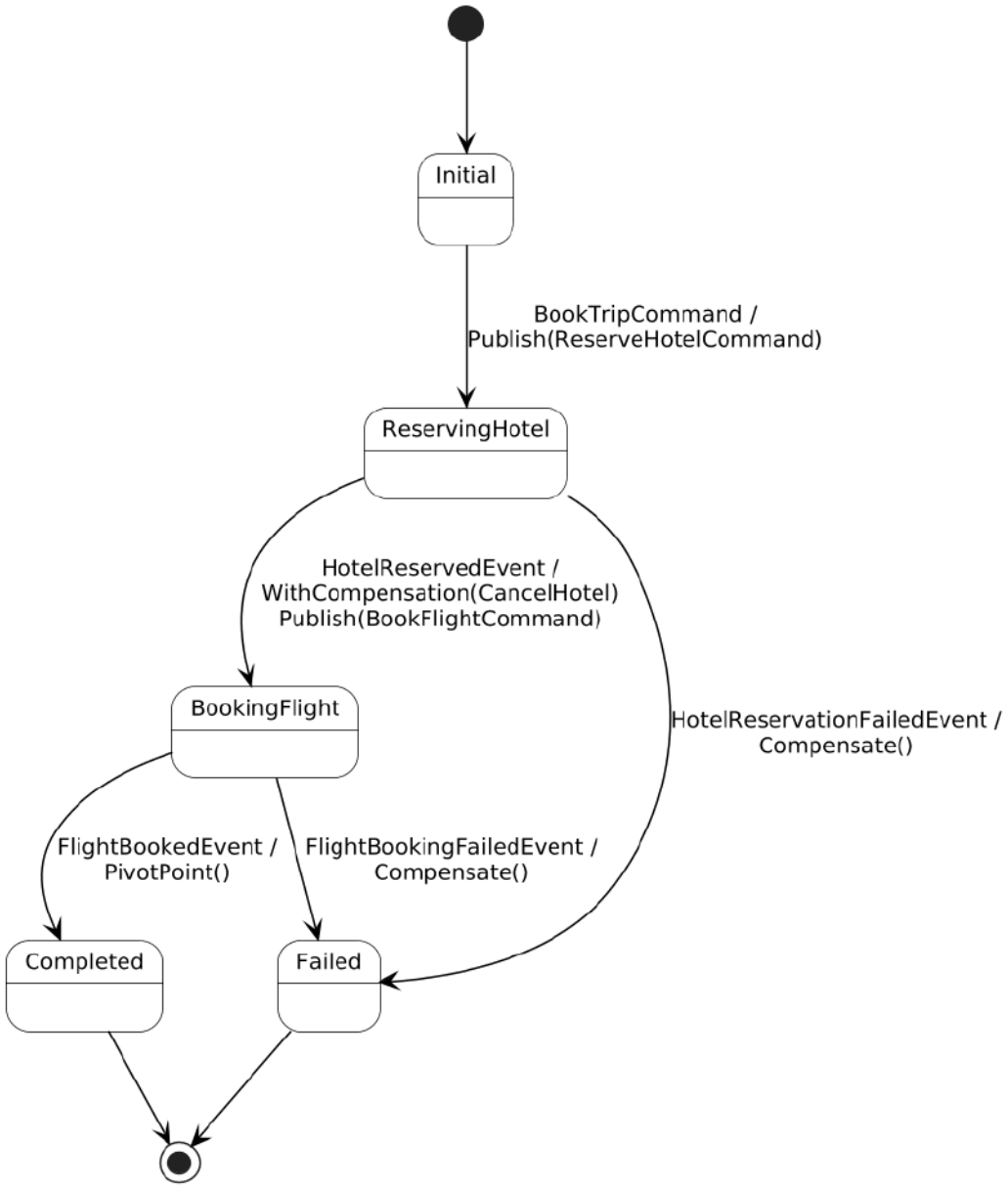

Рисунок 3.4 – Стан-машина саги бронювання поїздки (TripSaga) з компенсаційними переходами та точкою неповернення

Як видно з рисунка 3.4, кожен стан саги відповідає окремій фазі бізнес-сценарію, кожен перехід оголошений декларативно, а вся інфраструктурна робота — транзакції, outbox, ідемпотентність, контроль конкурентності, виконання компенсацій — залишається непомітною для прикладного коду. У цьому прикладі явно представлено всі ключові механізми, описані в попередніх параграфах: реєстрацію компенсації при успішному бронюванні готелю; точку неповернення після підтвердження польоту, що позначає момент, далі якого автоматичний відкат вже не виконується; автоматичне виконання послідовності

компенсацій при невдачах. Решта механізмів — серіалізація стану в JSON, запис вихідних команд в outbox у спільній транзакції зі зміною стану саги, періодична доставка цих команд у RabbitMQ, ідемпотентна обробка відповідей у сервісах-учасниках, маршрутизація відповідей назад до того самого екземпляра саги за CorrelationId — виконуються ядром бібліотеки без явної участі прикладного коду.

3.3 Інтеграційне тестування та моделювання помилкових сценаріїв

Описана в попередніх підрозділах реалізація механізмів координації саги формулює певний набір інваріантів, дотримання яких має забезпечуватися у будь-яких комбінаціях зовнішніх умов: атомарність зміни стану саги та запису в outbox, незалежність зворотного порядку компенсацій від конкретного порядку доставки повідомлень, коректне розв’язання конфліктів конкурентного доступу, дедуплікація повторних повідомлень. Перевірка цих інваріантів становить основну задачу даного підрозділу. Виклад побудовано так: спочатку обґрунтовано стратегію тестування на двох рівнях деталізації, далі описано інфраструктуру тестового середовища, після чого послідовно проаналізовано покриття всіх класів помилкових сценаріїв, сформульованих у підрозділі 2.5. Завершується підрозділ кількісним підсумком — обсягом тестового набору та метриками покриття коду.

3.3.1 Стратегія тестування та її обґрунтування

Тестове покриття SagaFlow побудовано за дворівневим принципом, що відповідає рекомендаціям С. Ньюмана щодо тестової піраміди для мікросервісних систем [2]. Перший рівень утворюють модульні тести з повністю керованою інфраструктурою — In-Memory-реалізаціями всіх портів ядра. Другий рівень становлять інтеграційні тести з реальними примірниками PostgreSQL та RabbitMQ, що піднімаються в окремих контейнерах перед запуском тестового класу. Поділ на два рівні зумовлений принциповою відмінністю задач, що ці рівні розв’язують. Модульні тести перевіряють логічну

коректність окремих компонентів — поведінку SagaProcessor, послідовність викликів у поведінковому конвеєрі, обчислення SHA-64 ключа в OutboxMessagePublisher — і виконуються за частки секунди, що уможливлює їх запуск після кожної зміни коду. Інтеграційні тести перевіряють властивості, що залежать від конкретної СКБД та брокера повідомлень: блокування рядка SELECT … FOR UPDATE, оптимістичну конкурентність через токен версії, фактичну доставку через Direct Exchange RabbitMQ. Жодне з цих явищ не може бути коректно змодельоване в пам’яті процесу — спроба замінити справжню СКБД мок-об’єктом неминуче призвела б до тестування мок-об’єкта замість справжньої поведінки бази даних, проти чого зокрема застерігає К. Річардсон у викладі патернів тестування мікросервісів [3].

