ХЕРСОНСЬКИЙ НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ
ФАКУЛЬТЕТ ІНФОРМАЦІЙНИХ ТЕХНОЛОГІЙ ТА ДИЗАЙНУ
КАФЕДРА ПРОГРАМНИХ ЗАСОБІВ І ТЕХНОЛОГІЙ

ПОЯСНЮВАЛЬНА ЗАПИСКА

до кваліфікаційної роботи
бакалавра
(освітньо-кваліфікаційний рівень)

на тему *«Розробка універсального месенджера для інтеграції комунікацій між користувачами на платформах онлайн-продажів та соціальних мереж»*

Виконав: здобувач *4* року навчання
групи *4ПР1*
напряму підготовки (спеціальності)
121 Інженерія програмного забезпечення
(шифр і назва напряму підготовки, спеціальності)

Орлов Кирило Євгенович
(підпис, прізвище та ініціали)

Керівник *Огнєва Оксана Євгенівна*
(підпис, прізвище та ініціали)

Рецензент *к.т.н., доцент Корніловська Н.В.*
(прізвище та ініціали)

Нормоконтролер *Комісаров О. С.*
(підпис, прізвище та ініціали)

Хмельницький – 2026

ХЕРСОНСЬКИЙ НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ

Інститут, факультет, відділення *Інформаційних технологій та дизайну*
Кафедра, циклова комісія *Програмних засобів і технологій*
Освітньо-кваліфікаційний рівень *бакалавр*
Освітньо-професійна підготовка *121 Інженерія програмного забезпечення*
(шифр і назва)
Спеціальність *121 Інженерія програмного забезпечення*
(шифр і назва)

ЗАТВЕРДЖУЮ
Завідувач кафедри *програмних засобів і технологій*
О.Є. Огнєва
«___» _____ 2026 року

ЗАВДАННЯ

НА КВАЛІФІКАЦІЙНУ РОБОТУ ЗДОБУВАЧА

Орлова Кирила Євгеновича
(прізвище, ім'я, по батькові)

1. Тема проєкту (роботи) *Розробка універсального месенджера для інтеграції комунікацій між користувачами на платформах онлайн-продажів та соціальних мереж*

керівник проєкту (роботи) *Огнєва Оксана Євгенівна, к.т.н., доцент*
(прізвище, ім'я, по батькові, науковий ступінь, вчене звання)

затверджені наказом вищого навчального закладу від «16» січня 2026 року №82-с

2. Строк подання здобувачем проєкту(роботи) 10.06.2026

3. Вихідні дані до проєкту (роботи) *літературні та періодичні джерела, матеріали переддипломної практики*

4. Зміст розрахунково-пояснювальної записки (перелік питань, які потрібно розробити)
Аналіз предметної області omnichannel-комунікацій у сфері українського e-commerce, проєктування доменної моделі та ER-схеми бази даних omnichat-hub, розробка серверної частини omnichat-hub на основі NestJS, реалтайм-доставки через Socket.IO з Redis-адаптером та транзакційного Outbox-патерну, тестування та розгортання omnichat-hub у виробничому середовищі.

5. Перелік графічного матеріалу (з точним зазначенням обов'язкових креслень)
21 таблиця, 51 рисунок

6. Консультанти розділів проєкту (роботи)

Розділ	Прізвище, ініціали та посада консультанта	Підпис, дата	
		завдання видав	завдання прийняв

7. Дата видачі завдання *14.01.2026*

КАЛЕНДАРНИЙ ПЛАН

№ з/п	Назва етапів кваліфікаційної роботи	Строк виконання етапів проєкту (роботи)	Примітка
1	Отримання завдання	14.01.2026	Виконано
2	Підбір літератури	17.01.2026-21.01.2026	Виконано
3	Аналіз предметної області	22.01.2026-26.01.2026	Виконано
4	Розробка та обґрунтування завдання	27.01.2026-12.02.2026	Виконано
5	Розробка концептуальної моделі	13.02.2026-19.02.2026	Виконано
6	Моделювання та проєктування системи	20.02.2026-02.03.2026	Виконано
7	Розробка omnichat-hub	03.03.2026-15.04.2026	Виконано
8	Тестування omnichat-hub	16.04.2026-20.04.2026	Виконано
9	Оформлення пояснювальної записки	21.04.2026-02.05.2026	Виконано
10	Захист кваліфікаційної роботи	24.06.2026	Виконано

Здобувач _____ *Орлов К.Є.*
(підпис) (прізвище та ініціали)

Керівник проєкту (роботи) _____ *Огнєва О.Є.*
(підпис) (прізвище та ініціали)

РЕФЕРАТ

Кваліфікаційна робота бакалавра викладена на 90 сторінках основного тексту, містить 51 рисунок, 21 таблицю, 2 додатки та 30 використаних джерел.

Об’єктом проєктування є процеси обміну повідомленнями між продавцями та покупцями на українських онлайн-платформах продажів (маркетплейсах), а саме моделі автентифікації каналів, схеми синхронізації вхідних повідомлень і патерни доставки подій між програмним інтерфейсом та фоновими процесами.

Предметом проєктування є канал-агностична веб-система омніканальної агрегації комунікаційних каналів з підтримкою реалтайм-доставки повідомлень, гарантованої доставки за моделлю at-least-once через транзакційний Outbox-патерн, багатотенантної ізоляції даних на рівні робочого простору та обгорткового шифрування облікових даних каналів.

Мета проєктування полягає у скороченні часу відповіді клієнтам та зниженні втрати звернень для українських ФОП і малого бізнесу сфери електронної комерції шляхом уніфікації комунікацій з маркетплейсами Rozetka, Prom і OLX у єдиний операторський інтерфейс з реалтайм-доставкою повідомлень та горизонтальним масштабуванням.

Галузь застосування програмного забезпечення охоплює електронну комерцію та клієнтську підтримку: обробку звернень покупців операторами інтернет-магазинів і продавців, що одночасно працюють на кількох українських маркетплейсах та потребують єдиного робочого місця для відповідей.

Основні характеристики розробленого програмного забезпечення такі. Реалізовано REST API з версіонуванням v1 та реалтайм-канал на базі Socket.IO з Redis-адаптером для горизонтального масштабування. Доставку подій забезпечує транзакційний Outbox-патерн з моделлю at-least-once і повторним захопленням застряглих подій за часом блокування. Облікові дані каналів зберігаються із застосуванням обгорткового шифрування AES-256-GCM з

ключем шифрування даних на рівні робочого простору. Дані описано реляційною схемою на 24 таблиці. Систему побудовано як monorepo з веб-клієнтом на Next.js і React, мобільним клієнтом на Expo, фоновими воркерами на BullMQ та інфраструктурою як кодом на Pulumi; розгортання виконано на DigitalOcean App Platform і Vercel.

Значущість роботи полягає у канал-агностичній архітектурі адаптерів, що уніфікує три різні моделі авторизації українських маркетплейсів (OAuth 2.0 потоку Authorization Code для OLX, статичні API-токени для Rozetka і Prom) на рівні єдиної абстракції з ізоляцією від доменного шару, завдяки чому нові канали додаються без зміни бізнес-логіки. Функціональне тестування на рівнях unit, integration та E2E підтвердило відповідність системи функціональним вимогам, сформульованим у розділах 1 і 2; продемонстровано здатність системи до виробничого розгортання.

КЛЮЧОВІ СЛОВА: ОМНІКАНАЛЬНИЙ МЕСЕНДЖЕР, МАРКЕТПЛЕЙС, OUTBOX-ПАТЕРН, ШИФРУВАННЯ КЛЮЧІВ, OAUTH 2.0, ОПЕРАТОРСЬКИЙ ДАШБОРД, SaaS

АНОТАЦІЯ

Кваліфікаційна робота бакалавра включає такі структурні частини: вступ, чотири розділи, висновки, список використаних джерел та додатки.

Перший розділ «Аналіз предметної області» містить огляд ринку електронної комерції України, дослідження каналів комунікації українських маркетплейсів Rozetka, Prom, OLX, аналіз існуючих світових та локальних аналогів (Sendbird, Front, Intercom, Re:amaze, Crisp, Zoho Desk, KeyCRM, Sitniks). У розділі описано перехід від моделі «ЯК Є» (роздроблені інтерфейси каналів) до моделі «ЯК ПОВИННО БУТИ» (єдиний інтерфейс оператора з реалтайм-доставкою) та сформульовано постановку задачі проєктування.

Другий розділ «Проєктування універсального месенджера omnichat-hub» складається з підрозділів: «Основні вимоги до системи», «Розробка проєктних специфікацій», «UML use-case діаграми», «Розробка поведінкових UML-діаграм», «Розробка діаграм потоків даних (DFD)», «Проєктування архітектури системи», «Розробка структурних UML-діаграм» та «Висновки до розділу 2». Спроєктовано доменну модель з канал-агностичним шаром, ER-схему бази даних на 24 таблиці, OpenAPI-специфікацію контракту як єдине джерело правди, потоки доставки повідомлень через транзакційний Outbox-патерн.

Третій розділ «Розробка універсального месенджера omnichat-hub» складається з підрозділів: «Обґрунтування вибору інструментальних засобів», «Розробка системи» та «Висновки до розділу 3». Обґрунтовано вибір NestJS як backend-фреймворку, PostgreSQL з Prisma ORM як шару даних, Socket.IO з Redis-адаптером як реалтайм-каналу, Next.js з React як веб-клієнта, Expo як мобільного клієнта. Реалізовано REST API на NestJS, фонові воркери BullMQ, інтеграційні модулі для Rozetka, Prom, OLX, транзакційний Outbox-патерн та обгорткове шифрування ключів каналів.

Четвертий розділ «Тестування та впровадження» складається з підрозділів: «Тестування системи», «Робота користувача з системою», «Встановлення та експериментальне застосування» та «Висновки до розділу

4». Розроблене програмне забезпечення пройшло функціональне тестування на рівнях unit (Vitest), integration (Postgres у Docker) та E2E (HTTP і WebSocket). Перевірка підтвердила відповідність системи функціональним вимогам, сформульованим у розділах 1 і 2. Описано процеси встановлення та виробничого розгортання через Pulumi на DigitalOcean App Platform і Vercel.

ABSTRACT

The bachelor's thesis consists of the following sections: an introduction, four chapters, conclusions, a list of references, and appendices.

The first chapter, "Analysis of the Subject Area", provides an overview of the Ukrainian e-commerce market and the communication channels of the Ukrainian marketplaces Rozetka, Prom, and OLX. The chapter analyses existing global and local omnichannel solutions (Sendbird, Front, Intercom, Re:amaze, Crisp, Zoho Desk, KeyCRM, Sitniks). It describes the transition from the "AS IS" model (fragmented channel interfaces) to the "TO BE" model (a unified operator interface with realtime delivery) and sets forth the design task.

The second chapter, "Design of the Universal Messenger omnichat-hub", consists of the following subsections: "Basic Requirements for the System", "Development of Project Specifications", "UML Use-Case Diagrams", "Development of Behavioral UML Diagrams", "Development of Data Flow Diagrams (DFD)", "Design of the System Architecture", "Development of Structural UML Diagrams", and "Conclusions to Chapter 2". A channel-agnostic domain model, an ER schema covering twenty-four tables, an OpenAPI specification as the single source of truth, and the message delivery flows through the transactional Outbox-pattern were designed.

The third chapter, "Development of the Universal Messenger omnichat-hub", consists of the subsections: "Justification for the Choice of Development Tools", "Development of the System", and "Conclusions to Chapter 3". NestJS was justified as the backend framework, PostgreSQL with Prisma ORM as the data layer, Socket.IO with the Redis adapter as the realtime channel, Next.js with React as the web client, and Expo as the mobile client. A NestJS REST API, BullMQ background workers, integration modules for Rozetka, Prom, and OLX, the transactional Outbox-pattern, and envelope encryption of channel keys were implemented.

The fourth chapter, "Testing and Deployment", consists of the subsections: "System Testing", "User Interaction with the System", "Installation and Experimental Deployment", and "Conclusions to Chapter 4". The developed

software underwent functional testing at the unit (Vitest), integration (Postgres in Docker), and E2E (HTTP and WebSocket) levels. The testing confirmed the system's compliance with the functional requirements set forth in Chapters 1 and 2. The processes of installation and production deployment via Pulumi on DigitalOcean App Platform and Vercel are described.

ЗМІСТ

ПЕРЕЛІК УМОВНИХ СКОРОЧЕНЬ

API - Application Programming Interface, прикладний програмний інтерфейс.

APNs - Apple Push Notification service, сервіс push-сповіщень Apple.

BPMN - Business Process Model and Notation, нотація моделювання бізнес-процесів.

CI/CD - Continuous Integration / Continuous Delivery, безперервна інтеграція та доставка.

CORS - Cross-Origin Resource Sharing, спільне використання ресурсів між доменами.

CQRS - Command Query Responsibility Segregation, розділення відповідальності команд і запитів.

CRM - Customer Relationship Management, система управління взаємовідносинами з клієнтами.

DEK - Data Encryption Key, ключ шифрування даних.

DFD - Data Flow Diagram, діаграма потоків даних.

DTO - Data Transfer Object, об’єкт передачі даних.

ER - Entity-Relationship, схема «сутність-зв’язок».

FCM - Firebase Cloud Messaging, сервіс push-сповіщень Google.

FTS - Full-Text Search, повнотекстовий пошук.

GIN - Generalized Inverted Index, узагальнений інвертований індекс PostgreSQL.

HTTPS - HyperText Transfer Protocol Secure, захищений протокол передачі даних.

IaC - Infrastructure as Code, інфраструктура як код.

IV - Initialization Vector, ініціалізаційний вектор шифрування.

JSON - JavaScript Object Notation, текстовий формат обміну даними.

JWT - JSON Web Token, токен авторизації у форматі JSON.

KEK - Key Encryption Key, ключ шифрування ключів.

KMS - Key Management Service, сервіс керування ключами.

LTS - Long-Term Support, версія з довготривалою підтримкою.

MVP - Minimum Viable Product, мінімально життєздатний продукт.

NPM - Node Package Manager, менеджер пакетів Node.js.

OAuth - Open Authorization, відкритий стандарт авторизації.

ORM - Object-Relational Mapping, об’єктно-реляційне відображення.

OTP - One-Time Password, одноразовий пароль.

REST - Representational State Transfer, архітектурний стиль для веб-сервісів.

RPS - Requests Per Second, запитів на секунду.

SaaS - Software as a Service, програмне забезпечення як послуга.

SDK - Software Development Kit, набір засобів розробки.

SQL - Structured Query Language, мова структурованих запитів.

SSO - Single Sign-On, єдиний вхід.

SSR - Server-Side Rendering, рендеринг на стороні сервера.

TLS - Transport Layer Security, протокол безпеки транспортного рівня.

UML - Unified Modeling Language, уніфікована мова моделювання.

БД - база даних.

ОКР - освітньо-кваліфікаційний рівень.

ОС - операційна система.

ПЗ - програмне забезпечення.

ПЗіТ - кафедра програмних засобів і технологій.

ПО - предметна область.

ФОП - фізична особа-підприємець.

ХНТУ - Херсонський національний технічний університет.

ВСТУП

Сучасний український малий та середній бізнес у сфері електронної комерції активно використовує одночасно декілька каналів збуту: маркетплейси (Rozetka, Prom, OLX, Allo, Hotline), соціальні мережі (Instagram, Facebook, TikTok Shop), власні веб-вітрини та месенджери (Telegram, Viber, WhatsApp). Кожен з цих каналів пропонує власний інтерфейс для комунікації з покупцями: окремий веб-кабінет, окремий мобільний застосунок, окреме сповіщення. Це призводить до низки практичних проблем: контекст спілкування з клієнтом втрачається при перемиканні між каналами, час відповіді зростає, виникають помилки персоналу, неможливо побудувати зведену CRM-аналітику по конверсії «канал - лід - продаж».