Сценарну логіку тестів архітектурно відокремлено від конкретної інфраструктури, на якій ця логіка виконується. Кожен сценарій реалізовано як абстрактний клас, що приймає набір портів через конструктор і містить публічні методи з фактичними перевірками; конкретні тестові класи лише фіксують підстановку інфраструктури. Завдяки такій декомпозиції той самий сценарій SimpleSagaScenario, що описує штатне виконання саги, виконується на чотирьох різних комбінаціях інфраструктури — In-Memory, PostgreSQL без зовнішнього транспорту, PostgreSQL з RabbitMQ та SQLite з RabbitMQ — без дублювання коду перевірок. Результат тестування на кожній комбінації незалежно фіксується у звіті xUnit, що уможливлює локалізацію розбіжностей у поведінці між адаптерами навіть тоді, коли загальний інваріант сценарію зберігається.

3.3.2 Інфраструктура тестового середовища

Модульні тести використовують In-Memory-реалізації, описані у підрозділах 2.2 та 3.1.3. Підписник InMemorySubscriber побудовано на бібліотеці System.Threading.Channels: канал з обмеженою ємністю та режимом BoundedChannelFullMode.Wait виконує роль внутрішньопроцесного транспорту з природною підтримкою backpressure. Сховище саг InMemorySagaRepository та outbox InMemoryOutboxStore використовують ConcurrentDictionary як сховище в

пам’яті. Жоден з цих класів не має зовнішніх залежностей: тестовий процес запускається без необхідності в Docker, локально встановленій СКБД або брокері повідомлень, що дає можливість виконати весь набір модульних тестів за лічені секунди в межах звичайного циклу розробки.

Інтеграційні тести використовують бібліотеку Testcontainers для .NET, що автоматизує життєвий цикл Docker-контейнерів у межах сесії тестування [23]. Дві фікстури реалізують підняття та зупинку контейнерів: PostgresContainerFixture запускає образ postgres:17-alpine з ініціалізованою схемою таблиць SagaFlow; RabbitMqContainerFixture запускає образ rabbitmq:4-management-alpine з типовою конфігурацією exchange та черг, потрібних адаптеру SagaFlow.Rabbitmq. Обидві фікстури зареєстровано як ICollectionFixture в термінах xUnit v3 [24], завдяки чому контейнери піднімаються один раз на весь набір тестових класів, що використовують їх, та звільняються після завершення сесії. Цей підхід дає короткий час одиничного запуску інтеграційного тесту (від другого тесту в наборі — мілісекунди на старт сценарію) ціною одноразової паузи на початку сесії, що йде на завантаження образів у Docker.

Розподіл основних класів тестів за рівнями інфраструктури наведено в таблиці 3.2.

Таблиця 3.2 – Розподіл тестових класів за рівнями інфраструктури

Тестовий клас	**Інфраструктура**
InMemorySimpleSagaTests	InMemory
InMemoryIdempotentMessageTests	InMemory
InMemoryConcurrentSagaAccessTests	InMemory
InMemoryCompensationSagaTests	InMemory
InMemoryCompensableConsumerTests	InMemory
InMemoryParallelMultiStartSagaTests	InMemory
PostgresSimpleSagaTests … PostgresCompensationSagaTests	Testcontainers PostgreSQL
SqlIdempotencyStoreTests, SqlOutboxStoreTests	Testcontainers PostgreSQL

SqlSagaRepositoryTests, SqlTransactionManagerTests	Testcontainers PostgreSQL
RabbitMqIntegrationTests	Testcontainers RabbitMQ
PostgresRabbitMqSimpleSagaTests … PostgresRabbitMqIdempotentMessageTests	Testcontainers PostgreSQL + RabbitMQ
SqlRabbitMqSimpleSagaTests … SqlRabbitMqCompensationSagaTests	Testcontainers PostgreSQL + RabbitMQ

3.3.3 Перевірка механізмів компенсації та точки неповернення

Найперше місце в покритті помилкових сценаріїв займає механізм автоматичної компенсації, запроєктований у підрозділі 2.5. Перевірці підлягають дві ключові властивості: впорядкованість виконання компенсацій у зворотному до прямого виконання порядку та коректне блокування компенсацій після проходження точки неповернення.

Тест Compensate_ThreeSteps_PublishesInReverseOrder створює сагу з трьома послідовними кроками, для кожного з яких через WithCompensation у Fluent API оголошено компенсаційне повідомлення. Сценарій штучно проводить сагу через три успішні переходи (заповнюючи Instance.CompletedSteps трьома індексами), після чого подає повідомлення про невдачу заключного кроку. Зафіксовані виклики IMessagePublisher.PublishAsync порівнюються із семантикою компенсацій, формалізованою Г. Гарсією-Моліною та К. Салемом [22], яка вимагає виконання C_n, C_{n-1}, …, C_1 саме в такому порядку. Тіло тесту наведено нижче.

```
[Fact]
public async Task Compensate_ThreeSteps_PublishesInReverseOrder()
{
    var saga = new ThreeStepCompensationSaga();
    var (sp, publisher) = CreateServiceProvider();
    var instance = new OrderSagaInstance();
    var publishedMessages = new List<IMessage>();

    publisher.PublishAsync(Arg.Any<IMessage>(),
Arg.Any<CancellationToken>())
        .Returns(Task.CompletedTask)
        .AndDoes(ci =>
publishedMessages.Add(ci.ArgAt<IMessage>(0)));
```

```
    await saga.HandleAsync(CreateContext(instance, new
OrderSubmitted(…), sp));
    await saga.HandleAsync(CreateContext(instance, new
StockReserved(…), sp));
    await saga.HandleAsync(CreateContext(instance, new
PaymentProcessed(…), sp));
    instance.CompletedSteps.Count.ShouldBe(3);
    publishedMessages.Clear();

    await saga.HandleAsync(CreateContext(instance, new
PaymentFailed(…), sp));

    instance.CurrentState.ShouldBe("Failed");
    publishedMessages.Count.ShouldBe(3);
    publishedMessages[0].ShouldBeOfType<RefundPayment>();
    publishedMessages[1].ShouldBeOfType<ReleaseStock>();
    publishedMessages[2].ShouldBeOfType<CancelOrder>();
}
```

Симетричний тест Compensate_AfterPivot_SkipsCompensationBecausePivotWasReached перевіряє альтернативну гілку поведінки. Сагу проводять до точки, на якій декларативно оголошено .PivotPoint(), що встановлює прапор IsPivotReached, після чого подають повідомлення про невдачу. Тест очікує, що екземпляр саги переходить у стан Failed, проте список опублікованих повідомлень залишається порожнім: компенсації заблоковано прапором, а їх виконання, як обґрунтовано в підрозділі 2.5, було б семантично некоректним після проходження точки неповернення.

Окремий тест CompensableConsumerTests.InvokeAsync_CompensableConsumer_Failure_CallsCompensateThenThrows перевіряє альтернативний механізм локальної компенсації — для прямих споживачів, що реалізують ICompensableMessageConsumer<T>: у разі винятку в Consume() ConsumerInvoker викликає Compensate() перед повторним пробросом виключення вгору, чим уможливлюється локальний відкат побічних ефектів конкретного сервісу-учасника без участі координатора саги. Симетричний тест на стороні SagaProcessor (ProcessAsync_WhenExecutorThrows_RollsBackTransaction) додатково

підтверджує: при будь-якому винятку, що залишає поведінковий конвеєр без обробки, SagaProcessor явно викликає transaction.RollbackAsync() і не викликає CommitAsync(), внаслідок чого ані зміни стану саги, ані записи в outbox не стають видимими для решти системи.

3.3.4 Перевірка механізмів конкурентного доступу

Песимістичний та оптимістичний режими контролю конкурентного доступу, описані в підрозділі 2.5, перевіряються двома самостійними сценаріями на справжній СКБД через Testcontainers.

Тест SaveAsync_WhenConcurrentModification_ShouldThrowSagaConcurrencyException моделює конфлікт оптимістичного режиму. Два незалежних DbContext завантажують один і той самий екземпляр саги з початковою версією токена, що дорівнює одиниці. Перший контекст зберігає зміни, унаслідок чого EF Core інкрементує версію до двох у базі. При спробі зберегти через другий контекст із початковою версією EF Core порівнює очікувану версію з фактичною (вона вже два) і кидає DbUpdateConcurrencyException. SqlSagaRepository перехоплює цей виняток і пробрасає далі як SagaConcurrencyException, заповнюючи поля CorrelationId та SagaType. Тест перевіряє і факт винятку, і коректність заповнення цих полів — інформація з SagaConcurrencyException потрібна верхнім шарам системи для прийняття рішення про повторну спробу обробки повідомлення.

Тест FindByCorrelationIdAsync_Pessimistic_BlocksConcurrentAccessToSameSagaRow перевіряє очікувану поведінку песимістичного режиму. У піднятий через Testcontainers примірник PostgreSQL вставлено один екземпляр саги. Дві паралельні задачі координуються через TaskCompletionSource: задача A відкриває транзакцію, виконує FindByCorrelationIdAsync у режимі SagaConcurrencyMode.Pessimistic (що транслюється у SELECT 1 … FOR UPDATE) і фіксує факт захоплення замка, після чого штучно затримує commit;

задача B після старту також намагається захопити той самий рядок і має очікувати на release замка. Через 400 мс тест перевіряє, що задача B справді ще не завершилася: цей факт є прямим свідченням того, що песимістичний замок реально утримує конкурентного зчитувача на рівні СКБД. Після виклику releaseLockTcs.SetResult() задача A фіксує транзакцію, після чого задача B розблоковується і завершує свою.

Поряд із цим тест StartTransactionAsync_WhenPessimistic_ShouldUseReadCommittedIsolation окремо перевіряє, що SqlTransactionManager відкриває транзакцію саме на рівні Read Committed: SQL-запит SELECT current_setting('transaction_isolation') повертає read committed. Цей факт є істотним саме для PostgreSQL, як зазначено в проєктному рішенні підрозділу 2.5: рівень Repeatable Read у PostgreSQL використовує MVCC-снепшоти, через що очікування на FOR UPDATE замінюється помилкою серіалізації SQLSTATE 40001, і поведінка системи стала б принципово іншою.

3.3.5 Перевірка ідемпотентності, таймаутів і політик повторних спроб

Дворівневу схему ідемпотентності, описану в підрозділі 2.5, перевірено двома незалежними тестами. На рівні споживача тест InvokeAsync_IdempotentMessage_AlreadyProcessed_SkipsExecution підставляє в ConsumerInvoker мок-реалізацію IIdempotencyStore, що повертає true для будь-якого ключа, і очікує жодного виклику Consume() на споживачі. На рівні Outbox тест PublishAsync_WhenDuplicateIdempotencyKey_ShouldSkipSave перевіряє, що OutboxMessagePublisher не зберігає повторне повідомлення з тим самим SHA-64 ключем; допоміжний тест SameIdempotentMessage_ShouldProduceSameKey фіксує детермінізм обчислення — два повідомлення з однаковими значеннями полів, позначених [IdempotencyKeyPart], завжди дають один і той самий 64-символьний шістнадцятковий рядок. Окремий E2E-сценарій ProcessMessage_SameCommandSentTwice_ConsumerInvokedOnlyOnce повторює

перевірку на повному стеку з реальним сховищем ідемпотентності у PostgreSQL, чим знімає сумнів у тому, що поведінка модульної реалізації не розходиться з реальним сховищем.

Тест ProcessAsync_WhenExecutorExceedsTimeout_ThrowsTimeoutException перевіряє правильність обгортання SagaExecutor у Polly TimeoutStrategy. У дескрипторі саги встановлено таймаут 100 мс; підставлений виконавець «зависає» на 10 с через виклик Task.Delay. Тест очікує TimeoutException приблизно через 100 мс після старту обробки. Граничні випадки самого атрибута [SagaTimeout] перевірено окремим класом SagaTimeoutAttributeTests: відмова від нульового та від'ємного значень TimeoutMilliseconds, коректне зберігання заданого значення Timeout та обмеження AttributeTargets лише класами стан-машин.

Політики повторних спроб, оголошені атрибутом [HandlerRetryPolicy], перевірено серією тестів класу ConsumerInvokerTests. Тест InvokeAsync_RetryableConsumer_RetriesOnFailure фіксує, що споживач з MaxRetries = 3 при кожному винятку отримує повторні виклики Consume() згідно з обраною стратегією Exponential Backoff. Симетричний тест InvokeAsync_NonRetryableException_FailsImmediately перевіряє інверсний інваріант: якщо тип кинутого винятку міститься в списку NonRetryableExceptions, ConsumerInvoker не виконує жодної повторної спроби, а виняток проростає одразу. Альтернативна форма того самого правила перевіряється тестом InvokeAsync_SelectiveRetryExceptions_DoesNotRetryUnlistedExceptions: якщо задано перелік RetryOnExceptions, винятки інших типів вважаються non-retryable за замовчуванням.

Поведінку Outbox при вичерпанні MaxRetries перевірено тестом ProcessPendingMessagesAsync_WhenRetriesExhausted_ShouldTransitionToFailed. У ньому OutboxOptions.MaxRetries встановлено в нуль, а підставлений IOutboxDispatcher щоразу кидає виняток. Тест очікує, що після першої ж невдалої спроби статус повідомлення стане Failed, а не залишиться Pending: це

відповідає дизайну з підрозділу 2.5, за яким вичерпання повторних спроб переводить запис у термінальний стан, з якого повідомлення не буде відправлено знову, але залишиться в таблиці для подальшої діагностики. Окремо тест Subscriber_WhenMessageTypeCannotBeResolved_ShouldRejectMessage фіксує поведінку RabbitMqSubscriber у разі надходження повідомлення з невідомим CLR-типом у заголовку: підписник логує попередження та надсилає BasicNack з requeue: false, що видаляє неоднозначне повідомлення з черги без повторної доставки. Цей сценарій важливий для запобігання нескінченній циркуляції «отруєних» повідомлень у системі — типовій проблемі при at-least-once-доставці [11].