Існуючі світові SaaS-рішення класу омніканального інбоксу (Sendbird, Front, Intercom, Re:amaze, Crisp, Zoho Desk) орієнтовані переважно на західні ринки і не підтримують українські маркетплейси як нативні канали. Українські аналоги (KeyCRM, Sitniks, Salesdrive) є повностековими CRM-системами з чатом як одним з модулів, що ускладнює користувацький досвід для замовників, яким потрібний лише уніфікований інбокс. Зазначене робить актуальною розробку спеціалізованого омніканального месенджера для українського ринку, що штатно інтегрується з провідними маркетплейсами Rozetka, Prom, OLX, працює як SaaS без необхідності для клієнта розгортати інфраструктуру, забезпечує надійну реалтайм-доставку повідомлень та горизонтальне масштабування.

Актуальність теми зумовлена стрімким зростанням ринку електронної комерції в Україні та підвищенням вимог користувачів до швидкості, зручності й персоналізації обслуговування. За результатами дослідження ринку e-commerce України, у 2025 році понад 73 відсотки продавців-фізосіб та малих ФОП ведуть продажі мінімум на трьох каналах одночасно [30], а перевищення часу відповіді на запит покупця понад 30 хвилин знижує ймовірність продажу на 40-60 відсотків. Збільшення кількості запитів та обсягів даних ускладнює процес взаємодії між оператором і клієнтом, що

підвищує навантаження на персонал служб підтримки та знижує ефективність роботи з традиційними роздробленими інтерфейсами каналів. У цих умовах розробка локалізованої омніканальної SaaS-платформи omnichat-hub є практично доцільним рішенням, оскільки дозволяє автоматизувати агрегацію вхідних повідомлень з усіх каналів, забезпечити операторам оперативний доступ до історії спілкування та підвищити рівень обслуговування клієнтів, що сприяє підвищенню конверсії продажів та конкурентоспроможності українського малого і середнього бізнесу.

Мета і задачі дослідження. Метою кваліфікаційної роботи бакалавра є розробка SaaS-платформи omnichat-hub для агрегації комунікацій з українськими маркетплейсами Rozetka, Prom, OLX у єдиний інтерфейс оператора з підтримкою реалтайм-доставки повідомлень, гарантованої доставки at-least-once [6] та горизонтального масштабування.

Для досягнення поставленої мети необхідно вирішити такі **задачі**:

1. проаналізувати ринок omnichannel-рішень, визначити прогалини на українському сегменті e-commerce та сформулювати функціональні вимоги до omnichat-hub;
2. дослідити API українських маркетплейсів OLX, Rozetka, Prom, моделі автентифікації (потік OAuth 2.0 Authorization Code для OLX, статичні API-токени з кабінету продавця для Rozetka та Prom) та обмеження частоти запитів;
3. розробити канал-агностичну доменну модель та ER-схему бази даних, що абстрагує канал-специфічні особливості та забезпечує багатотенантну ізоляцію даних на рівні workspace;
4. реалізувати REST API на основі NestJS [11] з версіонуванням v1, OpenAPI 3.1 як єдиним джерелом правди контракту [18] та транзакційним Outbox-патерном для гарантованої доставки подій;
5. реалізувати реалтайм-доставку повідомлень через Socket.IO з Redis-адаптером для горизонтального масштабування інстансів API;

6. створити веб-клієнт оператора на основі Next.js 16 та React 19 [20] і мобільний клієнт на основі Expo SDK 53 (React Native) з push-сповіщеннями;
7. розгорнути omnichat-hub у виробничому середовищі через Pulumi на платформах DigitalOcean App Platform та Vercel з підключенням моніторингу через Sentry та Grafana Cloud;
8. провести функціональне тестування системи на рівнях unit, integration та E2E і скласти технічну документацію розробленого ПЗ.

Об’єкт дослідження: процеси обміну повідомленнями між продавцями та покупцями на українських онлайн-платформах продажів (на прикладі маркетплейсів OLX, Rozetka, Prom).

Предмет дослідження: методи та засоби розробки веб-системи омніканальної агрегації комунікаційних каналів з підтримкою реалтайм-доставки повідомлень.

Методи дослідження. У роботі застосовано системний аналіз для виявлення прогалин ринку omnichannel-рішень та формування функціональних вимог; об’єктно-орієнтоване моделювання предметної області [3] з побудовою UML-діаграм класів та послідовності; патерни enterprise-розробки, серед яких транзакційний Outbox [1], логічне розділення команд і запитів (read/write шляхів) без окремого CQRS-фреймворку та channel-agnostic adapter для канал-специфічних інтеграцій; функціональне тестування на рівнях unit, integration та E2E з використанням Vitest, Postgres у Docker та Playwright; методи проєктування реляційних баз даних із застосуванням повнотекстового пошуку на tsvector та GIN-індексах PostgreSQL.

Наукова новизна одержаних результатів полягає у такому:

- запропоновано методологічну рамку уніфікації трьох різнорідних моделей авторизації українських маркетплейсів через канал-агностичний адаптер: потоку OAuth 2.0 Authorization Code з оновлюваним refresh-токеном для OLX і статичних API-токенів з

кабінету продавця для Rozetka та Prom, що зведені до єдиної абстракції IChannelAdapter з ізоляцією канал-специфічних особливостей від доменного шару omnichat-hub. Наукова новизна полягає не у факті інтеграції з маркетплейсами, а у способі уніфікації гетерогенних моделей авторизації та форматів повідомлень на рівні єдиного контракту адаптера, що дозволяє додавати нові канали без зміни бізнес-логіки ядра;

- удосконалено механізм гарантованої доставки повідомлень за рахунок поєднання транзакційного Outbox-патерну з горизонтально-масштабованим Socket.IO Redis-адаптером, що забезпечує доставку at-least-once навіть за умов збоїв воркерів та дозволяє додавати інстанси API без втрати реалтайм-зв’язку з клієнтами;
- запропоновано модель обгорткового шифрування (envelope encryption) credentials каналів з Data Encryption Key (DEK) на рівні workspace, що дозволяє виконувати ротацію ключів без перешифрування існуючих записів.

Практичне значення одержаних результатів. Розроблена SaaS-платформа omnichat-hub є основою продукту для українських ФОПів та малого і середнього бізнесу у сфері e-commerce і дозволяє об’єднати комунікації з провідних українських маркетплейсів у єдиному операторському дашборді. Очікуваний ефект від впровадження: скорочення часу відповіді клієнтам та відповідне підвищення конверсії продажів. Здатність до виробничого розгортання продемонстровано описаним у роботі процесом розгортання через Pulumi у середовищі DigitalOcean App Platform і Vercel з підключенням моніторингу через Sentry та Grafana Cloud. Архітектурні підходи (транзакційний Outbox, channel-agnostic domain layer, OpenAPI як єдине джерело правди, Redis-адаптер Socket.IO) можуть бути перенесені на інші проєкти у сфері інтеграції розподілених систем.

Апробація результатів дослідження. Основні результати кваліфікаційної роботи отримані під час проходження виробничої

(переддипломної) практики на базі ФОП Герецький Віктор Вікторович у період з 20.04.2026 по 17.05.2026 в обсязі 180 годин відповідно до навчального плану ХНТУ для здобувачів першого (бакалаврського) рівня вищої освіти спеціальності 121 «Інженерія програмного забезпечення». Архітектура та практична реалізація omnichat-hub описані у звіті з виробничої практики; підсумкові результати винесено на захист кваліфікаційної роботи 24.06.2026.

Структура і обсяг роботи. Кваліфікаційна робота складається з вступу, чотирьох розділів, висновків, списку використаних джерел та двох додатків. Перший розділ містить аналіз предметної області omnichannel-комунікацій у сфері українського e-commerce та постановку задачі проєктування. Другий розділ присвячено проєктуванню omnichat-hub: розробці доменної моделі, ER-схеми бази даних, use-case діаграм, діаграм потоків даних та структурних UML-діаграм. Третій розділ описує обґрунтування вибору інструментальних засобів та практичну розробку системи. Четвертий розділ висвітлює функціональне тестування та впровадження omnichat-hub у виробничому середовищі. Загальний обсяг роботи становить 90 сторінок основного тексту, 21 таблицю, 51 рисунок, 30 джерел у списку використаних джерел.

РОЗДІЛ 1. АНАЛІЗ ПРЕДМЕТНОЇ ОБЛАСТІ

1.1. Опис та загальна характеристика предметної області

Предметною областю роботи є електронна комерція України, а саме процеси комунікації між продавцями та покупцями на цифрових платформах онлайн-продажів. Ринок є фрагментованим: Rozetka займає близько 35% транзакцій, Prom.ua - близько 25%, OLX - близько 20%, Allo - близько 10%, решта 10% припадає на інші майданчики. Структуру ринку подано на рис. 1.1.

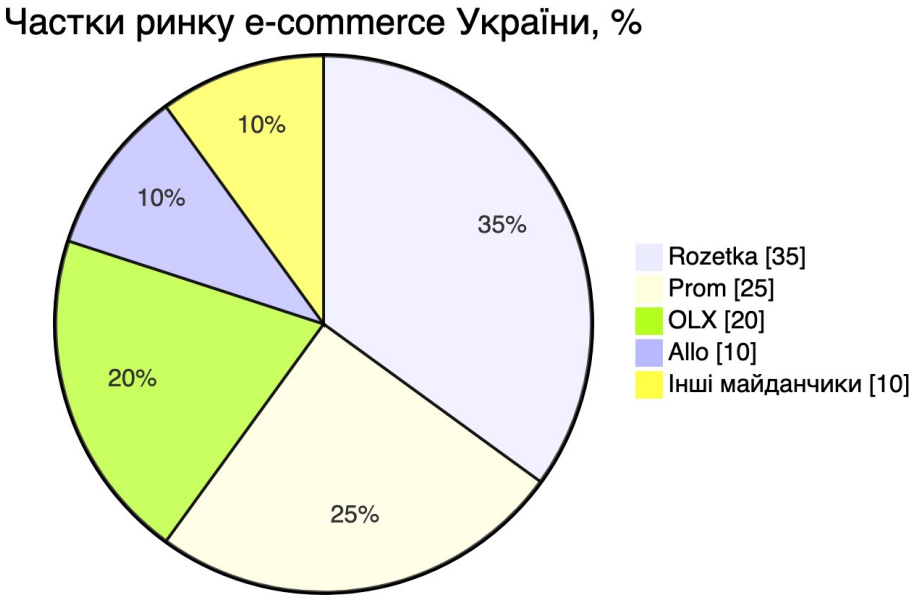


Рисунок 1.1. Структура українського ринку e-commerce за обсягом транзакцій

Головна особливість предметної області - багатоканальність продавця. У 2025 році 73% продавців-фізичних осіб і малих ФОП продають мінімум на трьох каналах одночасно [30]; для середнього бізнесу з 500-5000 артикулів типовими є 5-7 каналів. До маркетплейсів додаються Instagram, Facebook, Telegram, Viber, пошта і власні вітрини. Це розширює охоплення аудиторії, але розділяє комунікацію між непов’язаними кабінетами.

Типовий оператор магазину з 50-100 замовленнями на добу одночасно працює з Rozetka, Prom, OLX, Instagram Direct, Telegram, поштою або CRM. За день він переключається між інтерфейсами 200-400 разів, витрачаючи 5-15 секунд на кожне переключення. Поточний процес показано на рис. Б.1: оператор вручну контролює окремі джерела, а між ними немає спільного стану діалогу.

Фрагментація породжує чотири системні проблеми. По-перше, втрачається контекст: покупець, який почав діалог в OLX і продовжив у Telegram, для оператора виглядає як новий клієнт. По-друге, зростає час відповіді: перевищення 30 хвилин знижує конверсію оголошення у продаж на 40-60% [30]. По-третє, частина каналів залишається без уваги через слабкі або несвоєчасні push-сповіщення. По-четверте, неможливо побудувати єдину CRM-аналітику «канал - лід - продаж», бо кожен майданчик зберігає статистику окремо.

Таблиця 1.1. Приблизні обсяги повідомлень у середньому українському інтернет-магазині

Канал	**Повідомлень на добу**	**Унікальних покупців на добу**	**Середній час відповіді без агрегації**
Rozetka	80-150	40-80	25-45 хв
Prom	60-120	30-60	30-50 хв
OLX	50-100	25-50	40-70 хв
Соцмережі та інше	30-80	15-40	60-120 хв
Разом	220-450	110-230	30-60 хв

За 2-3 хвилини на повідомлення такий обсяг дає 7-22 години чистого часу оператора на добу. Один оператор фізично не покриває процес, тому магазин або наймає додаткових співробітників, або втрачає звернення та продажі.

Особливо проблемними є звернення, що потребують швидкого уточнення: наявність товару, характеристики, доставка, оплата, торг на OLX, претензії, повернення та повторні звернення. Вони типові й частково шаблонізуються, але без єдиного інтерфейсу оператор щоразу витрачає час не

на відповідь, а на пошук вкладки, авторизацію, відновлення історії та зіставлення звернення з товаром. У великих магазинах із 5-15 операторами ця проблема масштабується: менеджер не бачить, який канал створює найбільше лідів, де порушується SLA і які типові запитання потрібно винести у шаблони.

Технічно канали також неоднорідні. Rozetka надає REST API зі статичним API-токеном з кабінету продавця [27], Prom використовує аналогічну модель зі статичним токеном [29], а OLX підтримує OAuth 2.0 Authorization Code з refresh-токенами [7]. Відрізняються формати запитів, пагінація, обмеження, webhooks і правила автентифікації. Самостійна інтеграція всіх каналів для одного малого продавця економічно невиправдана.

На систему впливають також нормативні та локальні вимоги. Персональні дані покупців підпадають під Закон України «Про захист персональних даних», тому потрібні шифрування at-rest, аудит і контроль видалення даних. Локалізація має враховувати українську та російську мови з помилками, гривню, телефонний формат +380, дати «дд.мм.рррр» і часовий пояс Europe/Kyiv. Отже, предмет проєктування - омніканальна платформа omnichat-hub, яка агрегує повідомлення з українських маркетплейсів у єдиний інтерфейс оператора.

1.2. Моделювання та аналіз бізнес-процесів предметної області

Для формалізації процесів побудовано модель поточного стану «ЯК Є» (AS-IS) і модель бажаного стану «ЯК ПОВИННО БУТИ» (TO-BE). Основною методологією обрано SADT/IDEF0: робота подається функціональним блоком, а взаємодія описується стрілками ICOM - Input, Control, Output, Mechanism. Для часової послідовності використано IDEF3, а BPMN-моделі додано як допоміжний погляд на доріжки учасників.

1.2.1. Побудова комплексу моделей «ЯК Є»

Поточна модель обробки повідомлень є децентралізованою: кожен канал обробляється окремо, без спільного накопичувача даних і без єдиного стану діалогу. Точкою зору моделювання є оператор магазину, метою - виявлення вузьких місць у щоденній комунікації.

Контекстна діаграма IDEF0 процесу «ЯК Є» (рис. Б.2) описує роботу «Обробити звернення покупців у багатоканальному середовищі». На вхід надходять звернення покупців і контекст товару або оголошення; вихід - відповіді покупцям і розрізнені звіти каналів. Керуванням є регламент часу відповіді, правила автентифікації каналів і законодавчі вимоги; механізмами - оператор, окремі кабінети маркетплейсів і push-сповіщення.