Підсумкову відповідність між класами помилкових сценаріїв, сформульованими в підрозділі 2.5, та конкретними тестовими методами наведено в таблиці 3.3.

Таблиця 3.3 – Покриття помилкових сценаріїв тестовими методами

Сценарій	**Тестовий метод**
Повторна спроба після винятку у Consume()	ConsumerInvokerTests.InvokeAsync_RetryableConsumer_RetriesOnFailure
Відкат транзакції при винятку та re-delivery	SagaProcessorTests.ProcessAsync_WhenExecutorThrows_RollsBackTransaction
Компенсації у зворотному порядку	CompensationTests.Compensate_ThreeSteps_PublishesInReverseOrder
Блокування компенсацій після PivotPoint()	CompensationTests.Compensate_AfterPivot_SkipsCompensationBecausePivotWasReached
Дедуплікація через IIdempotencyStore	ConsumerInvokerTests.InvokeAsync_IdempotentMessage_AlreadyProcessed_SkipsExecution
Дедуплікація Outbox через SHA-64	OutboxMessagePublisherTests.PublishAsync_WhenDuplicateIdempotencyKey_ShouldSkipSave
Конфлікт оптимістичної конкурентності	SqlSagaRepositoryTests.SaveAsync_WhenConcurrentModification_ShouldThrowSagaConcurrencyException
Песимістичне блокування SELECT … FOR UPDATE	SqlSagaRepositoryTests.FindByCorrelationIdAsync_Pessimistic_BlocksConcurrentAccessToSameSagaRow

Таймаут саги через [SagaTimeout]	SagaProcessorTests.ProcessAsync_WhenExecutorExceedsTimeout_ThrowsTimeoutException
MaxRetries у Outbox → Failed	OutboxProcessorTests.ProcessPendingMessagesAsync_WhenRetriesExhausted_ShouldTransitionToFailed
Non-retryable виняток → fail-fast	ConsumerInvokerTests.InvokeAsync_NonRetryableException_FailsImmediately
Авто-компенсація ICompensableConsumer	CompensableConsumerTests.InvokeAsync_CompensableConsumer_Failure_CallsCompensateThenThrows
Reject повідомлення з невідомим CLR-типом	RabbitMqIntegrationTests.Subscriber_WhenMessageTypeCannotBeResolved_ShouldRejectMessage

3.3.6 Кількісні показники та покриття коду тестами

Загальний обсяг тестового набору становить 381 тест у чотирьох тестових проєктах: 229 модульних тестів у проєкті SagaFlow.Tests, 47 інтеграційних тестів у SagaFlow.Sql.Test, 51 інтеграційний тест у SagaFlow.Rabbitmq.Tests та 54 наскрізні (E2E) тести у SagaFlow.E2E.Test. Усі тести проходять успішно (0 failed) у конфігурації Debug на платформі .NET 10. Жоден тест не позначено як [Skip], відсутні нестабільні тести (flaky), залежні від часу або порядку виконання.

Покриття коду виміряно інструментами Coverlet та dotnet-coverage у режимах покриття за рядками, гілками та методами для пакета SagaFlow.Core. Сумарні значення такі: покриття за рядками — 95,4 % (4087 з 4283 рядків), покриття за гілками — 86,6 % (428 з 494 гілок), покриття за методами — 93,5 % (610 з 652 методів). Різниця між покриттям за методами 93,5 % та повним покриттям за методами 90,0 % означає, що приблизно в кожному двадцятому методі залишається гілка, не виконана у жодному тесті — здебільшого це гілки оброблення винятків при некоректній конфігурації, які важко відтворити синтетично без перебудови самого DI-контейнера. Розподіл покриття за рядками між ключовими класами наведено в таблиці 3.4.

Таблиця 3.4 – Покриття коду тестами за класами SagaFlow.Core

Клас	**Line coverage**
SagaProcessor<TSaga>	100 %
SagaExecutor<TSaga>	100 %
SagaStateMachine<TSaga>	100 %
OutboxProcessor	100 %
OutboxMessagePublisher	100 %
IdempotencyKeyResolver	100 %
SagaConcurrencyException	100 %
NoOpTransactionManager	100 %
CompensateActivity	88,2 %
ConsumerInvoker	85,3 %
SagaFlowConfigurator	75,4 %
SubscribersBackgroundService	0 % (юніт) / покривається E2E
JsonOutboxMessageSerializer	0 % (юніт) / покривається E2E

Класи, що відповідають за центральні інваріанти системи — SagaProcessor, SagaExecutor, OutboxProcessor, OutboxMessagePublisher, SagaStateMachine, SagaConcurrencyException, IdempotencyKeyResolver, NoOpTransactionManager — мають стовідсоткове покриття за рядками, що відповідає вимозі NFR-5 «тестованість», сформульованій у підрозділі 2.1. Декілька класів мають частково неповне покриття, і для кожного з них наявне окреме обґрунтування. Клас SagaFlowConfigurator (75,4 %) реєструє сервіси в контейнері впровадження залежностей, і неперевірені гілки відповідають комбінаціям конфігурації, що не входять до типового сценарію використання — переважно це шляхи оброблення винятків при некоректному порядку викликів API. Клас SubscribersBackgroundService на рівні модульних тестів має нульове покриття, оскільки є реалізацією IHostedService, ізольоване виконання якої без хосту .NET принципово неможливе; перевірка його поведінки винесена в E2E-тести з Testcontainers, де ця функціональність покривається опосередковано — через спостереження за фактом доставки повідомлень з реального брокера у реальний обробник. Аналогічна ситуація склалася з JsonOutboxMessageSerializer, чия

поведінка перевіряється не власними юніт-тестами, а через коректність round-trip серіалізації повідомлень в усіх E2E-сценаріях.

3.4 Аналіз продуктивності та масштабованості системи

У підрозділі 3.3 показано функціональну коректність SagaFlow на сукупності 381 тесту, які перевіряють заявлені інваріанти системи. Цей підрозділ присвячений другому аспекту якості — кількісним характеристикам продуктивності та масштабованості. Завдання вимірювань — не зафіксувати абсолютні максимуми пропускної здатності, а локалізувати, в яких компонентах системи витрачається час і пам’ять, на скільки порядків ці витрати відрізняються між собою та які налаштування інфраструктури реально впливають на спостережувані показники. Виклад побудовано від найменшого зрізу системи до найбільшого: спочатку методика вимірювань, далі накладні витрати самого конвеєра обробки, потім пропускна здатність на синтетичному In-Memory-сховищі, після цього однопотокова латентність на PostgreSQL та її залежність від режиму конкурентного доступу, далі поведінка системи при паралельному виконанні з контеншном і без, і нарешті — наскрізна латентність крізь RabbitMQ з аналізом основних параметрів транспорту.

3.4.1 Методика вимірювання та тестове середовище

Усі бенчмарки виконано бібліотекою BenchmarkDotNet версії 0.15.8 [25], яка автоматизує необхідні для коректних вимірювань кроки: прогрів JIT-компілятора, ізоляцію кожного сценарію в окремому процесі, відкидання очевидно аномальних запусків і обчислення довірчих інтервалів за серією повторних замірів. Для всіх сценаріїв, що змінюють зовнішній стан (рядки в БД, повідомлення в черзі), встановлено InvocationCount = 1: один замір — одна інвокація. Це позбавляє BenchmarkDotNet можливості амортизувати фіксовані витрати запуску за множину викликів, тож замість компенсації — підвищено iterationCount до 10–15 і warmupCount до 2–3, що відповідає рекомендаціям А. Акіньшина [26] для бенчмарків з I/O.

Апаратна частина — Intel Core i9-11900 (8 фізичних і 16 логічних ядер), операційна система Windows 11 (10.0.26200.8246). Виконавче середовище — .NET 10.0.7, JIT-компілятор RyuJIT з цільовим набором інструкцій x86-64-v4. Контейнерована інфраструктура піднімалася в Docker Desktop on WSL2: для PostgreSQL використано образ postgres:17-alpine, для RabbitMQ — rabbitmq:4-management-alpine. Вимірювання виконано в єдиному прогоні 2026-05-05; його сумарна тривалість становила приблизно 28 хвилин на 28 бенчмарк-кейсів. Усі виміряні значення нижче подано у вигляді «середнє ± стандартне відхилення» в одиницях, природних для відповідної шкали.

3.4.2 Накладні витрати конвеєра та пропускна здатність In-Memory-конфігурації

Бенчмарк PipelineOverheadBenchmark вимірює фіксовану складову часу, яку додає поведінковий конвеєр SagaFlow до простого виклику делегата. Базою для порівняння є виклик асинхронного делегата, що повертає завершений Task: 57,96 ± 0,71 нс, 32 Б виділеної пам’яті — це нижня межа, з якою має сенс порівнювати будь-яку індиректність взагалі. Повний цикл саги через конвеєр з In-Memory-сховищем (FullCycle_InMemory: ініціювальне повідомлення, його опрацювання, повідомлення-продовження, фіксація стану) виконується за 2,38 ± 0,02 мкс при виділеній пам’яті 3640 Б на цикл — приблизно у 41 раз повільніше за чистий делегат. Сценарій ініціювального повідомлення (Initiate_InMemory) повільніший за повний цикл і триває 2,97 ± 0,26 мкс, оскільки перший прохід оплачує одноразові витрати JIT і кешу, а екземпляр саги ще треба створити викликом Activator.CreateInstance<TSaga>(); другий прохід у повному циклі вже знаходить екземпляр у словнику й уникає цих витрат.

Структура виділення 1872 Б, що припадають на ініціювальне повідомлення, складається з кількох добре локалізованих джерел. Це сам екземпляр стан-машини саги, scope-залежності, які створює DI-контейнер на час одного повідомлення (SagaProcessor, SagaExecutor, SagaRepository, TransactionManager, SagaConcurrencyContext), асинхронні стан-машини async-

методів ProcessAsync, ProcessCoreAsync, ExecuteAsync і SaveAsync, оголошений запис IMessageContext<TMessage> та неуникний запис структурованого логування. Жоден з цих компонентів не є зайвим з погляду архітектури системи, проте всі разом утворюють константний префікс часу, який лежить на кожному викликові SagaFlow і яким, у пропорції до подальших операцій, можна знехтувати тільки за умови, що подальші операції суттєво довші.

Бенчмарк InMemoryThroughputBenchmark проганяє послідовно та паралельно (через Task.WhenAll) пакет з N сагалей, де N набуває значень 10, 50, 100. Усереднені показники наведено в таблиці 3.5.

Таблиця 3.5 – Латентність та виділена пам’ять для In-Memory-конфігурації

Сценарій	**N**	**Середнє, мкс**	**На повідомлення, мкс**	**Виділена пам’ять**
Sequential	10	25,30	2,530	35,55 КБ
Parallel	10	25,25	2,525	35,77 КБ
Sequential	50	121,56	2,431	177,74 КБ
Parallel	50	123,45	2,469	177,96 КБ
Sequential	100	242,75	2,428	355,49 КБ
Parallel	100	244,35	2,444	355,71 КБ

Перший і найважливіший висновок з таблиці 3.5: послідовний і паралельний режими на цьому навантаженні дають практично однакові значення латентності в межах одного N, а виділена пам’ять навіть при N=100 розрізняється на одиниці байтів. Причина — у природі самого навантаження: одиничний цикл саги в In-Memory-режимі займає лічені мікросекунди й витрачає процесорний час майже без I/O, а вартість планування завдання в пулі потоків і передачі продовження між робочими потоками з’їдає весь потенційний виграш від розпаралелювання. Цей результат не специфічний для SagaFlow, а є відомим обмеженням Task.WhenAll над дрібнозернистими CPU-задачами на платформі .NET.

Другий висновок: вартість на повідомлення в In-Memory-режимі стабілізується на рівні 2,4 мкс і збігається із середнім часом FullCycle_InMemory з PipelineOverheadBenchmark. Це підтверджує, що пропускна здатність ядра системи лінійно масштабується з N і не має фіксованих витрат на пакет понад вартість самого конвеєра. Стеля пропускної здатності в чистому In-Memory-режимі на даному обладнанні — приблизно 395 тис. сага/с — є орієнтиром, з яким мають співвідноситися всі вимірювання, у яких задіяно інфраструктуру: будь-яке відставання від цієї стелі мусить бути пояснене конкретною операцією I/O.

3.4.3 Однопотокова латентність на PostgreSQL та режими конкурентного доступу

Бенчмарк PostgresThroughputBenchmark проводить одну сагу через повний цикл (ініціювальне повідомлення, продовження, завершення) на реальному примірнику PostgreSQL, який запущено в Docker-контейнері на тій самій машині. Вимірювання виконано окремо для оптимістичного та песимістичного режимів конкурентного доступу. У режимі Optimistic середня латентність повного циклу становить 5,09 ± 0,38 мс при 170,23 КБ виділеної пам’яті; у режимі Pessimistic — 6,67 ± 0,25 мс при 177,84 КБ. Різниця в латентності 1,58 мс відповідає одній додатковій транзакційній операції — запитові SELECT 1 FROM «SagaFlowSagaInstances» WHERE … FOR UPDATE, який песимістичний режим виконує перед завантаженням сутності. На localhost-через-Docker середній round-trip до PostgreSQL за виміряними значеннями становить 0,2–0,5 мс, що відповідає очікуваному порядку величини. Виділена пам’ять обох режимів збігається в межах 5 %: додатковий запит блокування використовує ті самі буфери Npgsql і ті самі сутності змінного стану EF Core, тож додаткових алокацій не спричиняє.