Декомпозиція рівня A0 (рис. Б.3) містить чотири роботи: A1 «Виявити нове звернення в каналі», A2 «Відновити контекст діалогу вручну», A3 «Сформулювати та надіслати відповідь», A4 «Сформувати звітність по каналах». Ключовий дефект моделі: A1 виконується незалежно для кожного каналу, а A4 не отримує даних зі спільного джерела, тому аналітика лишається ручною.

Роботу A2 деталізовано на другому рівні (рис. Б.4), оскільки саме вона створює найбільші втрати часу. Оператор ідентифікує покупця за локальним іменем або нікнеймом, шукає попередню історію тільки в межах одного каналу та зіставляє звернення з оголошенням. Спільного ключа покупця між каналами немає, тому звернення тієї самої людини з іншого каналу сприймається як новий діалог.

IDEF3-сценарій (рис. Б.5) фіксує точку втрати контексту: покупець пише у канал A, оператор відповідає, після переходу покупця в канал B оператор не зв’язує ці події й починає діалог наново. Менеджер магазину також не бачить повної картини та змушений вручну порівнювати звіти маркетплейсів.

У межах IDEF0 це проявляється як відсутність спільного накопичувача між роботами A1-A4: A1 виявляє звернення тільки у локальному каналі, A2 відновлює контекст вручну, A3 надсилає відповідь у тому самому кабінеті, а

A4 формує звіт без автоматичного входу з попередніх робіт. Тобто в моделі «ЯК Є» інформація не рухається як керований потік даних, а залишається розпорошеною між інтерфейсами маркетплейсів і пам’яттю оператора.

1.2.2. Аналіз недоліків існуючої бізнес-моделі

Аналіз моделі «ЯК Є» виявив чотири недоліки. Перший - відсутність спільного ключа ідентифікації покупця між каналами, що дублює роботи A2 і A3 та спричиняє втрату контексту. Другий - неефективний рух інформації до звітності: робота A4 не має входу зі спільного накопичувача, тому неможлива автоматична аналітика «канал - лід - продаж». Третій - відсутність зворотного зв’язку за пріоритетами в A1: канали зі слабкими сповіщеннями залишаються без уваги. Четвертий - накладні витрати на 200-400 переключень інтерфейсів щодня.

Сукупний ефект недоліків - зростання часу відповіді, падіння конверсії на 40-60% при затримці понад 30 хвилин [30], втрата частини звернень, погіршення якості сервісу та відсутність керованої аналітики. Усунення проблеми потребує не покращення окремого кабінету, а переходу до спільного накопичувача стану та єдиного інтерфейсу оператора.

1.2.3. Побудова комплексу моделей «ЯК ПОВИННО БУТИ»

Модель TO-BE вводить omnichat-hub як проміжний шар між маркетплейсами та оператором. Система отримує повідомлення через API каналів, нормалізує їх у спільні сутності Conversation і Message, зберігає у PostgreSQL, доставляє оператору в real-time і формує зведену аналітику.

Контекстна IDEF0-діаграма «ЯК ПОВИННО БУТИ» (рис. Б.6) зберігає ті самі входи й виходи, але змінює механізми: замість окремих кабінетів працюють канал-агностичні адаптери, PostgreSQL, транзакційний Outbox, Socket.IO з Redis-адаптером і єдиний web/mobile-інтерфейс. До керування додаються правила доменного шару (computeNeedsReply, snooze, auto-archive) і політика at-least-once delivery.

Декомпозиція TO-BE (рис. Б.7) містить п’ять робіт: A1 «Синхронізувати повідомлення з каналів», A2 «Нормалізувати та зберегти у спільну БД», A3 «Доставити повідомлення оператору в реальному часі», A4 «Сформулювати та надіслати відповідь», A5 «Сформувати зведену аналітику». Принципова відмінність від AS-IS - спільна БД, яка є джерелом істини для інбоксу, історії та аналітики.

Роботу A3 деталізовано окремо (рис. Б.8), бо вона реалізує гарантовану доставку. Подія записується в Outbox у тій самій транзакції, що й повідомлення, потім поллер і процесор захоплюють її через оптимістичне умовне оновлення з lockedAt і lockId. Подія у статусі PROCESSING, чий lockedAt старший за OUTBOX_LOCK_TTL_SEC (типово 300 секунд), повторно обробляється. Стани OutboxEvent: PENDING, PROCESSING, PROCESSED, FAILED.

IDEF3-сценарій TO-BE (рис. Б.9) показує два паралельні шляхи після COMMIT: негайне додавання job у BullMQ і періодичний поллер відновлення кожні 60 секунд. Після захоплення події є дві сторони доставки: WebSocket (wsDeliveredAt) і push (pushDeliveredAt). Тільки після обох подія переходить у PROCESSED. Така схема дає at-least-once delivery [1; 6], а дублікати на клієнті відсіюються за outboxEventId.

Для усунення втрати cross-channel контексту передбачено спільне зберігання атрибутів покупця у Conversation, зокрема buyerPhone, і перспективну концепцію canonical contact. Базове зіставлення за телефоном або email віднесено до пріоритету Could, а повна евристика з машинним навчанням - до наступних ітерацій; однак сама поява спільного сховища вже усуває структурну причину недоліку AS-IS.

TO-BE також додає real-time inbox через Socket.IO з Redis-адаптером [14], агреговану аналітику, шаблони відповідей, теги, статуси діалогів, мобільний клієнт на Expo SDK 53 [19] і захищене зберігання токенів через envelope encryption. Очікуваний ефект подано в табл. 1.2.

Важливо, що ці функції змінюють не лише інтерфейс, а й саму організацію процесу. Оператор перестає бути «маршрутизатором» між кабінетами й працює з єдиною чергою діалогів; менеджер отримує дані не постфактум з розрізнених звітів, а з тієї самої БД, з якої працює інбокс; розробник може додавати канали через контракт адаптера, не змінюючи доменну логіку повідомлень.

Таблиця 1.2. Порівняння часу відповіді «ЯК Є» проти «ЯК ПОВИННО БУТИ»

Параметр	**AS-IS**	**TO-BE**	**Покращення**
Час до прочитання	25-40 хв	1-3 хв	у 10-20 разів
Час від прочитання до відповіді	5-15 хв	2-7 хв	у 2-3 рази
Сумарний час відповіді	30-60 хв	5-10 хв	у 6 разів
«Забуті» повідомлення на добу	8-20	0-2	у 10 разів
Переключення вкладок	30-90 хв	0 хв	усунено
Зведена аналітика	відсутня	dashboard	якісне покращення

BPMN-моделі (рис. Б.10, Б.11) підтверджують висновок IDEF0/IDEF3: у AS-IS канали обробляються ізольовано, а в TO-BE сходяться в omnichat-hub як єдине джерело істини для інбоксу оператора та dashboard менеджера.

1.3. Постановка задачі проєктування

Аналіз предметної області доводить потребу в інформаційній системі, що агрегує повідомлення з українських маркетплейсів, зменшує час відповіді та забезпечує керованість комунікацій. Постановка задачі включає огляд аналогів, позиціонування omnichat-hub і формулювання вимог.

1.3.1. Огляд існуючих омніканальних рішень

Світові SaaS-платформи розв’язують суміжні задачі, але не покривають українські маркетплейси. Sendbird орієнтований на вбудований чат у власному застосунку, а не на агрегацію OLX/Rozetka/Prom; ціна стартує від 399 доларів США на місяць. Front підтримує email, чат, SMS і SLA для support-команд, але не має marketplace-конекторів; ціна 25-99 доларів за оператора. Intercom працює як customer messaging для сайтів, Re:amaze фокусується на Shopify/BigCommerce/WooCommerce, Crisp підтримує live-chat і соцмережі за 25-95 євро на місяць, Zoho Desk є частиною великого helpdesk-стека. Жодне з цих рішень не підтримує Rozetka, Prom і OLX нативно.

Ключова причина непридатності світових платформ полягає не тільки в ціні, а й у моделі інтеграції. Вони оптимізовані для власного сайту, email/support-черги або глобальних соцмереж, тоді як український продавець отримує ліди саме з локальних маркетплейсів з різними правилами доступу, форматами діалогів і обмеженнями API. Написання кастомних конекторів до кожної платформи перекладає витрати інтеграції на продавця, що суперечить економіці малого бізнесу.

Український ринок зайнятий CRM-системами. KeyCRM підтримує Rozetka, Prom, OLX, Etsy, eBay, Amazon, Allo, Shopify, WooCommerce, Opencart, Prestashop, Instagram, Facebook Messenger, Telegram, Viber, WhatsApp, TikTok і email; має товари, замовлення, склад, фінанси, аналітику, ролі та мобільний клієнт, ціна - від 19 доларів США на місяць. Sitniks підтримує Instagram, TikTok, Facebook, Prom, Rozetka, Нову Пошту, LiqPay, Rozetka Pay і Plata by Mono, має фіскалізацію, склад і платежі, ціна - від 40 доларів США на місяць. RemOnline, Salesdrive, OneBox і Bitrix24 також покривають частину CRM-задач, але комунікація з маркетплейсами не завжди є core feature.

Через це конкуренція відбувається не за сам факт існування CRM, а за фокус продукту. Повноцінна CRM корисна бізнесу, який уже веде склад, замовлення, платежі й фінансовий облік у єдиній системі. Невеликий

продавець, який має лише проблему розрізнених повідомлень, часто не готовий переносити весь операційний процес у велику CRM. Для нього важливі швидкий старт, мінімальна конфігурація, зрозумілий інбокс і низька когнітивна складність.

Висновок: KeyCRM і Sitniks мають ширше функціональне покриття, але позиціонуються як повні CRM. За публічною інформацією не видно, що вони роблять акцент на Outbox-first delivery, OpenAPI-first контроль розбіжностей схеми у CI або архітектурно ізольовані канал-адаптери. Це не доводить відсутність таких механізмів у закритому коді, але визначає технічну нішу omnichat-hub.

1.3.2. Позиціонування omnichat-hub у конкурентному ландшафті

omnichat-hub не є заміною KeyCRM. Це chat-first продукт для продавців, яким потрібен швидкий, простий і надійний inbox без складського, фінансового та повного CRM-обліку. Його цінність - вузький фокус на комунікації: три українські маркетплейси, real-time доставка, мобільні push-сповіщення, захищені токени, зрозумілий UX і керована архітектура.

Архітектурно це означає, що ядро системи не прив’язується до сутності «замовлення» як центральної, а будується навколо Conversation, Message, ChannelAccount і Workspace. Така модель зменшує складність MVP, залишає можливість інтеграції з CRM у майбутньому та дозволяє додавати нові канали через адаптери, не переписуючи інбокс і доменні правила.

Матрицю порівняння наведено на рис. Б.12. Західні рішення не мають нативних каналів Rozetka/Prom/OLX; KeyCRM має найширший CRM-набір; omnichat-hub свідомо не включає склад і фінанси, але робить inbox, OpenAPI-first і Outbox-first центральними властивостями. Архітектурні переваги: транзакційний Outbox-pattern [1], Socket.IO з Redis-адаптером [14], канал-агностична доменна модель [4], OpenAPI 3.1 як джерело правди контракту [18].

1.3.3. Функціональні вимоги

На основі аналізу сформульовано функціональні вимоги: підключення OLX через OAuth 2.0 Authorization Code [7], Rozetka через статичний токен [27], Prom через статичний токен [29]; єдиний real-time inbox у web-клієнті Next.js 16 + React 19 та mobile-клієнті Expo SDK 53; гарантована at-least-once delivery через Outbox; шифрування API-токенів і OAuth credentials через envelope encryption; мультитенантність з ролями OWNER, ADMIN, MEMBER; повнотекстовий пошук через PostgreSQL tsvector, GIN і pg_trgm [9]; шаблони відповідей, теги, статуси, snooze, pin та архівування діалогів.

1.3.4. Нефункціональні вимоги

Ключові нефункціональні вимоги: end-to-end p99 latency до 800 мс; доступність 99.5%; TLS 1.2+ для мережевих з'єднань; JWT access 15 хв і refresh 30 днів з ротацією [8]; Email-OTP як bootstrap-автентифікація; Helmet, CORS allowlist і rate limiting; заборона логування Authorization, *token*, *secret*; горизонтальне масштабування API через Redis-адаптер Socket.IO; масштабування воркерів кількома BullMQ-консьюмерами [15]; вертикальне масштабування PostgreSQL у DigitalOcean Managed Postgres [25] з потенціалом read-replicas; спостережуваність через Sentry [23] і Grafana Cloud [24]. Цільовий load 1000 RPS на POST /v1/messages є архітектурним таргетом, навантажувальне тестування заплановано на третій квартал 2026 року.

Окремо зафіксовано межі MVP: AI-suggested replies, Telegram, Instagram і Facebook не входять до кваліфікаційної роботи, хоча архітектура передбачає їх як майбутні канал-адаптери. Це дозволяє не розмивати обсяг розробки й зосередитися на трьох основних маркетплейсах, які безпосередньо відповідають предметній області.

Концептуальну архітектуру показано на рис. Б.13: OLX, Rozetka і Prom інтегруються через API та адаптери; backend на NestJS записує дані у PostgreSQL і Outbox; Redis використовується для кешу, Pub/Sub і BullMQ;

Socket.IO доставляє події у web/mobile; Sentry і Grafana Cloud забезпечують моніторинг.

Таблиця 1.3. Перелік функціональних та нефункціональних вимог до omnichat-hub з пріоритизацією MoSCoW

№	Категорія	Вимога	Пріоритет
1	Функціональна	Підключення OLX через OAuth 2.0	Must
2	Функціональна	Підключення Rozetka через статичний токен	Must
3	Функціональна	Підключення Prom через статичний токен	Must
4	Функціональна	Real-time inbox через Socket.IO	Must
5	Функціональна	At-least-once delivery через Outbox-pattern	Must
6	Функціональна	Envelope encryption ключів каналів	Must
7	Функціональна	Web-клієнт Next.js + React	Must
8	Функціональна	Mobile-клієнт Expo з push	Should
9	Функціональна	Workspace-isolation	Must
10	Функціональна	Повнотекстовий пошук	Should
11	Функціональна	Шаблони, теги, статуси	Should
12	Функціональна	Canonical contact між каналами	Could
13	Функціональна	Зведена аналітика менеджера	Should
14	Функціональна	AI-suggested replies	Won't (Q4 roadmap)
15	Функціональна	Telegram, Instagram, Facebook	Won't (Q3 roadmap)
16	Нефункціональна	p99 latency ≤ 800 мс	Must
17	Нефункціональна	Доступність 99.5%	Should
18	Нефункціональна	TLS 1.2+	Must
19	Нефункціональна	Шифрування at-rest секретів	Must
20	Нефункціональна	Горизонтальне масштабування API	Must
21	Нефункціональна	Цільовий load 1000 RPS	Should
22	Нефункціональна	Структуроване логування pino	Should
23	Нефункціональна	Sentry	Must
24	Нефункціональна	Grafana Cloud	Should
25	Нефункціональна	Infrastructure as Code Pulumi	Should

Із 25 вимог 13 мають пріоритет Must, 9 - Should, 1 - Could, 2 - Won’t. Ці вимоги визначають подальші архітектурні рішення: Outbox для надійності, Socket.IO + Redis для real-time, Workspace + guards для ізоляції, PostgreSQL FTS для пошуку.