Розкладання повного циклу на елементарні взаємодії з СКБД дає таку картину. Транзакція починається викликом BEGIN. У песимістичному режимі негайно після цього виконується SELECT 1 … FOR UPDATE з єдиною метою —

захопити блокування рядка. Далі йде основний завантажувальний запит SELECT * FROM SagaFlowSagaInstances, JSON-десеріалізація екземпляра саги, виконання стан-машини для ініціювального повідомлення, JSON-серіалізація і одна з двох операцій INSERT … RETURNING (для нової саги) або UPDATE (для наявної). Той самий цикл повторюється для повідомлення-продовження, з другою серіалізацією і другим UPDATE. Якщо сага в результаті переходить у завершений стан, додається DELETE. До цього додаються записи в outbox-таблицю для всіх повідомлень, що породжуються кроками саги. Транзакція завершується COMMIT, який чекає fsync журналу WAL. Простий підрахунок дає близько 10 round-trip'ів, що при середньому часі 0,3–0,5 мс на одну операцію якраз і утворює спостережувані 5–7 мс.

Виділена пам'ять близько 170 КБ є непропорційно великою для роботи, що зводиться до зміни кількох полів. Профілювання показало, що основними джерелами є дві групи об'єктів. Перша — внутрішні структури change tracker EF Core, які утримують графи відстежуваних сутностей разом із прив'язаними метаданими. Друга — пули буферів Npgsql, які цей драйвер використовує для підготовлених команд та параметрів запитів. Жодне з цих джерел не є виправним у самій SagaFlow: change tracker є частиною EF Core, без якої не працює запис, а буфери Npgsql є частиною його стандартної конфігурації, прийнятної для типових застосунків.

3.4.4 Масштабування при паралельному виконанні та поведінка під контеншном

Бенчмарк PostgresParallelThroughputBenchmark характеризує стелю пропускної здатності, коли N сагалей опрацьовуються одночасно і кожна з них торкається свого, не спільного з іншими, рядка таблиці. Для N = 10 загальний час становить 13,57 ± 0,67 мс (на повідомлення — 1,36 мс, пропускна здатність — близько 737 сага/с); для N = 50 — 40,91 ± 4,48 мс (1222 сага/с); для N = 200 — 136,35 ± 4,13 мс (1467 сага/с). Збільшення N у 20 разів призводить лише до приблизно дворазового зростання пропускної здатності. Це означає виразне

сублінійне масштабування: насичення починається в регіоні N ≈ 50 і виходить на плато при N = 200. Порівняно з однопотоковими 196 сага/с з підрозділу 3.4.3 виграш приблизно у 7,5 разу, при цьому виділена пам’ять на одне повідомлення (близько 180 КБ) лишається такою самою, як у однопотоковому режимі — паралелізм не привносить додаткових алокацій понад ті, що зумовлені самим обсягом роботи.

Причини плато на N = 200 локалізовано в трьох місцях. Перше — типова межа Maximum Pool Size клієнта Npgsql у 100 з’єднань: при 200 паралельних викликах перші сто захоплюють пул, решта серіалізуються в чергу. Друге — конкуренція в самій базі за вставку рядків в outbox-таблицю; навантаження на B-tree-індекс її первинного ключа спричиняє точкову конкуренцію навіть тоді, коли основні рядки sa-таблиці не перетинаються. Третє — латентність fsync журналу WAL: за стандартної конфігурації PostgreSQL synchronous_commit = on кожен COMMIT чекає на запис у журнал, і ця затримка не амортизується паралелізмом для незалежних транзакцій. Із цих трьох вузьких місць перші два долаються конфігураційно (підвищення розмірів пулу клієнта і параметрів max_connections та max_wal_size самої СКБД), третє — лише ціною надійності, тож на продакшен-навантаженнях саме воно лишається фундаментальною межею пропускної здатності.

Бенчмарк PostgresContentionBenchmark моделює протилежний випадок — N паралельних оновлень того самого рядка, що відповідає сазі з тим самим CorrelationId. Така ситуація виникає, наприклад, якщо в систему майже одночасно надходять кілька повідомлень-продовжень для тієї самої саги. Результати наведено в таблиці 3.6.

Таблиця 3.6 – Латентність під контеншном на одній і тій самій сазі

Режим	**Concurrency**	**Середнє, мс**	**Завершених оновлень**
Pessimistic	2	13,87	Усі N
Optimistic	2	11,35	1 (переможець)
Pessimistic	8	33,87	Усі N

Optimistic	8	13,80	1 (переможець)
Pessimistic	32	116,82	Усі N
Optimistic	32	28,49	1 (переможець)

Поверхове прочитання таблиці 3.6 справляє враження, ніби оптимістичний режим у 4 рази продуктивніший за песимістичний при високому контеншні: при concurrency = 32 — 28,49 мс проти 116,82 мс. Проте права колонка кардинально змінює інтерпретацію цих чисел. У песимістичному режимі усі 32 оновлення виконуються послідовно, кожне завершується успішно і фіксується в БД. У оптимістичному режимі лише одне оновлення (переможець) фіксується, а решта 31 у мить виявлення розбіжності токена версії падає з SagaConcurrencyException, відкочує транзакцію і виходить. Спостережувана низька латентність оптимістичного режиму — це усереднення часу одного успішного циклу і 31 швидкого скасування; вона не відповідає кількості виконаної корисної роботи.

З цього випливає чітке практичне правило вибору режиму. Якщо застосунок не реалізує ретраї на SagaConcurrencyException (наприклад, через бібліотеку Polly з експоненційним backoff та повторним завантаженням саги), то для саг, які потенційно отримують паралельні події з тим самим CorrelationId, єдиним коректним вибором є песимістичний режим. Оптимістичний режим у такому сценарії втрачає N − 1 з N оновлень — у наведеному прикладі це 31 з 32, тобто 97 % зовнішньої роботи зникає з системи. Якщо ретраї реалізовано, оптимістичний режим починає мати реальний сенс: на короткий проміжок часу пов’язаний контеншн розв’язується природно, без серіалізації на блокуванні рядка, і за умови низької або помірної конкуренції оптимістичний режим випереджає песимістичний приблизно на 18 % при concurrency = 2. Сама SagaFlow не нав’язує жодного з варіантів — вибір лишається за прикладним кодом, але в реалізації передбачено явний тип винятку SagaConcurrencyException, що дає змогу побудувати ретрай-політику без спеціального відстежування внутрішніх типів EF Core.

3.4.5 Наскрізна латентність та налаштування транспорту RabbitMQ

Бенчмарк RabbitMqEndToEndBenchmark вимірює час, що минає від моменту, коли сага записує вихідне повідомлення в outbox-таблицю, до моменту, коли інша сага в тому самому застосунку отримує це повідомлення з черги, опрацьовує і сигналізує про завершення. У середньому цей час дорівнює 6,14 ± 0,49 мс при 101,21 КБ виділеної пам’яті за один цикл. Різниця між цим значенням і 5,09 мс однопотокового PostgreSQL з підрозділу 3.4.3 — приблизно 1 мс — відповідає сумарній вартості шести операцій: вставки рядка в outbox-таблицю в межах транзакції саги, опитування цього рядка outbox-поллером, виклику publish у канал AMQP, маршрутизації повідомлення в broker до черги, доставки споживачеві через consumer prefetch і фінальної десеріалізації перед поверненням у конвеєр. Стандартне відхилення вимірювання вище, ніж в однопотоковому Postgres-бенчмарку (8,0 % проти 7,6 %), що пов’язано з шумом у плануванні broker і channel-thread в RabbitMQ; при латентності 6 мс це абсолютна похибка приблизно 0,5 мс, що для бенчмарку з I/O лишається в межах прийнятного.