1.4. Висновки до розділу 1

У розділі 1 проаналізовано предметну область українського e-commerce і доведено потребу в omnichat-hub. Ринок фрагментований між Rozetka, Prom, OLX, Allo та іншими майданчиками; 73% малих ФОП продають щонайменше на трьох каналах [30]. Типовий оператор обробляє 220-450 повідомлень на добу, переключається між інтерфейсами 200-400 разів і працює з часом відповіді 30-60 хвилин, тоді як затримка понад 30 хвилин знижує конверсію на 40-60%.

У підрозділі 1.2 побудовано моделі AS-IS і TO-BE у нотаціях IDEF0 та IDEF3, а також допоміжні BPMN-моделі. AS-IS виявляє фрагментацію каналів, втрату cross-channel контексту, ручну звітність і надмірні переключення. TO-BE усуває ці недоліки через спільну БД, єдиний real-time inbox, Outbox-патерн, канал-адаптери та зведену аналітику. Очікуване покращення: час до прочитання у 10-20 разів, сумарний час відповіді у 6 разів, повне усунення часу на переключення вкладок.

У підрозділі 1.3 розглянуто світові SaaS-платформи і локальні CRM-рішення. Світові платформи не підтримують українські маркетплейси нативно, а локальні CRM (передусім KeyCRM і Sitniks) покривають ширший функціонал, але є full-stack CRM. omnichat-hub позиціонується як chat-first альтернатива для продавців, яким потрібна швидка та надійна комунікація без складського і фінансового обліку. Сформульовано 25 вимог із MoSCoW-пріоритизацією.

Обмеження аналізу: частки ринку й обсяги повідомлень є оцінками, точні обороти каналів не публікуються, а внутрішня архітектура конкурентів є закритою. Попри це, моделі процесів, огляд ринку та вимоги формують достатню основу для проєктування, реалізації та тестування omnichat-hub у наступних розділах.

РОЗДІЛ 2. ПРОЄКТУВАННЯ УНІВЕРСАЛЬНОГО МЕСЕНДЖЕРА omnichat-hub

2.1. Основні вимоги до системи

На основі аналізу предметної області, ринку омніканальних месенджерів та API українських маркетплейсів сформовано перелік вимог до omnichat-hub. Вимоги поділено на функціональні та нефункціональні згідно зі стандартом ISO/IEC 25010.

Функціональні вимоги: (1) агрегація повідомлень з усіх підключених каналів у єдиний інбокс, відсортований за датою останньої активності; (2) підключення нових каналів власником робочого простору (для OLX потік OAuth 2.0 Authorization Code [7], для Rozetka і Prom введення статичного API-токена з кабінету продавця); (3) обмін повідомленнями у режимі реального часу з появою вхідних не пізніше однієї секунди (Socket.IO та pub-sub шар Redis [16]); (4) підтримка вихідних відповідей з коректним статусом доставки (SENDING, SENT, DELIVERED, READ, FAILED); (5) управління станом розмови (needsReply, архівування, snooze, pin, теги, статуси); (6) повнотекстовий пошук по історії з підтримкою опечаток [9]; (7) багатотенантна ізоляція робочих просторів; (8) ролі (OWNER, ADMIN, MEMBER) і запрошення учасників; (9) push-сповіщення на мобільний пристрій; (10) dashboard з агрегованими метриками за день.

Нефункціональні вимоги: (1) продуктивність end-to-end latency p99 не більше 800 мс; (2) масштабованість до 1000 RPS за рахунок горизонтального масштабування (навантажувальне тестування k6/Artillery заплановане на третій квартал 2026 року, наразі виконані лише ad-hoc заміри); (3) надійність at-least-once для кожного повідомлення [6] з архітектурним таргетом 0 втрат з 600 000 подій (верифікація Q3 2026); (4) безпека: TLS 1.3, відсутність паролів (email-OTP та Google OAuth), envelope encryption токенів на базі AES-256-GCM з власним DEK на робочий простір; (5) сумісність із Chrome, Firefox,

Safari, Edge (Last 2 versions); (6) мобільні платформи iOS 15+, Android 9+; (7) локалізація українською, гривні, формат «дд.мм.рррр»; (8) спостережуваність через Sentry [23] і Grafana Cloud (Loki, Prometheus) [24]; (9) підтримка щонайменше трьох каналів (OLX, Rozetka, Prom), розширення на Telegram/Instagram/Messenger у roadmap Q3 2026; (10) багатотенантність на рівні усіх REST-ендпоінтів та Socket.IO-кімнат (workspaceId перевіряється guard у NestJS, scoped-запитами Prisma, ізоляцією Redis-каналів).

2.2. Розробка проєктних специфікацій

Проєктні специфікації формалізують вимоги у вимірювані параметри з чіткими метриками прийому (acceptance criteria), які можна автоматизувати у CI-пайплайні або підтвердити інструментально. Зведену таблицю наведено в табл. 2.1.

Таблиця 2.1. Проєктні специфікації omnichat-hub

№	Назва	Опис	Метрика прийому
S-01	Інтеграція з OLX	Підключення облікового запису OLX через OAuth 2.0 потік Authorization Code з підтримкою refresh-токенів	Успішне отримання access та refresh токенів, рівень помилок при підключенні менше 1%
S-02	Інтеграція з Rozetka	Підключення кабінету Rozetka через статичний API-токен, який користувач отримує в кабінеті продавця Rozetka	Валідний токен зберігається у зашифрованому вигляді, наступний sync виконується протягом 60 с
S-03	Інтеграція з Prom	Підключення кабінету Prom через статичний API-токен з кабінету продавця Prom.ua	Валідний токен зберігається у зашифрованому вигляді, інкрементальна синхронізація стартує
S-04	Латентність доставки	Час повідомлення «маркетплейс - клієнт оператора» включає інкрементальний sync, обробку, broadcast	p99 ≤ 800 мс на production-середовищі (архітектурний таргет, верифікація Q3 2026)

Продовження табл. 2.1

№	Назва	Опис	Метрика прийому
S-05	Гарантована доставка	Кожна стороння подія повинна бути доставлена реалтайм-клієнту або зафіксована у Outbox для повторної спроби	0 з 600 000 згенерованих подій загублених (архітектурний таргет, верифікація Q3 2026)
S-06	Інкрементальний sync	Полінг кожного каналу виконується раз на 60 секунд із використанням checkpoint (cursor або timestamp)	Дублі повідомлень виключаються через unique constraint (conversationId, externalMessageId)
S-07	Підтримка браузерів	Web-клієнт працює у Chrome, Firefox, Safari, Edge	Тестове покриття Last 2 versions кожного з браузерів проходить E2E-набір Playwright
S-08	Підтримка mobile	iOS 15+, Android 9+ через Expo SDK 53	Базовий E2E flow логіну та обміну повідомленнями проходить на симуляторах обох платформ
S-09	Криптографічна стійкість	Усі токени каналів та інші чутливі credentials шифруються алгоритмом AES-256-GCM	KEK + DEK ротуються, тести розшифрування на старих версіях DEK проходять
S-10	Багатотенантність	Ізоляція робочого простору на рівні усіх REST-ендпоінтів та Socket.IO-кімнат	Жоден запит без workspaceId у JWT-payload не повертає дані іншого робочого простору
S-11	Pub-Sub масштабування	Socket.IO підтримує горизонтальне масштабування через Redis-адаптер	Подія, опублікована одним інстансом API, доставляється підпискам інших інстансів
S-12	Outbox-патерн	INSERT message, UPDATE conversation, INSERT outbox_event виконуються в одній транзакції БД	Збій воркера не призводить до втрати події: при перезапуску PENDING-запис підхоплюється
S-13	Push-сповіщення	Доставка push-нотифікацій на iOS (Apple APNs) та Android (Firebase Cloud Messaging)	Push отримано на пристрої протягом 5 с після появи нового вхідного повідомлення
S-14	Повнотекстовий пошук	Postgres tsvector + GIN-індекс для FTS та pg_trgm для fuzzy fallback	Пошук по 100 000 повідомлень виконується за ≤ 200 мс
S-15	Логування	Структуроване логування через nestjs-pino з request-id, без виводу секретів	Жодне поле з назвою *token*, *secret*, authorization не з'являється у raw logs

S-01 описує OAuth 2.0 потік для OLX: після підтвердження прав OLX надсилає authorization code на callback, бекенд обмінює його на пару токенів, шифрує refresh-токен через envelope encryption і зберігає у authDataEncrypted таблиці ChannelAccount. S-02 і S-03 регламентують підключення Rozetka і Prom, де OAuth відсутній і користувач сам копіює персональний API-токен з

кабінету продавця [27], [29]; бекенд перевіряє його пробним запитом та зберігає зашифрованим. S-04 наразі перевірено лише ad-hoc замірами (180-250 мс локально, 400-650 мс на staging DigitalOcean Frankfurt); навантажувальне тестування k6/Artillery при 1000 RPS заплановане на Q3 2026. S-05 і S-12 разом описують Outbox-патерн з транзакційним записом, що гарантує незагубленість події між API і воркером [1]. S-11 забезпечує реалтайм-семантику при горизонтальному розгортанні через Redis-адаптер Socket.IO.

2.3. UML use-case діаграми

Актор у нотації UML це сутність (людина, інша система або зовнішній сервіс), що ініціює дії над системою або реагує на її вихідні дії. Для omnichat-hub визначено чотирьох акторів: **Власник робочого простору (Owner)** реєструє робочий простір, оплачує підписку, керує командою, підключає канали; **Адміністратор (Admin)** модерує доступ, переглядає аналітику, налаштовує робочий простір без права його видалити; **Оператор (Operator)** відповідає на повідомлення покупців і керує діалогами (у БД роль MEMBER); **Зовнішній канал (OLX / Rozetka / Prom)** система-джерело повідомлень. Діаграму варіантів використання показано на рисунку Б.14.

Опис ключових use cases наведено у табл. 2.2.

Таблиця 2.2. Опис ключових варіантів використання

Use case	Актор	Precondition	Main flow	Postcondition
ConnectOlxAccount	Owner / Admin	Користувач авторизований, обрав робочий простір	1. Натиснути «Підключити OLX». 2. Перенаправлення на OLX OAuth. 3. Підтвердити доступ. 4. OLX повертає authorization code на callback. 5. Бекенд обмінює code на access+refresh. 6. Токени шифруються і зберігаються	Створено запис ChannelAccount з channel=OLX, authType=OAUTH, isEnabled=true
SyncMessages	External Channel (worker)	ChannelAccount активний, токен валідний, минув 1 хв з останнього sync	1. Worker читає ChannelSyncCheckpoint. 2. Запитує API каналу з cursor. 3. Для кожного нового повідомлення відкриває транзакцію: INSERT Message, UPDATE Conversation, INSERT OutboxEvent. 4. Оновлює checkpoint	Нові Message доступні в інбоксі, OutboxEvent зі статусом PENDING очікує доставки
SendReply	Operator	Розмова відкрита, токен каналу валідний	1. Operator вводить текст. 2. POST /v1/messages. 3. API відкриває транзакцію: INSERT Message direction=OUT, status=SENDING; INSERT OutboxEvent type=message.send_requested. 4. COMMIT. 5. Worker підхоплює подію, викликає API каналу. 6. Оновлює deliveryStatus=SENT або FAILED	Повідомлення доставлено у канал або зафіксовано помилку у Message.errorCode
ViewInbox	Operator	Користувач авторизований, є членство у робочому просторі	1. GET /v1/conversations?folder=all&cursor=… 2. API фільтрує по workspaceId з JWT. 3. Повертає список з пагінацією. 4. Клієнт підписується на Socket.IO-кімнату workspace:{id}	На клієнті оновлюється UI; нові події надходять у реальному часі
SearchMessages	Operator	Користувач у робочому просторі	1. Вводить запит у поле пошуку. 2. GET /v1/search?q=… 3. API виконує запит з tsvector + GIN. 4. Якщо результатів мало, виконує fuzzy fallback через pg_trgm	Повертається список релевантних Message з підсвічуванням збігів
ManageMembers	Owner / Admin	Поточний користувач має роль OWNER або ADMIN	1. GET /v1/workspaces/:id/members. 2. POST /v1/workspaces/:id/invitations з email. 3. Запрошений отримує email з deep-link. 4. Після прийняття створюється Membership	До робочого простору додано нового користувача з обраною роллю

Use cases охоплюють повний обсяг функціональних вимог підрозділу 2.1 і утворюють основу для подальшого проєктування.

2.4. Розробка поведінкових UML-діаграм

Поведінкові UML-діаграми описують динаміку системи. Для omnichat-hub побудовано діаграми діяльності (activity diagram), що деталізують ключові бізнес-сценарії з розгалуженнями та паралельними потоками, та діаграми

послідовності (sequence diagram), що показують обмін повідомленнями між компонентами у часі для архітектурно складних потоків.

2.4.1. Діаграми діяльності

Розроблено три діаграми діяльності для найскладніших сценаріїв: підключення каналу OLX, надсилання вихідного повідомлення та інкрементальна синхронізація. Першу показано на рисунку Б.15.

Потік має дві точки завершення помилкою: відмова користувача на стороні OLX та невалідний CSRF-state на callback. Перевірка state захищає від підробки міжсайтового запиту: значення генерується сервером, зберігається у Redis з TTL 600 секунд і використовується одноразово. ChannelAccount створюється лише після обміну code на токени та їх шифрування.

Другу діаграму показано на рисунку Б.16.

Діаграма ілюструє ідемпотентність вихідних повідомлень (повторний запит з тим самим clientMessageId повертає наявне повідомлення завдяки унікальному обмеженню (conversationId, clientMessageId)) та обчислення needsReply чистою доменною функцією applySellerReplySuccessEffects з пакета @omnichat/domain, результат якої зберігається у тій самій транзакції, що Message та OutboxEvent. Цикл повторних спроб воркера реалізує семантику at-least-once з експоненційною затримкою.

Третю діаграму показано на рисунку Б.17.

Дедуплікація на рівні БД забезпечується унікальним обмеженням (conversationId, externalMessageId), тому повторний прохід після збою безпечний. Оновлення checkpoint відбувається лише після успішного збереження пакета, гарантуючи, що наступний прохід почне з тієї ж позиції. Операція refreshAuth нетривіальна лише для OLX, оскільки Rozetka і Prom використовують статичні токени.

Повні версії діаграм діяльності винесено у додаток Б (рис. Б.15–Б.17).

2.4.2. Діаграми послідовності

Діаграми послідовності деталізують обмін повідомленнями між компонентами у часі для трьох потоків: транзакційний Outbox-патерн, OAuth 2.0 OLX та envelope encryption.

Транзакційний Outbox-патерн забезпечує гарантовану at-least-once доставку реалтайм-подій і push-сповіщень навіть за збоїв окремих компонентів [1]. Діаграму показано на рисунку Б.18.

Перевага над прямим socket.emit() з контролера подвійна. Транзакція БД атомарна (або всі чотири INSERT/UPDATE відбулися, або жодна; неможливий стан «Message збережено, OutboxEvent ні»), а реалтайм-доставка decoupled від HTTP-запиту: оператор отримує 201 Created одразу після COMMIT [2], а доставка підписникам відбувається асинхронно у фоновому воркері. Захоплення подій реалізовано оптимістичним умовним updateMany з полями lockId та lockedAt, що дозволяє запускати кілька процесорів конкурентно (concurrency = 10) без подвійної обробки [10].

OLX є єдиним з трьох каналів, що використовує повноцінний OAuth 2.0 потік Authorization Code [7]; Rozetka і Prom працюють зі статичними токенами. Діаграму показано на рисунку Б.19.

Усі звернення до OLX виконуються на єдиний хост www.olx.ua [28]: authorize за шляхом /oauth/authorize, обмін і оновлення токенів за /api/open/oauth/token, профіль за /api/partner/users/me. Запит scope складається з трьох значень через пробіл: read write v2.

Чутливі дані зберігаються у БД лише зашифрованими. Схема обгорткового шифрування має три сутності: KEK (Key Encryption Key, що ніколи не залишає зовнішнього Infisical KMS), DEK (Data Encryption Key, окремий на кожен робочий простір, version-aware) і шифротекст payload. Діаграму показано на рисунку Б.20.

Схема розділяє ризик компрометації: навіть з дампом бази зловмисник не розшифрує токени без доступу до KEK у зовнішньому Infisical KMS, оскільки у БД лежить лише обгорнутий DEK і шифротекст. Версіонування

DEK через (workspaceId, version) дозволяє ротувати ключі без перешифрування історії [22].

Перелік ключових архітектурних патернів наведено у табл. 2.3.

Таблиця 2.3. Опис ключових архітектурних патернів

Патерн	**Призначення**	**Реалізація в omnichat-hub**
Transactional Outbox	Гарантована at-least-once доставка side-ефектів узгоджено з БД	Таблиця outbox_event + Worker з оптимістичним захопленням через умовний updateMany (lockId/lockedAt)
Repository	Інкапсуляція доступу до даних, абстракція над Prisma	Тонкі сервіси *.repository.ts у кожному NestJS-модулі
Dependency Injection	Тестабельність, заміна реалізацій у тестах	NestJS Module + Provider DI-контейнер
Adapter	Уніфікація гетерогенних API маркетплейсів	IChannelAdapter інтерфейс + OlxAdapter, RozetkaAdapter, PromAdapter
Strategy	Вибір протоколу автентифікації залежно від каналу	OAuthStrategy (OLX), TokenStrategy (Rozetka, Prom)
Envelope Encryption	Шифрування чутливих даних у БД з підтримкою ротації ключів	KEK у Infisical KMS, DEK per-workspace в channel_dek
Pub/Sub (Redis)	Горизонтальне масштабування реалтайму	Socket.IO Redis adapter + Redis emitter у воркерах
OpenAPI-first	Контрактна сумісність між backend і клієнтами	Декоратори @ApiResponse у контролерах → openapi.json → генерація packages/api-client

2.5. Розробка діаграм потоків даних (DFD)

Для omnichat-hub побудовано трирівневу декомпозицію: рівень 0 (контекстна діаграма), рівень 1 (декомпозиція ядра на 5 процесів) та рівень 2 (деталізація процесу Channel Sync). Контекстну діаграму показано на рисунку Б.21.

Контекстна діаграма показує omnichat-hub як єдиний процес, що отримує повідомлення з трьох маркетплейсів і доставляє їх двом типам клієнтів. Деталізацію наведено на рисунку Б.22.

Channel Workers відіграють ключову роль як міст між зовнішніми каналами і ядром: вони виконують вхідну синхронізацію, обробку Outbox-подій і вихідну доставку. Найскладніший процес Channel Sync декомпозовано на рівні 2 (рисунок Б.23).

Деталізація рівня 2 пояснює надійність архітектури: збій будь-якого підпроцесу не призводить до втрати повідомлень [10]. Якщо Message Persister упав між INSERT Message та INSERT OutboxEvent, спрацьовує rollback транзакції; якщо впали Outbox Poller або Outbox Processor, запис у PENDING чи застрягле PROCESSING підхоплюється наступним циклом polling.

2.5.1. Словник даних DFD

Словник даних формалізує елементи моделі потоків даних і усуває неоднозначність трактування діаграм. Зміст кожного сховища деталізується відповідною сутністю логічної моделі (підрозділ 2.6). Перелік зовнішніх сутностей наведено у табл. 2.4.

Таблиця 2.4. Словник даних: зовнішні сутності

Сутність	Тип	Опис
OLX	Зовнішня система	Маркетплейс OLX, джерело вхідних повідомлень покупців і приймач вихідних відповідей; авторизація OAuth 2.0
Rozetka	Зовнішня система	Маркетплейс Rozetka, джерело/приймач повідомлень; авторизація статичним API-токеном
Prom	Зовнішня система	Маркетплейс Prom.ua, джерело/приймач повідомлень; авторизація статичним API-токеном
Web Client	Зовнішній актор	Браузер оператора (Next.js), споживач REST API та реалтайм-подій Socket.IO
Mobile Client	Зовнішній актор	Мобільний застосунок оператора (Expo), споживач REST API, Socket.IO та push-сповіщень

Перелік процесів моделі наведено у табл. 2.5.

Таблиця 2.5. Словник даних: процеси

№	Процес	Рівень	Опис
0	omnichat-hub	0	Контекстний процес: агрегація комунікацій з маркетплейсів та доставка клієнтам

№	Процес	Рівень	Опис
1	API Gateway	1	Прийом REST і Socket.IO, валідація DTO, маршрутизація у модулі
2	Channel Workers	1	Фонові процеси синхронізації, обробки Outbox і вихідної доставки
3	Realtime Broker	1	Pub-sub шар розсилки подій через Socket.IO Redis-адаптер
4	Storage Layer	1	Типобезпечний доступ до PostgreSQL через Prisma
5	Auth Module	1	Email-OTP, Google OAuth, видача та ротація JWT-сесій
2.1	Sync Scheduler	2	Крон-планувальник завдань синхронізації каналів
2.2	Channel Adapter	2	Протокол-специфічний адаптер каналу (OLX/Rozetka/Prom)
2.3	Checkpoint Reader/Writer	2	Читання та оновлення позиції інкрементального sync
2.4	Message Persister	2	Транзакційне збереження Message, Conversation, OutboxEvent
2.5	Outbox Poller	2	Вибірка непідтверджених подій і постановка у чергу
2.6	Outbox Processor	2	Захоплення, публікація та позначення подій як оброблених

Перелік сховищ даних з відповідністю сутностям логічної моделі наведено у табл. 2.6.

Таблиця 2.6. Словник даних: сховища даних

Позначення	Сховище	Носій	Основний вміст
D1	PostgreSQL	Реляційна СУБД	24 таблиці логічної моделі: User, Workspace, ChannelAccount, Conversation, Message, OutboxEvent, ChannelDek та інші
D2	Redis	In-memory сховище	BullMQ-черги, Socket.IO-канали, CSRF-state OAuth, кеш сесій, hash refresh-токенів
D3	Spaces (S3)	Об'єктне сховище	Бінарний вміст вкладень (Attachment), доступ через storageKey

Перелік основних потоків даних наведено у табл. 2.7.

Таблиця 2.7. Словник даних: потоки даних

Потік	Джерело → Приймач	Структура (елементи даних)
Вхідне повідомлення каналу	Channel Adapter → Message Persister	externalMessageId, conversationId, direction=IN, text, sentAt, attachments
Транзакційний запис	Message Persister → D1	Message{...} + Conversation{lastActivityAt, needsReply} + OutboxEvent{type, payload, status=PENDING}
Подія Outbox	D1 → Outbox Poller	outboxEventId, type, payload, aggregateId, workspaceId, status, attempts, nextAttemptAt, lockedAt
Реалтайм-подія	Outbox Processor → Realtime Broker	event(type), payload(JSON), workspaceId (ключ кімнати)
Socket.IO emit	Realtime Broker → Web/Mobile Client	подія message.created/message.updated/conversation.updated + outboxEventId
OAuth callback	OLX → API Gateway	code, state
Токен-відповідь OLX	OLX → API Gateway	access_token, refresh_token, expires_in, token_type
Зашифровані credentials	API Gateway → D1	authDataEncrypted (формат v1:version:iv:tag:ciphertext)
REST-запит клієнта	Web/Mobile Client → API Gateway	HTTP-метод, шлях /v1/*, JWT (з workspaceId), тіло DTO
Push-нотифікація	Channel Workers → Mobile Client	platform (APNs/FCM), token пристрою, payload сповіщення

Перелік ключових елементарних елементів даних наведено у табл. 2.8.

Таблиця 2.8. Словник даних: елементарні елементи даних

Елемент	Тип	Опис
workspaceId	String (cuid)	Ідентифікатор робочого простору, наскрізний ключ багатотенантної ізоляції
conversationId	String (cuid)	Ідентифікатор діалогу з покупцем у конкретному каналі
externalMessageId	String	Ідентифікатор повідомлення на стороні маркетплейсу, ключ дедуплікації вхідних
clientMessageId	String	Клієнтський ідентифікатор, ключ ідемпотентності вихідних повідомлень
outboxEventId	String (cuid)	Ідентифікатор події Outbox, використовується для клієнтської дедуплікації подій
deliveryStatus	Enum	Статус доставки повідомлення: SENDING, SENT, DELIVERED, READ, FAILED
authDataEncrypted	String	Зашифровані credentials каналу у форматі envelope encryption
needsReply	Boolean	Прапорець «потребує відповіді», обчислюється доменним шаром

2.6. Проєктування архітектури системи

Архітектура omnichat-hub побудована на принципах модульності та явного розподілу відповідальності. Кодова база організована як monorepo на pnpm workspaces та оркестраторі Turborepo, що дозволяє ділити спільний код без публікації окремих npm-пакетів [4]. Монорепозиторій складається з 5 апів (самостійних застосунків із власним runtime) і 6 спільних пакетів (бібліотек коду). Загальну архітектурну діаграму показано на рисунку Б.24.

Підхід monorepo гарантує синхронну еволюцію коду (зміна правила в packages/domain одразу відбивається в API і Workers) [4], забезпечує одноразове налаштування інструментів і дозволяє інкрементальну збірку через кеш Turborepo. Топологію сервісів показано на рисунку Б.25.

Окремий перелік модулів представлено у табл. 2.9.

Таблиця 2.9. Список модулів monorepo

Тип	Шлях	Опис
App	apps/api	NestJS REST API з версіонуванням /v1, Socket.IO сервер, Swagger /docs, JWT та workspace guards
App	apps/workers	BullMQ-споживач для Outbox, push, sync, cleanup, rollups; health на порту 4122
App	apps/web	Next.js 16 з App Router, TanStack Query, Tailwind; інтерфейс оператора
App	apps/mobile	Expo SDK 53 з NativeWind та Expo Notifications для push
Package	packages/db	Канонічна Prisma-схема, міграції, seed-скрипти
Package	packages/domain	Pure-функції бізнес-правил (needsReply, snooze, auto-archive) з 100% unit-тестами
Package	packages/contracts	Zod-DTO та realtime event envelopes (типи подій Socket.IO)
Package	packages/api-client	Згенерований з openapi.json TypeScript-клієнт з retry на 401
Package	packages/config	Валідований env-loader (Zod), lint/prettier-конфіги
Package	packages/ui-tokens	Дизайн-токени для Tailwind (без компонентів)

Два бекенд-застосунки (apps/api і apps/workers) не викликають один одного напряму, а обмінюються роботою виключно через Redis: API ставить завдання у чергу BullMQ і публікує події, воркери їх споживають. Така непряма взаємодія розв’язує застосунки у часі і дозволяє масштабувати кожен незалежно [2].

2.7. Розробка структурних UML-діаграм

Основні структурні UML-діаграми системи — діаграми класів модулів Conversations і Channels, діаграма компонентів і діаграма розгортання, а також логічна модель даних у нотації IDEF1X та діаграма станів повідомлення — представлені в додатку Б.

2.8. Висновки до розділу 2

У другому розділі виконано повний цикл проєктування omnichat-hub: від формалізації вимог до структурних UML-діаграм і логічної моделі даних. Кожне рішення обґрунтовано з огляду на специфіку предметної області (агрегація комунікацій з українських маркетплейсів) та технічні цілі (надійність, низька латентність, горизонтальне масштабування).

Сформульовано 10 функціональних і 10 нефункціональних вимог, на основі яких розроблено 15 проєктних специфікацій (табл. 2.1) з вимірюваними метриками прийому для автоматизації приймального тестування у CI. Побудовано діаграму варіантів використання (рис. Б.14) з чотирма акторами (Owner, Admin, Operator, External Channel) та 20 use cases. Розроблено поведінкові UML-діаграми: три діаграми діяльності (рис. Б.15–Б.17) та три діаграми послідовності (рис. Б.18–Б.20). Побудовано три рівні DFD: контекстну діаграму (рис. Б.21), декомпозицію на 5 процесів (рис. Б.22) та деталізацію Channel Sync на 6 підпроцесів (рис. Б.23), а також словник даних DFD.

Спроєктовано архітектуру як monorepo з 5 апами і 6 спільними пакетами на pnpm + Turborepo, що забезпечує синхронну еволюцію коду між backend (apps/api, apps/workers) і клієнтами (apps/web, apps/mobile), уніфіковану конфігурацію та інкрементальну збірку. Виокремлення доменної логіки у пакет packages/domain як pure-функцій дозволяє переюзовувати бізнес-логіку в API і воркерах та забезпечує 100% unit-test coverage цих правил [5].

Ключовим архітектурним рішенням є транзакційний Outbox-патерн (рис. Б.18), що гарантує at-least-once-доставку реалтайм-подій і push-сповіщень, узгоджену з даними у БД [1]. У одній транзакції виконуються INSERT Message, UPDATE Conversation і INSERT OutboxEvent, що виключає неконсистентність «повідомлення збережене, але клієнти його не отримали». Воркери Outbox Poller і Outbox Processor обробляють події оптимістичним захопленням умовним updateMany з полями lockId і lockedAt, що дозволяє

масштабувати обробників горизонтально без подвійної обробки; подія у PROCESSING із застарілим lockedAt (понад lock TTL, за замовчуванням 300 секунд) повторно захоплюється поллером, забезпечуючи самовідновлення після збоїв.

Для горизонтального масштабування реалтайм-шару обрано Socket.IO з Redis-адаптером [14]: подія, опублікована одним інстансом у кімнату робочого простору, доставляється підписникам інших інстансів через Pub/Sub Redis, а воркери публікують події через Redis emitter. Обрано Prisma як ORM для PostgreSQL з огляду на типобезпечність, code-generation і міграції [13]; прямий SQL зведено до мінімуму через тонкі repository-сервіси. Повнотекстовий пошук реалізовано засобами PostgreSQL (tsvector + GIN, pg_trgm для fuzzy fallback) без окремого пошукового движка [9].

Безпека даних забезпечується багатошарово. Чутливі credentials (refresh-токени OAuth OLX і статичні API-токени Rozetka та Prom) шифруються AES-256-GCM у схемі envelope encryption (рис. Б.20); кожен робочий простір має власний DEK з версіонуванням і ротацією, а обгортковий KEK зберігається у зовнішньому Infisical KMS поза БД. Канальна автентифікація диференційована: OLX використовує OAuth 2.0 Authorization Code з refresh-токенами, Rozetka і Prom статичні API-токени з кабінету продавця. Багатотенантна ізоляція забезпечується на трьох рівнях: workspace-membership guard у NestJS, scoped-запити Prisma за workspaceId, ізольовані Socket.IO-кімнати.

Структурні UML-діаграми (рис. Б.26–Б.29) деталізують внутрішню організацію модулів Conversations і Channels, компонентну структуру та деплоймент-топологію на DigitalOcean App Platform і Vercel. Логічну модель даних побудовано у нотації IDEF1X (рис. Б.30) на основі канонічної Prisma-схеми; модель містить 24 сутності, серед яких дві зі складеним або успадкованим первинним ключем (ConversationTag і WorkspaceSettings). Кінцевий автомат стану повідомлення (рис. Б.31) формалізує життєвий цикл доставки від SENDING до READ з обробкою термінального стану FAILED.