Бенчмарк RabbitMqPrefetchBenchmark зосереджений на одному параметрі підписника RabbitMQ — PrefetchCount, який задає, скільки повідомлень broker дозволяє надіслати споживачеві без отримання підтвердження ack [27]. На навантаженні з 200 паралельних сагалей за ітерацію значення PrefetchCount = 1 дає 751,6 ± 76,3 мс (≈266 повід./с), значення 10 — 660,7 ± 40,5 мс (≈303 повід./с), значення 50 — 670,3 ± 62,0 мс (≈298 повід./с), значення 100 — 654,3 ± 55,9 мс (≈306 повід./с). Різниця між 1 і 10 — близько 14 %, тоді як подальше збільшення вже не дає виміряного приросту: довірчі інтервали значень 10, 50 і 100 істотно перекриваються. Поясненням є логіка ack-cycle самого протоколу AMQP: при PrefetchCount = 1 broker пересилає повідомлення лише після отримання ack за попереднє, що утворює послідовний стіл round-trip’ів типу «один зайшов — один підтвердив»; при будь-якому значенні від 10 і вище broker встигає вислати наступну партію до моменту, коли споживач закінчить попередню. Збільшення понад 10 не дає виграшу, оскільки пропускну здатність на цьому навантаженні

обмежує не транспорт, а сам конвеєр обробки в споживачі. Виходячи з цих вимірювань, типове значення PrefetchCount = 10 є розумним за замовчуванням для застосунків, побудованих на SagaFlow, з одним зауваженням: для повідомлень із великим обсягом тіла або з дорогою прикладною обробкою на споживачеві встановлення значення помітно більшого за 10 створює ризик надмірного буферування і витіснення з пам’яті споживача, тож слід підвищувати його лише за умови підтвердженої необхідності.

Бенчмарк RabbitMqBatchSizeBenchmark характеризує другий параметр — розмір пакета OutboxOptions.BatchSize, що визначає, скільки повідомлень outbox-поллер забирає за одне опитування. На навантаженні з 1500 паралельних сагалей за ітерацію значення BatchSize = 10 дає 9,29 ± 0,02 с (≈161 повід./с), BatchSize = 100 — 8,62 ± 0,10 с (≈174 повід./с), BatchSize = 500 — 8,30 ± 0,27 с (≈181 повід./с). Зростання пропускної здатності від 10 до 500 — близько 12 %, причому з 100 на 500 ріст відбувається в межах одного стандартного відхилення.

Кореневою причиною неефективності малого BatchSize є кількість опитувань: щоб опрацювати 1500 повідомлень з пакетом 10, потрібно 150 запитів SELECT … FROM SagaFlowOutboxMessages WHERE Status = 0 ORDER BY CreatedAtUtc LIMIT 10, тоді як з пакетом 500 — лише 3. Кожен такий запит — round-trip до СКБД, який у наведених вимірюваннях відповідає приблизно 0,3 мс. Параметр BatchSize при цьому не є реальним важелем оптимізації пропускної здатності outbox-шляху: основні витрати лежать не в кількості опитувань, а в самій per-message-вартості — приблизно 240 КБ виділеної пам’яті на повідомлення проти 102 КБ при прямій публікації. Ця різниця у 2,3 рази утворюється з трьох додаткових компонентів: round-trip EF Core на запис рядка outbox, повного циклу зміни стану саги при опрацюванні події-продовження, і логічного запису операцій outbox-поллера. Конкретні шляхи зниження цієї вартості (групове оновлення статусу повідомлень одним SQL-запитом замість поодиноких ExecuteUpdate; обхід подвійної серіалізації payload у тих випадках, коли він уже представлений як JObject або byte[]; використання SELECT … FOR UPDATE SKIP LOCKED для безпечного горизонтального масштабування

кількох екземплярів outbox-поллера) виходять за межі поточної версії SagaFlow і становлять окрему інженерну задачу для подальших ітерацій.

Висновки до розділу 3

У розділі описано реалізацію та експериментальну перевірку SagaFlow. Кодова база організована за принципом «портів і адаптерів»: ядро не залежить від конкретних провайдерів сховища і транспорту, а інфраструктурні модулі (PostgreSQL з EF Core, RabbitMQ) підключаються на етапі конфігурації контейнера впровадження залежностей. Координацію транзакцій реалізовано через декларативну стан-машину, виконану як ланцюг відповідальності, з відокремленим механізмом компенсацій і явною точкою незворотності.

Інтеграційне тестування на 381 тесті з покриттям 95,4 % за рядками підтвердило коректність ключових інваріантів — автоматичної компенсації у зворотному порядку, точки неповернення, обох режимів конкурентного доступу, дворівневої ідемпотентності, таймаутів і політик повторних спроб. Сценарно-адаптерна архітектура тестового набору дала змогу перевірити одні й ті самі сценарії на чотирьох комбінаціях інфраструктури без дублювання коду.

Бенчмарки показали, що ядро (близько 2,4 мкс на цикл саги) не є вузьким місцем у жодному реалістичному сценарії. На PostgreSQL одна сага в однопотоковому режимі виконується за 5,1–6,7 мс, наскрізна латентність крізь RabbitMQ становить близько 6,1 мс. Для саг, що приймають паралельні події, оптимістичний режим без явної ретрай-політики втрачає більшість оновлень під контеншном, тож практичний вибір зводиться до песимістичного контролю конкурентності. У межах протестованих навантажень система задовольняє вимоги, заявлені в підрозділі 2.1.

ВИСНОВКИ

Мета роботи — проєктування та реалізація системи координації розподілених транзакцій між мікросервісами на основі оркестрації Saga — досягнута. Розроблено бібліотеку SagaFlow для платформи .NET, яка підтримує декларативний опис саг, автоматичне виконання компенсувальних дій, дворівневу ідемпотентність, керовані політики повторів і таймаутів, а також розподілений сценарій із PostgreSQL і RabbitMQ. Завдання, сформульовані у вступі, виконано в обсязі, який дозволив дійти результатів, перевірених як модульними, так і інтеграційними тестами на реальній інфраструктурі.

У теоретичній частині систематизовано підходи до координації розподілених транзакцій у мікросервісних системах: транзакційний поштовий ящик, захоплення змін даних, модель подієвого джерела, протокол Try-Confirm-Cancel і патерн Saga. Показано, чому класичні протоколи на основі двофазної фіксації несумісні з принципом незалежного розгортання сервісів і неприйнятні в мікросервісному середовищі. Порівняння оркестрації та хореографії продемонструвало, що для складних і часто змінюваних бізнес-процесів оркестрація дає кращу керованість і діагностованість, тоді як хореографія залишається економнішим вибором для коротких і стабільних сценаріїв.

У проєктній частині запропоновано архітектуру SagaFlow за принципом «портів і адаптерів»: ядро не залежить від конкретних провайдерів сховища і транспорту, а інфраструктурні модулі підключаються на етапі конфігурації контейнера впровадження залежностей. Принципові проєктні рішення — декларативний DSL поверх Fluent API, ланцюг відповідальності для виконання переходів, поділ типу й екземпляра саги, явна точка незворотності — зведено в єдину модель. Семантичну компенсацію відмежовано від синтаксичного відкату, ідемпотентність винесено на рівень обов’язкового контракту обробників, для контролю конкурентного доступу передбачено оптимістичний і песимістичний режими.

У реалізаційній частині створено робочу бібліотеку, що складається з ядра SagaFlow.Core та трьох інфраструктурних адаптерів. Інтеграційне тестування

на 381 тесті з покриттям 95,4 % за рядками підтвердило коректність ключових інваріантів — автоматичної компенсації у зворотному порядку, точки неповернення, обох режимів конкурентного доступу, дворівневої ідемпотентності, таймаутів і політик повторних спроб. Сценарно-адаптерна архітектура тестового набору дала змогу перевірити одні й ті самі сценарії на чотирьох комбінаціях інфраструктури без дублювання коду перевірок.

Бенчмарки показали, що ядро (близько 2,4 мкс на цикл саги) не є вузьким місцем у реалістичних сценаріях. На PostgreSQL одна сага в однопотоковому режимі виконується за 5,1–6,7 мс, наскрізна латентність крізь RabbitMQ становить близько 6,1 мс. Експерименти на 200 паралельних сагах виявили, що оптимістичний режим без явної ретрай-політики втрачає більшість оновлень під контеншном, тож практичним вибором для саг із паралельними подіями залишається песимістичний контроль конкурентності. У межах протестованих навантажень система задовольняє вимоги пропускної здатності й латентності, заявлені в підрозділі 2.1.

Практичне значення роботи полягає в тому, що SagaFlow може застосовуватись як готова бібліотека в .NET-проєктах, де потрібна координація операцій між сервісами, а також використовуватися в навчальному процесі під час вивчення архітектурних патернів і розподілених систем. Серед напрямів подальшого розвитку — горизонтальне масштабування outbox-поллера через SELECT … FOR UPDATE SKIP LOCKED, групове оновлення статусу повідомлень одним SQL-запитом замість поодиноких ExecuteUpdate, додавання адаптерів для Apache Kafka та Azure Service Bus, а також інструментарій для статичного аналізу декларативного опису саг — перевірка досяжності станів, виявлення непокритих переходів і потенційно нескінченних циклів.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Lewis J. Microservices [Electronic resource] / J. Lewis, M. Fowler. — 2014. — URL: https://martinfowler.com/articles/microservices.html.
2. Newman S. Building Microservices: Designing Fine-Grained Systems / S. Newman. — 2nd ed. — Sebastopol: O'Reilly Media, 2021. — 612 p.
3. Richardson C. Microservices Patterns: With examples in Java / C. Richardson. — Shelter Island: Manning Publications, 2018. — 520 p.
4. Wolff E. Microservices: Flexible Software Architecture / E. Wolff. — Boston: Addison-Wesley, 2016. — 368 p.
5. Richards M. Fundamentals of Software Architecture: An Engineering Approach / M. Richards, N. Ford. — Sebastopol: O'Reilly Media, 2020. — 432 p.
6. Richardson C. Pattern: Microservice Architecture [Electronic resource] / C. Richardson // Microservice Architecture. — URL: https://microservices.io/patterns/microservices.html.
7. Cockburn A. Hexagonal architecture [Electronic resource] / A. Cockburn. — 2005. — URL: https://alistair.cockburn.us/hexagonal-architecture/.
8. Evans E. Domain-Driven Design: Tackling Complexity in the Heart of Software / E. Evans. — Boston: Addison-Wesley Professional, 2003. — 560 p.
9. Microservices architecture style [Electronic resource] // Microsoft Azure Architecture Center. — URL: https://learn.microsoft.com/en-us/azure/architecture/guide/architecture-styles/microservices.
10. Lamport L. Time, Clocks, and the Ordering of Events in a Distributed System / L. Lamport // Communications of the ACM. — 1978. — Vol. 21, № 7. — P. 558–565.
11. Kleppmann M. Designing Data-Intensive Applications: The Big Ideas Behind Reliable, Scalable, and Maintainable Systems / M. Kleppmann. — Sebastopol: O'Reilly Media, 2017. — 616 p.

12. Gilbert S. Brewer’s conjecture and the feasibility of consistent, available, partition-tolerant web services / S. Gilbert, N. Lynch // ACM SIGACT News. — 2002. — Vol. 33, № 2. — P. 51–59.
13. Brewer E. CAP Twelve Years Later: How the «Rules» Have Changed / E. Brewer // Computer. — 2012. — Vol. 45, № 2. — P. 23–29.
14. Abadi D. Consistency Tradeoffs in Modern Distributed Database System Design: CAP is Only Part of the Story / D. Abadi // Computer. — 2012. — Vol. 45, № 2. — P. 37–42.
15. Attiya H. Sequential consistency versus linearizability / H. Attiya, J. L. Welch // ACM Transactions on Computer Systems. — 1994. — Vol. 12, № 2. — P. 91–122.
16. Waldron W. The Dual Write Problem [Electronic resource] / W. Waldron // Confluent Developer. — URL: https://developer.confluent.io/courses/microservices/the-dual-write-problem/.
17. Janssen T. Distributed Transactions — Don’t use them for Microservices [Electronic resource] / T. Janssen. — 2021. — URL: https://thorben-janssen.com/distributed-transactions-microservices/.
18. Ibryam B. Distributed transaction patterns for microservices compared [Electronic resource] / B. Ibryam // Red Hat Developer. — 2023. — URL: https://developers.redhat.com/articles/2021/09/21/distributed-transaction-patterns-microservices-compared.
19. Helland P. Life beyond Distributed Transactions: An Apostate’s Opinion / P. Helland // Proceedings of the 3rd Biennial Conference on Innovative Data Systems Research (CIDR). — Asilomar: CIDR, 2007. — P. 132–141.
20. Debezium Documentation [Electronic resource] // Debezium Community. — URL: https://debezium.io/documentation/.
21. Fowler M. Event Sourcing [Electronic resource] / M. Fowler. — 2005. — URL: https://martinfowler.com/eaaDev/EventSourcing.html.
22. Garcia-Molina H. Sagas / H. Garcia-Molina, K. Salem // ACM SIGMOD Record. — 1987. — Vol. 16, № 3. — P. 249–259.

23. Testcontainers for .NET [Electronic resource] // Testcontainers. — URL: https://dotnet.testcontainers.org/.
24. xUnit.net v3 Documentation [Electronic resource] // xUnit.net. — URL: https://xunit.net/docs/getting-started/v3/getting-started.
25. BenchmarkDotNet: Powerful .NET library for benchmarking [Electronic resource] // BenchmarkDotNet. — URL: https://benchmarkdotnet.org/.
26. Akinshin A. Pro .NET Benchmarking: The Art of Performance Measurement / A. Akinshin. — Berkeley: Apress, 2019. — 624 p.
27. Consumer Prefetch [Electronic resource] // RabbitMQ Documentation. — URL: https://www.rabbitmq.com/docs/consumer-prefetch.