РОЗДІЛ 3. РОЗРОБКА УНІВЕРСАЛЬНОГО МЕСЕНДЖЕРА omnichat-hub

3.1. Обґрунтування вибору інструментальних засобів

Реалізація omnichat-hub поєднує серверну логіку обробки повідомлень з трьох українських маркетплейсів, веб-інтерфейс оператора, мобільний клієнт із push-сповіщеннями, систему черг для гарантованої доставки та інфраструктурний шар розгортання у хмарі. Кожна частина потребує власного набору інструментів, обраних із розрахунку на горизонтальне масштабування, тип-безпеку, низький поріг входу та документованість. Нижче розглянуто альтернативи для кожної категорії та обґрунтовано остаточний вибір.

3.1.1. Серверне середовище виконання

Розглянуто чотири альтернативи: Node.js 22 LTS, Deno, Bun та Python 3.12 з FastAPI. Node.js 22 LTS обрано з трьох причин [22]. По-перше, це реліз з підтримкою до квітня 2027 року, що покриває весь життєвий цикл роботи та розгортання MVP (серпень 2026). По-друге, npm містить понад 2,5 мільйона пакетів, серед яких підтримувані бібліотеки для всіх інтеграцій: BullMQ, Socket.IO, Prisma, драйвер PostgreSQL. По-третє, єдина мова TypeScript для backend, frontend, mobile та інфраструктури зменшує когнітивне навантаження на розробника-одинака і дозволяє переюзовувати типи через спільні пакети monorepo [21]. Deno відхилено через слабшу npm-сумісність, Bun як експериментальне середовище без LTS, Python через потребу другої мови та відсутність єдиного типового рівня.

3.1.2. Backend-фреймворк

Розглянуто NestJS, Express, Fastify та AdonisJS. Обрано NestJS через модульну архітектуру з вбудованим контейнером залежностей (Dependency Injection), декларативний роутінг через декоратори та інтеграцію з OpenAPI-генератором [11]. Express без структурних обмежень для проєкту з понад 50

ендпоінтами призводить до зростання технічного боргу; Fastify не має офіційного DI-контейнера; AdonisJS занадто важкий для API-only сервісу. NestJS використовує патерн модулів, де кожна функціональна область (auth, conversations, messages, channel-accounts, realtime) інкапсульована у власному модулі з контролерами, сервісами та провайдерами [2]. Це відповідає принципам Domain-Driven Design і дозволяє замінювати реалізації через інтерфейси, що критично для тестування [3].

3.1.3. ORM і база даних

Для шару доступу до БД розглянуто Prisma, TypeORM, Drizzle та Knex. Обрано Prisma через автоматичну генерацію TypeScript-типів зі схеми schema.prisma, вбудовану систему міграцій та тип-безпечні запити з підказками IDE [13]. TypeORM використовує декоратори, що ускладнює статичний аналіз; Drizzle має меншу екосистему; Knex не має типобезпеки.

Як СУБД розглянуто PostgreSQL 16, MySQL 8, Microsoft SQL Server 2022 та документоорієнтовану MongoDB 7. Порівняння наведено в таблиці 3.1. PostgreSQL 16 обрано через повноцінну підтримку ACID-транзакцій, критичну для транзакційного Outbox-патерну, де запис Message, оновлення Conversation і запис OutboxEvent комітяться атомарно в одній транзакції [10]. Вагомими стали також вбудований повнотекстовий пошук через tsvector і GIN-індекси [9], тип JSONB для payload OutboxEvent, розширення pg_trgm для нечіткого пошуку покупців за іменем та мова PL/pgSQL для тригерів і збережених процедур. MySQL відхилено через слабшу підтримку JSON і відсутність триграмного пошуку; MS SQL Server через пропрієтарну ліцензію; MongoDB не підтримує JOIN-запити та серверні тригери, що для глибокої реляційної моделі

(Workspace → Conversation → Message → Attachment) унеможливлює перенесення логіки цілісності на рівень БД.

Таблиця 3.1. Порівняння систем керування базами даних для omnichat-hub

Критерій	PostgreSQL 16	MySQL 8	MS SQL Server 2022	MongoDB 7
Модель даних	реляційна	реляційна	реляційна	документна
ACID-транзакції	повні, на рівні рядків	повні (InnoDB)	повні	багатодокументні, з обмеженнями
Повнотекстовий пошук	вбудований tsvector + GIN	базовий FULLTEXT	Full-Text Search	текстові індекси
Нечіткий пошук	pg_trgm (триграми)	відсутній нативно	відсутній нативно	відсутній нативно
Тип JSON	JSONB з індексацією	JSON без GIN	JSON без GIN	нативний BSON
Тригери і збережені процедури	PL/pgSQL, повна підтримка	обмежена	T-SQL, повна підтримка	відсутні (тільки change streams)
Ліцензія/вартість	відкрита, безкоштовна	відкрита (GPL)	пропрієтарна, платна	SSPL
Інтеграція з Prisma	повна	повна	повна	часткова

3.1.4. Кеш, Pub/Sub та черги

Redis обрано як єдиний компонент для трьох ролей: кеш сесій і throttling-каунтерів, Pub/Sub-шина для масштабування Socket.IO, broker черги BullMQ [16]. Memcached не підтримує Pub/Sub; NATS потребує окремого сервісу поряд із кешем. Для черг обрано BullMQ через нативну інтеграцію з Node.js, використання Redis як broker, вбудований retry з exponential backoff і dead-letter queues та офіційний модуль @nestjs/bullmq [15]. RabbitMQ потребує окремого AMQP-сервісу; AWS SQS прив’язує до AWS, що суперечить хостингу на DigitalOcean.

3.1.5. Реалтайм-шар

Розглянуто Socket.IO, raw WebSocket (ws), Server-Sent Events (SSE) та Pusher. Обрано Socket.IO через готовий Redis-адаптер для горизонтального масштабування, автоматичний fallback на long-polling за блокувальних проксі, вбудовану підтримку кімнат (workspace:{id}) та офіційні клієнти для веб, React

Native і Node.js [14]. Raw WebSocket вимагає ручної реалізації reconnect і кімнат; SSE не підтримує двостороннього зв’язку; Pusher додає зовнішню залежність із вартістю та latency.

3.1.6. Frontend і mobile

Для веб-інтерфейсу оператора обрано Next.js 16 з React 19 [12; 20] через React Server Components у App Router, ISR для public-сторінок, інтеграцію з Vercel та екосистему UI-бібліотек (Tailwind CSS, shadcn/ui, Radix UI). Remix має меншу екосистему; SvelteKit і Nuxt прив’язують до інших мов.

Для мобільного клієнта обрано Expo SDK 53 [19]: Managed Workflow з готовими нативними модулями для push (expo-notifications), глибоких посилань (expo-linking), безпечного сховища (expo-secure-store) та OTA-оновлень EAS Update без проходження review у App Store і Google Play. React Native CLI потребує налаштування Xcode і Android Studio; Flutter додає Dart як четверту мову; Capacitor обгортає веб-додаток у WebView з гіршою продуктивністю.

3.1.7. Інфраструктура, CI/CD та моніторинг

Для опису інфраструктури як код обрано Pulumi через підтримку TypeScript та повноцінну тип-безпеку ресурсів DigitalOcean і Vercel [17]; Terraform використовує власну DSL. Для CI/CD обрано GitHub Actions через native-інтеграцію з репозиторієм, безкоштовні хвилини та маркетплейс actions. Хостинг бекенду - DigitalOcean App Platform з білдом з Dockerfile, автомасштабуванням та інтеграцією з Managed PostgreSQL і Spaces [25]; frontend - Vercel через рідну інтеграцію з Next.js [26]. Моніторинг помилок - Sentry (NestJS і Next.js, source maps, breadcrumbs, performance monitoring) [23]; метрики та логи централізовано у Grafana Cloud з Loki і Prometheus, що дозволяє кореляцію між Sentry і логами через trace_id [24]. Локальна розробка інкапсульована у Docker Compose, який однією командою pnpm infra:up піднімає PostgreSQL 16, Redis 7 та MinIO (S3-сумісний емулятор Spaces).

3.1.8. Підсумкова таблиця інструментів

Підсумковий перелік обраних інструментальних засобів наведено в таблиці 3.2.

Таблиця 3.2. Підсумкова таблиця обраних інструментальних засобів omnichat-hub

Категорія	**Технологія**	**Призначення**	**Обґрунтування вибору**
Runtime	Node.js 22 LTS	Виконання серверного коду	LTS до 2027, єдина мова TS у стеку
Backend framework	NestJS	Структура API і workers	Модулі, DI, OpenAPI-генератор
Мова	TypeScript 5.6 (strict)	Тип-безпека	Усуває цілий клас runtime-помилок
ORM	Prisma	Доступ до БД	Генерація типів, міграції, type-safe queries
База даних	PostgreSQL 16	Збереження даних	ACID, JSONB, tsvector, pg_trgm
Кеш / Pub-Sub	Redis 7	Кеш + Pub/Sub + черги	Універсальний компонент
Realtime	Socket.IO 4	WebSocket із fallback	Redis-адаптер Socket.IO, кімнати
Черги	BullMQ	Обробка фонових задач	Node-native, retry, backoff
Валідація DTO	class-validator + class-transformer	Перевірка вхідних DTO	Декоратори узгоджені з NestJS
Контракти	Zod	Runtime + статичні схеми	Спільна валідація API і UI

Продовження табл. 3.2

Категорія	**Технологія**	**Призначення**	**Обґрунтування вибору**
Логування	pino + nestjs-pino	Структуровані JSON-логи	Швидкість, інтеграція з Loki
Безпека	helmet + CORS allowlist + throttler	HTTP-захист	Стандартні best practices
Frontend	Next.js 16 + React 19	Веб-інтерфейс оператора	App Router, Server Components, ISR
State management	TanStack Query	Кешування server-state	Standard de-facto для REST
Стилі	Tailwind CSS + ui-tokens	Дизайн-система	Спільні токени з mobile
Mobile	Expo SDK 53	Мобільний клієнт	Managed workflow, EAS, push
Mobile стилі	NativeWind + Tamagui	Tailwind для React Native	Узгоджено з web-токенами
OpenAPI	OpenAPI 3.1	Контракт API	Source-of-truth для клієнтів
Згенерований клієнт	@omnichat/api-client (openapi-fetch)	TS-клієнт для web/mobile	Drift gate у CI
IaC	Pulumi (TypeScript)	Інфраструктура як код	TS-нативний, типобезпечний
CI/CD	GitHub Actions	Автоматизація pipeline	Безкоштовно, native до GitHub
Хостинг API	DigitalOcean App Platform	Розгортання NestJS-апів	Managed, автомасштабування
Хостинг frontend	Vercel	Розгортання Next.js	Native-інтеграція з Next.js
Хостинг БД	DigitalOcean Managed Postgres	PostgreSQL у production	Managed backup, моніторинг
Об'єктне сховище	DigitalOcean Spaces	Attachments	S3-сумісний API
Моніторинг помилок	Sentry	Error tracking + sourcemaps	NestJS + Next.js інтеграції
Метрики/логи	Grafana Cloud (Loki + Prometheus)	Централізовані логи	Trace correlation
Push-сповіщення	Expo Push (APNs/FCM)	Доставка push на iOS/Android	Через Expo SDK 53
Локальна розробка	Docker Compose	Postgres + Redis + MinIO	Однаковість середовищ
Менеджер пакетів	pnpm	Управління monorepo	Швидкий, дисковий-ефективний
Оркестрація monorepo	Turborepo	Інкрементальні білди	Кешування CI-результатів
Тестування unit	Vitest	Юніт-тести domain	Швидкий, ESM-сумісний
Тестування E2E	Playwright	Кінцеві сценарії	Браузерна автоматизація

Узагальнену шарову структуру обраного стеку показано на рисунку Б.32.

Стек складається з п’яти шарів: клієнти (веб-дашборд на Next.js 16 з React 19 і мобільний клієнт на Expo SDK 53, обидва через згенерований клієнт @omnichat/api-client з контракту OpenAPI 3.1), шар API (apps/api на NestJS), шар фонової обробки (apps/workers), шар даних і шин (PostgreSQL 16, Redis 7, об’єктне сховище) та шар спостережуваності й розгортання (Sentry, Grafana Cloud, GitHub Actions, Pulumi).

3.2. Розробка системи

Розробку omnichat-hub велено в архітектурі pnpm-monorepo, керованому через Turborepo. Кожен компонент реалізовано як окремий додаток або переюзовний пакет з власним package.json, незалежним білдом і спільними скриптами координації з кореневого package.json.

3.2.1. Структура monorepo

Корінь містить директорії apps/ для запускних додатків, packages/ для переюзовних бібліотек, infra/ для інфраструктури, docs/ для документації. Усього п’ять додатків і шість основних спільних пакетів.

Додатки: - apps/api - NestJS-сервер, що обслуговує REST API за версією /v1/*, WebSocket-сервер Socket.IO на /ws, Swagger на /docs та health-перевірки на /health і /ready. Усі вхідні DTO валідуються глобальним ValidationPipe (whitelist, transform, forbidNonWhitelisted). Авторизація через JwtAuthGuard і WorkspaceMembershipGuard. Порт 4121. - apps/workers - окремий NestJS-процес, що споживає чергу BullMQ і виконує доставку реалтайм-подій (outbox), інкрементальну синхронізацію каналів (channel-sync), доставку push (push), обслуговуючі задачі (maintenance) та щоденні агрегати метрик (metrics: WorkspaceDailyRollup). Health-перевірки на порту 4122. - apps/web - Next.js 16 App Router, дашборд оператора. Сторінки /inbox, /conversations/[id], /settings, /auth/*. TanStack Query для кешування, Socket.IO-client для realtime. -

apps/mobile - Expo SDK 53 React Native: inbox, перегляд діалогу, відправка повідомлень з вкладеннями, push через expo-notifications.

Спільні пакети: - packages/api-client - згенерований TS-клієнт з OpenAPI-специфікації через openapi-fetch; автоматичний retry на 401 з оновленням JWT. - packages/contracts - Zod-схеми та DTO для повторного використання у backend і frontend. - packages/domain - pure-функції бізнес-правил (computeNeedsReply, applySellerReplySuccessEffects) [5]; покриття Vitest 100%. - packages/db - Prisma-схема (schema.prisma) і міграції; експортує PrismaClient з типами. - packages/config - валідований loader змінних середовища через Zod. - packages/ui-tokens - дизайн-токени як Tailwind-preset для web і Tamagui-config для mobile.

Окремо існує packages/kms для обгорткового шифрування (EnvelopeCryptoService, DekManager). Ієрархію директорій показано на рисунку Б.33.

Корінь містить файли конфігурації monorepo; apps/ розкривається у п’ять додатків, packages/ - у шість основних пакетів плюс kms/, infra/ містить Pulumi для production і Docker Compose для локального середовища.

3.2.2. Реалізація транзакційного Outbox-патерну

Ключове архітектурне рішення - транзакційний Outbox-патерн для гарантованої доставки at-least-once реалтайм-подій і push-сповіщень [1; 6]. При створенні повідомлення API виконує одну транзакцію БД, в якій атомарно: створюється Message, оновлюються деривовані поля Conversation (needsReply, lastSellerReplyAt, lastActivityAt) і записується рядок OutboxEvent зі статусом PENDING та payload. Лише після коміту фоновий OutboxPoller у apps/workers опитує OutboxEvent, забирає прострочені події і ставить їх у BullMQ-чергу outbox.process.

Поллер виконує prisma.outboxEvent.findMany з предикатом status ∈ {PENDING, FAILED, PROCESSING}, nextAttemptAt ≤ now, attempts < maxAttempts, OR: [{lockedAt: null}, {lockedAt: {lt: staleBefore}}], з orderBy:

{createdAt: "asc"} і take: 200, після чого ставить кожен outboxEventId у чергу (повний код доступний у репозиторії проєкту). Реалізація має три характеристики. Опитування виконується раз на 60 секунд (OUTBOX_POLL_INTERVAL_SEC), а в тестовому середовищі раз на секунду. У вибірку потрапляють події у трьох статусах: PENDING (нові), FAILED (чекають retry), PROCESSING (із простроченим блокуванням, тобто попередній воркер впав і не звільнив запис). Обмеження attempts < maxAttempts (за замовчуванням 10) запобігає вічному ретраю отруйних повідомлень: після вичерпання спроб подія залишається у БД зі статусом FAILED для ручного розгляду через /v1/debug/outbox/*.

Захоплення події воркером-процесором реалізовано не через пессимістичне блокування FOR UPDATE SKIP LOCKED, а через оптимістичне захоплення засобами Prisma: атомарний умовний updateMany із тим самим предикатом WHERE, що й у поллера, і блоком data: { status: "PROCESSING", lockedAt: now, lockId, attempts: { increment: 1 } }. Ідентифікатор lockId генерується через randomUUID(). Якщо updateMany повертає count === 0, подію вже захопив інший воркер, і поточний тихо завершує обробку. Конкурентність процесора дорівнює 10. Це забезпечує безпечну паралельну обробку без міжпроцесних блокувань рядків PostgreSQL. Поріг застарілого блокування - OUTBOX_LOCK_TTL_SEC за замовчуванням 300 секунд; подія у PROCESSING, чий lockedAt старший за цей поріг, повторно потрапляє у вибірку і перезахоплюється. При помилці застосовується експоненційний backoff з капом 1 година. При перевищенні backlog у 200 подій модуль реєструє попередження у структурованих логах (pino) і Sentry для алертингу.

Один OutboxEvent має два незалежні прапорці доставки: wsDeliveredAt (WebSocket у workspace-кімнату, baseline для кожної події) і pushDeliveredAt (push, опційно, залежно від типу події й преференцій). Кожен виставляється окремо й ідемпотентно (if (!event.wsDeliveredAt)), тому повторна обробка не дублює доставку. Статус PROCESSED і processedAt встановлюються лише

після обох сторін доставки, після чого lockedAt і lockId обнуляються. Послідовність обробки відкладеної події показано на рисунку Б.34.

На рисунку Б.34 зображено повний шлях події: атомарну транзакцію запису, періодичне опитування поллером, постановку в чергу, оптимістичне захоплення процесором через умовний updateMany, дві незалежні сторони доставки та фінальний перехід у PROCESSED.

3.2.3. OAuth 2.0 інтеграція з OLX

Серед трьох маркетплейсів лише OLX реалізує повноцінну авторизацію за стандартом OAuth 2.0 з потоком Authorization Code та підтримкою refresh-токенів [7]. Rozetka і Prom використовують статичні API-токени, які користувач отримує в кабінеті продавця і вставляє у форму omnichat-hub [27; 29].

Потік OLX складається з двох етапів. На етапі start API формує авторизаційний URL на хості https://www.olx.ua зі шляхом /oauth/authorize і параметрами client_id, response_type=code, redirect_uri, state, scope (значення read write v2), зберігає state у Redis з TTL 600 секунд і повертає URL клієнту (повний код доступний у репозиторії проєкту). На етапі callback після перенаправлення користувача API отримує code і state. Спочатку перевіряється state через getDelJson (атомарна операція read-and-delete у Redis запобігає replay-атакам). Далі виконується POST на token-ендпоінт https://www.olx.ua/api/open/oauth/token з grant_type=authorization_code, content-type: application/x-www-form-urlencoded і тілом code, client_id, client_secret, redirect_uri.

У відповідь OLX повертає JSON з access_token, refresh_token, expires_in, scope, token_type [28]. Усі поля проходять через type-safe парсери safeString і safeNumber. Authorize-ендпоінт і token-ендпоінт розташовані під різними базовими шляхами одного хоста: authorize під /oauth/authorize, token під /api/open/oauth/token.

Перед записом у БД токени шифруються через сервіс обгорткового шифрування. Метод crypto.encrypt(workspaceId, payload) отримує payload з полями schemaVersion, provider: "OLX", workspaceId, userId, obtainedAt, accessToken, refreshToken, expiresIn, scope, tokenType і повертає рядок з версією DEK, IV, auth tag і ciphertext (повний код доступний у репозиторії проєкту). Цей рядок зберігається у полі authDataEncrypted таблиці ChannelAccount. Ключ provider: "OLX" - це поле зашифрованого payload, а не колонка таблиці; тип каналу зберігається в окремій колонці channel типу Channel. Сам DEK ніколи не залишає процеса у відкритому вигляді: він тримається у пам’яті лише на час шифрування і розшифровується через ключ KEK, що зберігається у зовнішньому Infisical KMS.

Окремо реалізована перевірка квоти підключень OLX на workspace. Для нового акаунта (без переданого channelAccountId) кількість активних OLX-акаунтів порівнюється з лімітом плану (maxOlxAccounts); re-auth flow для існуючого ChannelAccount пропускає цю перевірку, інакше зменшення ліміту нижче поточного використання заблокувало б оновлення токенів. Послідовність OAuth-обміну показано на рисунку Б.35.

На рисунку Б.35 зображено OAuth-обмін з п’ятьма учасниками: генерацію state, перенаправлення на форму згоди OLX, перевірку state через атомарний read-and-delete, обмін code на пару токенів, отримання профілю продавця та шифрування токенів перед записом у ChannelAccount.

3.2.4. Шифрування ключів каналів через envelope encryption

Усі чутливі дані ChannelAccount (статичні токени Rozetka і Prom, пара access/refresh OLX) шифруються двошарово за схемою envelope encryption. Схема забезпечує per-workspace ізоляцію ключів, ротацію без перешифрування існуючих даних та відсутність master-ключа у відкритому вигляді у БД.

Шар 1: Data Encryption Key (DEK) на рівні workspace. Кожен workspace має власний DEK завдовжки 256 біт, який генерується через

crypto.randomBytes(32) при першому використанні [22] і зберігається в БД у зашифрованому вигляді (поле encryptedDek таблиці ChannelDek). DEK має поле version, яке інкрементується при ротації; старі версії зберігаються, поки існують зашифровані ними дані. Прапорець isActive позначає активну версію.

Шар 2: Key Encryption Key (KEK) - обгортковий master-ключ. KEK ніколи не залишає зовнішнього Infisical KMS; у БД і коді зберігається лише ідентифікатор kekKeyId. Wrap і unwrap DEK виконуються віддаленими викликами Infisical KMS, а сам DEK у відкритому вигляді тримається у in-memory кеші DekManager з TTL 5 хвилин і максимумом 100 записів.

Власне шифрування виконується алгоритмом AES-256-GCM, який забезпечує одночасно конфіденційність і автентичність (auth tag). Метод encrypt отримує активний DEK через DekManager.getActiveDek(workspaceId), серіалізує payload у JSON, виконує AES-GCM-шифрування з випадковим IV (12 байт) і повертає форматований рядок виду v1:<dekVersion>:<iv_base64url>:<tag_base64url>:<ciphertext_base64url> (повний код доступний у репозиторії проєкту). Перший сегмент v1 - версія формату envelope, другий - версія DEK для ротації; OpenTelemetry-span дозволяє відстежити час шифрування у Grafana Cloud. Дзеркальний метод decrypt парсить рядок, витягує версію DEK, отримує відповідний DEK через DekManager.getDekByVersion (включно зі старими версіями), виконує розшифровування з перевіркою auth tag і десеріалізує JSON. При підробці шифротексту перевірка GCM auth tag не проходить і метод кидає помилку CiphertextTampered.

Ротація DEK виконується методом DekManager.rotate(): створюється новий DEK з версією N+1, виконується його wrap через Infisical KMS, у транзакції старі версії деактивуються (isActive=false, rotatedAt), а нова позначається активною. Старі версії залишаються в ChannelDek для розшифровування історичних даних. Кожна операція пише запис у AuditLog (dek.create, dek.rotate, dek.unwrap_legacy_version). Для запобігання leak секретів у логах pino відрізає поля Authorization, access_token, refresh_token,

password через redact; аналогічний beforeSend-хук у Sentry прибирає чутливі поля з breadcrumbs і extras. Схему показано на рисунку Б.36.

На рисунку Б.36 зображено схему envelope encryption: KEK у Infisical KMS, що ніколи не виходить назовні; таблиця ChannelDek із зашифрованим DEK та активна версія DEK у пам'яті процесу з TTL 5 хвилин; payload токенів, зашифрований через AES-256-GCM і збережений у форматі v1:version:iv:tag:ciphertext у полі authDataEncrypted.

3.2.5. Real-time доставка через Socket.IO з Redis-адаптером

Реалтайм-канал побудовано на Socket.IO 4 з Redis-адаптером для горизонтального масштабування [14]. Кожен інстанс apps/api піднімає власний Socket.IO-сервер, який ділиться станом кімнат з іншими інстансами через Redis Pub/Sub [16]. Це дозволяє розгортати API горизонтально (3+ інстанси за load-balancer DigitalOcean) без втрати реалтайму: незалежно від того, до якого інстансу підключений клієнт, події з будь-якого джерела доходять до нього.

Потік доставки складається з п'яти етапів. API отримує POST на /v1/conversations/:id/messages, виконує транзакцію БД (INSERT Message + UPDATE Conversation + INSERT OutboxEvent зі статусом PENDING). OutboxPoller опитує таблицю кожні 60 секунд і ставить прострочені події у чергу outbox.process. Процесор забирає завдання, читає OutboxEvent і публікує подію через Redis emitter (@socket.io/redis-emitter) у Pub/Sub-канал. Усі інстанси Socket.IO-сервера, підписані на цей канал, отримують подію і транслюють її клієнтам у кімнаті workspace:{workspaceId}; клієнти (web і mobile) отримують подію через WebSocket.

Дедуплікація на клієнті виконується через ідентифікатор outboxEventId [6]. Кожна подія несе цей ідентифікатор у payload; клієнт зберігає набір отриманих outboxEventId у кеші TanStack Query (web) або Zustand-стейті (mobile) і ігнорує дублікати. Це реалізує at-least-once-доставку без необхідності точно-разової семантики на сервері: у разі повторної доставки тієї ж події (наприклад, при ретраї воркера) клієнт відкине дублікат.

Ізоляція workspace забезпечується через кімнати. При підключенні клієнта виконується автентифікація через JWT, отримується workspaceId, клієнт автоматично приєднується до кімнати workspace:{workspaceId}. Перевірка членства (WorkspaceMembershipGuard) виконується для кожного WebSocket-handshake. Спроба підключитися до чужого workspace неможлива: трансляція виконується тільки у власну кімнату, без можливості клієнта обрати кімнату. Архітектуру масштабування показано на рисунку Б.37.

На рисунку Б.37 зображено три інстанси API під load-balancer, центральний Redis з чергами BullMQ і Pub/Sub-каналами адаптера та воркер-процесор. Подія, опублікована воркером у Pub/Sub, надходить до всіх інстансів API, кожен з яких транслює її у локальну копію кімнати workspace:{id}, тому оператор, підключений до будь-якого інстансу, отримує подію в реальному часі.

3.2.6. Повнотекстовий пошук діалогів і повідомлень

Для пошуку діалогів і повідомлень реалізовано повнотекстовий пошук (FTS) на PostgreSQL 16 [9]. Стратегія поєднує два механізми: native FTS через to_tsvector і GIN-індекс для збігів за словами та розширення pg_trgm для нечіткого пошуку (триграмні збіги, опечатки, ILIKE-fallback).

Замість матеріалізованої колонки застосовано **expression-індекси GIN** безпосередньо над виразом to_tsvector. Це уникає додаткової колонки й дозволяє оптимізатору використовувати індекс для запитів виду to_tsvector('simple', COALESCE("text", '')) @@ plainto_tsquery('simple', :q). Конфігурація simple обрана навмисно: вона не виконує стемінг (на відміну від english чи russian), що критично для української і змішаної української/російської мови. Перелік пошукових індексів наведено в таблиці 3.3.

Таблиця 3.3. Індекси повнотекстового пошуку у схемі PostgreSQL omnichat-hub

Таблиця	Вираз / колонка	Тип індексу	Призначення
Message	to_tsvector('simple', COALESCE(text,''))	GIN	FTS за словами повідомлення
Message	text gin_trgm_ops	GIN (trigram)	Нечіткий ILIKE-пошук тексту
Conversation	tsvector за buyerDisplayName і contextTitle	GIN	FTS за іменем покупця і назвою контексту
Conversation	buyerDisplayName gin_trgm_ops	GIN (trigram)	Нечіткий пошук за іменем покупця
Conversation	contextTitle gin_trgm_ops	GIN (trigram)	Нечіткий пошук за назвою контексту

Для Message створено FTS-індекс над to_tsvector('simple', COALESCE("text", '')) і триграмний індекс gin_trgm_ops над колонкою text. Для Conversation створено об’єднаний FTS-індекс над конкатенацією buyerDisplayName і contextTitle та окремі триграмні індекси для buyerDisplayName (ім’я покупця) і contextTitle (назва контексту). Усі індекси оформлено у міграції 20260214112000_search_indexes (повний текст доступний у репозиторії проєкту).

Модуль search у apps/api/src/modules/search об’єднує обидва механізми в ендпоінті GET /v1/search?q=..., що повертає одночасно знайдені діалоги (за іменем покупця і назвою контексту) і повідомлення (за вмістом) у двох колекціях. Запит поєднує FTS-предикат @@ plainto_tsquery('simple', :q) з ILIKE-fallback для часткових збігів і сортує результати за lastActivityAt DESC для діалогів і createdAt DESC для повідомлень.

3.2.7. Розробка алгоритмів обробки даних

Частину логіки забезпечення достовірності та цілісності даних винесено на рівень бази даних у вигляді тригерів PostgreSQL і збереженої процедури PL/pgSQL. Перенесення критичних правил у СУБД гарантує їх виконання незалежно від того, який компонент (API, воркери, ручні міграції) виконує запис. Нижче наведено два тригери підтримки деривованих полів і

дедуплікації, тригер супроводу пошукового шару та збережену процедуру обчислення щоденних агрегатів WorkspaceDailyRollup.

Тригер 1. Підтримка Conversation.lastActivityAt при вставці повідомлення. Поле Conversation.lastActivityAt визначає порядок сортування діалогів у списку оператора і використовується в індексі @@index([workspaceId, lastActivityAt]). У застосунку це поле оновлюється доменними функціями applyIncomingMessageEffects і applySellerReplySuccessEffects з пакета @omnichat/domain. Щоб гарантувати узгодженість навіть для записів в обхід доменного шару (масові міграції, бекфіли синхронізації), той самий інваріант продубльовано тригером БД. Тригер спрацьовує AFTER INSERT на Message і піднімає lastActivityAt батьківського діалогу до часу створення повідомлення, додатково оновлюючи lastIncomingAt для вхідних і lastSellerReplyAt для вихідних:

Застосування GREATEST із поточним значенням робить тригер ідемпотентним щодо порядку надходження подій: пізніше повідомлення не відкочує lastActivityAt назад. Алгоритм показано на рисунку Б.38.

Залежно від напряму повідомлення оновлюється lastIncomingAt або lastSellerReplyAt, після чого піднімається загальний lastActivityAt через GREATEST, що забезпечує монотонність значення.

Тригер 2. Дедуплікація вхідних повідомлень за зовнішнім ідентифікатором. Адаптери каналів під час синхронізації можуть повторно отримати одне й те саме повідомлення (після ретраю або повторного проходу за курсором). Базовим запобіжником є складений унікальний ключ @@unique([conversationId, externalMessageId]) у моделі Message. Проте externalMessageId опційний (String?): для вихідних повідомлень оператора він порожній і проставляється пізніше, тому UNIQUE-індекс пропускає рядки з NULL. Щоб повертати застосунку зрозумілий код помилки, реалізовано тригер BEFORE INSERT, який блокує вставку дубля лише тоді, коли externalMessageId заданий і вже існує в межах того самого діалогу:

Тригер доповнює, а не замінює унікальний індекс: індекс залишається фінальним бар’єром цілісності, а тригер дає раннє й семантично прозоре відхилення дубля (DUPLICATE_EXTERNAL_MESSAGE) ще до спроби запису, що спрощує ідемпотентну обробку у воркері синхронізації. Алгоритм показано на рисунку Б.39.

Вихідні повідомлення без зовнішнього ідентифікатора пропускаються, а для вхідних перевіряється наявність дубля в межах діалогу; за наявності дубля вставка відхиляється з кодом unique_violation.

Тригер 3. Сигналізація про зміну пошукового вмісту діалогу. Об’єднаний FTS-індекс діалогу побудовано над to_tsvector('simple', buyerDisplayName || ' ' || contextTitle). PostgreSQL автоматично перебудовує expression-індекс при зміні колонок, проте для діагностики дрейфу пошукового шару введено легкий тригер AFTER UPDATE, що надсилає асинхронне сповіщення каналом pg_notify лише тоді, коли реально змінилися поля, які впливають на пошук:

Оператор IS DISTINCT FROM коректно обробляє NULL-значення, тому сповіщення надсилається лише за фактичної зміни пошукового вмісту. Тригер не записує додаткової колонки і використовує лише наявні поля Conversation, тому не потребує змін схеми.

Збережена процедура обчислення щоденних агрегатів WorkspaceDailyRollup. Для аналітичної панелі система щодоби обчислює агреговані метрики кожного workspace: кількість вхідних (messagesIn) і вихідних (messagesOut) повідомлень, кількість створених діалогів (convCreated), середній (avgFirstResponseMs) і 95-й перцентиль (p95FirstResponseMs) часу першої відповіді в мілісекундах. У застосунку ці метрики обчислює воркер WorkspaceDailyRollupProcessor. Ту саму логіку реалізовано як процедуру PL/pgSQL fn_compute_daily_rollup(p_date date), що дозволяє виконувати агрегацію в СУБД (за розкладом pg_cron або ручним викликом):

Процедура перебирає всі workspace, для кожного рахує три лічильники простими count(*) з фільтром за добою, а час першої відповіді обчислює через подвійне JOIN LATERAL: для кожного діалогу береться найперше вхідне повідомлення доби і найперше вихідне після нього, після чого по всіх діалогах усереднюється різниця та обчислюється 95-й перцентиль через PERCENTILE_CONT. Запис виконується через INSERT ... ON CONFLICT (workspaceId, date) DO UPDATE, що робить процедуру ідемпотентною: повторний виклик за ту саму дату перерахує і перезапише агрегати без дублікатів завдяки ключу @@unique([workspaceId, date]). Алгоритм показано на рисунку Б.40.

На рисунку Б.40 зображено цикл по всіх workspace: для кожного три лічильники за добу, обчислення часу першої відповіді через подвійний LATERAL JOIN і середнє з 95-м перцентилем, після чого ідемпотентний upsert агрегатів. Реалізовані тригери цілісності, тригер супроводу пошуку та збережена процедура агрегації виконують вимогу щодо наявності засобів забезпечення достовірності й цілісності даних і обробки масивів даних на рівні СУБД.

3.2.8. Розробка фізичної моделі даних

Фізичну модель побудовано на підставі логічної моделі сутність-зв’язок з розділу 2 з урахуванням особливостей PostgreSQL 16. Схема schema.prisma декларує 24 моделі (таблиці) і 13 переліків (enum); жодна модель не має директиви @@map, тому ім’я Prisma-моделі дорівнює імені таблиці в БД. Первинний ключ майже всіх таблиць - String @id @default(cuid()); два винятки: WorkspaceSettings має первинним ключем зовнішній workspaceId (зв’язок 1:1 з Workspace), а ConversationTag - складений @@id([conversationId, tagId]).

Центральним агрегатом-коренем є таблиця Workspace: до неї з правилом onDelete: Cascade прив’язано 16 дочірніх колекцій (членства, канальні акаунти, діалоги, повідомлення, вкладення, теги, статуси, шаблони,

налаштування, push-токени, журнал аудиту, події Outbox, щоденні агрегати, ключі шифрування каналів, стани замовлень Rozetka). Правило onDelete: SetNull застосовано лише у трьох місцях: Conversation.status, Template.category і AuditLog.user. Фізичні характеристики ключових таблиць наведено в таблиці 3.4.

Таблиця 3.4. Фізична структура ключових таблиць бази даних omnichat-hub

Таблиця	Ключові поля та типи	Первинний ключ	Унікальні обмеження
User	id text, displayName text?, defaultWorkspaceId text?	id	-
Identity	provider IdentityProvider, providerAccountId text, email text?, userId text	id	(provider, providerAccountId)
Workspace	name text, maxOlxAccounts int default 1	id	-
Membership	role MembershipRole default OWNER, workspaceId, userId	id	(workspaceId, userId)
WorkspaceSettings	autoArchiveDays int default 7, notificationPrefsJson jsonb?	workspaceId	- (1:1)
ChannelAccount	channel Channel, alias text, authType ChannelAuthType, authDataEncrypted text?, isEnabled bool default true	id	-
ChannelDek	version int, encryptedDek text, kekKeyId text, isActive bool default true, rotatedAt timestamptz?	id	(workspaceId, version)
Conversation	buyerDisplayName text, needsReply bool default false, lastActivityAt timestamptz?, paymentStatus PaymentStatus?, statusId text?	id	-
ConversationTag	conversationId, tagId, assignedAt timestamptz	(conversationId, tagId)	-

Продовження табл. 3.4

Таблиця	**Ключові поля та типи**	**Первинний ключ**	**Унікальні обмеження**
Message	direction MessageDirection, text text?, externalMessageId text?, clientMessageId text?, deliveryStatus default SENT, metadata jsonb?	id	(conversationId, externalMessageId), (conversationId, clientMessageId)
Attachment	kind AttachmentKind, mimeType text, sizeBytes int, storageKey text	id	-
OutboxEvent	type text, payload jsonb, status default PENDING, lockedAt timestamptz?, lockId text?, attempts int default 0	id	-
WorkspaceDailyRollup	date timestamptz, messagesIn/Out int, convCreated int, avgFirstResponseMs int?	id	(workspaceId, date)
DevicePushToken	platform DevicePlatform, token text, isEnabled bool	id	(platform, token)

Окрему увагу приділено індексній стратегії, оскільки від неї залежить продуктивність двох найгарячіших операцій: завантаження інбоксу і повнотекстового пошуку. Для Conversation визначено складені індекси (workspaceId, lastActivityAt), (workspaceId, needsReply), (workspaceId, isArchived) і композитний (workspaceId, isArchived, needsReply, lastActivityAt) для основного запиту інбоксу. Для Message визначено (conversationId, createdAt) для завантаження стрічки діалогу та два унікальні складені ключі дедуплікації: (conversationId, externalMessageId) для анти-дублювання вхідних і (conversationId, clientMessageId) для ідемпотентності вихідних. Для OutboxEvent визначено три індекси для поллера: (workspaceId, status, nextAttemptAt), (status, nextAttemptAt) і (aggregateId, status). Поверх декларованих індексів окремою міграцією додано п’ять GIN-індексів повнотекстового та триграмного пошуку (таблиця 3.3). Узагальнену фізичну ER-діаграму ядра показано на рисунку Б.41.

На рисунку Б.41 зображено ядро фізичної моделі: агрегат Workspace з прив’язаними колекціями та ланцюг ChannelAccount → Conversation →

Message → Attachment → ShortLink, а також зв’язок багато-до-багатьох Conversation ↔ Tag через асоціативну таблицю ConversationTag. У боксах показано вибрані фізичні поля з типами PostgreSQL і позначенням первинних ключів. Повну фізичну схему всіх 24 таблиць наведено у схемі даних Prisma у відкритому репозиторії проєкту.

3.2.9. Розробка інтерфейсу користувача

Інтерфейс реалізовано у двох клієнтах зі спільною дизайн-системою: веб-дашборді оператора (apps/web, Next.js 16 App Router) і мобільному застосунку (apps/mobile, Expo SDK 53). Візуальна узгодженість гарантується пакетом packages/ui-tokens, що експортує єдиний набір дизайн-токенів (кольори, відступи, типографіку) у двох формах: Tailwind-preset для web і Tamagui-config для mobile.

Інформаційну архітектуру побудовано навколо центрального робочого процесу оператора - обробки вхідних звернень. Основні екрани веб-клієнта: сторінка автентифікації (/auth/*), список діалогів (інбокс, /inbox), картка діалогу з історією повідомлень (/conversations/[id]) і налаштування (/settings). Навігаційну схему показано на рисунку Б.42.

Після входу через email-OTP оператор потрапляє в інбокс, звідки відкриває картку діалогу, застосовує фільтри (потребує відповіді, канал, статус, теги), виконує повнотекстовий пошук або переходить у налаштування для підключення каналів і керування статусами, тегами та шаблонами.

Ключовий екран - інбокс - двопанельне компонування. Ліва панель містить список діалогів, відсортований за lastActivityAt спадно, з індикатором needsReply, іконкою каналу (OLX, Prom, Rozetka), іменем покупця (buyerDisplayName), назвою контексту (contextTitle) та прев’ю останнього повідомлення. Над списком розташовано панель фільтрів і поле пошуку, що звертається до GET /v1/search. Права панель показує обрану картку: заголовок з контекстом замовлення (за наявності - чип статусу замовлення Rozetka з поля contextStatusName), стрічку повідомлень із розрізненням напряму (вхідні IN /

вихідні OUT), статусами доставки (deliveryStatus) і вкладеннями, а внизу - поле введення відповіді з можливістю вставити шаблон і прикріпити файл.

Реалтайм-оновлення побудовано на TanStack Query: при отриманні Socket.IO-події message.new, message.updated або conversation.updated клієнт інвалідовує або точково оновлює відповідні кеші, завдяки чому список діалогів і відкрита картка оновлюються без перезавантаження. Дедуплікація за outboxEventId (підрозділ 3.2.5) гарантує, що повторно доставлена подія не призведе до дублювання. Оптимістичні оновлення застосовуються при надсиланні відповіді: повідомлення з’являється у стрічці одразу зі статусом SENDING, а після підтвердження сервером статус змінюється на SENT/DELIVERED.

Мобільний клієнт відтворює той самий робочий процес у компонуванні під сенсорний екран: список діалогів, екран окремого діалогу, форма відповіді. Додатково мобільний застосунок отримує push через expo-notifications: токени пристроїв зберігаються в таблиці DevicePushToken (поля platform, token, isEnabled, унікальний ключ (platform, token)), а доставка push виконується воркером як друга сторона доставки Outbox-події (підрозділ 3.2.2). Натискання на push відкриває відповідний діалог через глибоке посилання (expo-linking). Екранні форми обох клієнтів проілюстровано скріншотами в розділі 4 і винесено в додаток з екранними формами.

3.3. Висновки до розділу 3

У розділі 3 виконано всі задачі, поставлені у розділі 2: обґрунтовано вибір інструментальних засобів для кожного шару системи, спроєктовано та реалізовано основні підсистеми omnichat-hub, описано ключові архітектурні рішення, розроблено алгоритми обробки даних на рівні бази даних, фізичну модель даних і інтерфейс користувача.

В обґрунтуванні вибору розглянуто понад 30 технологій з академічним порівнянням альтернатив за критеріями стабільності, типобезпеки, ціни,

екосистеми та операційної простоти, у тому числі окреме порівняння трьох реляційних СУБД клієнт-серверного типу. Обрано стек, центрований навколо TypeScript: Node.js 22 LTS, NestJS, Prisma, PostgreSQL 16 з вбудованим FTS, JSONB, тригерами і збереженими процедурами, Redis 7 як універсальний компонент, Socket.IO з Redis-адаптером, BullMQ, Next.js 16 з React 19, Expo SDK 53, Pulumi, GitHub Actions, Sentry і Grafana Cloud.

Описано структуру pnpm-monorepo з чотирма додатками (apps/api, apps/workers, apps/web, apps/mobile) і шістьма основними спільними пакетами (api-client, contracts, domain, db, config, ui-tokens), а також додатковим пакетом kms. Детально розкрито чотири ключові підсистеми.

Перша підсистема - транзакційний Outbox-патерн. Атомарна транзакція БД одночасно записує Message, оновлює деривовані поля Conversation і створює OutboxEvent зі статусом PENDING. Фоновий OutboxPoller опитує таблицю раз на 60 секунд і ставить прострочені події у BullMQ-чергу. Захоплення події виконується не через FOR UPDATE SKIP LOCKED, а через оптимістичне захоплення засобами Prisma - атомарний умовний updateMany з полями lockId і lockedAt. Подія зі статусом PROCESSING, чий lockedAt старший за поріг OUTBOX_LOCK_TTL_SEC (за замовчуванням 300 секунд), повторно потрапляє у вибірку і перезахоплюється. Дві незалежні сторони доставки (wsDeliveredAt, pushDeliveredAt) виставляються ідемпотентно; лише після обох подія переходить у PROCESSED. Це забезпечує гарантовану доставку at-least-once.

Друга підсистема - OAuth 2.0 інтеграція з OLX за стандартом Authorization Code. Реалізовано безпечний state-обмін через Redis з TTL 600 секунд і атомарним read-and-delete проти replay-атак, обмін code на пару access_token + refresh_token через token-ендпоінт https://www.olx.ua/api/open/oauth/token зі scope read write v2, отримання профілю продавця через /api/partner/users/me, шифрування токенів через envelope encryption перед записом у authDataEncrypted таблиці ChannelAccount. На відміну від OLX, Rozetka і Prom використовують

статичний API-токен, який користувач отримує в кабінеті продавця маркетплейсу.

Третя підсистема - обгорткове шифрування у пакеті packages/kms. Реалізовано двошарову схему: DEK на рівні workspace для шифрування payload через AES-256-GCM з випадковим IV (12 байт) і auth tag, та KEK, що ніколи не залишає зовнішнього Infisical KMS і відомий коду лише за ідентифікатором kekKeyId. Зашифрований DEK і його версії зберігаються в таблиці ChannelDek (поля encryptedDek, kekKeyId, version, isActive, rotatedAt). Підтримка версій DEK дозволяє ротацію без перешифрування історичних даних. Шифротекст має формат v1:version:iv:tag:ciphertext.

Четверта підсистема - реалтайм-шар Socket.IO з Redis-адаптером. Архітектура дозволяє горизонтально масштабувати кількість інстансів apps/api за load-balancer DigitalOcean без втрати реалтайм-семантики. Воркери публікують події через Redis emitter у Pub/Sub-канал, усі інстанси API отримують подію і транслюють її клієнтам у власній копії кімнати workspace:{id}. Ізоляція workspace гарантована перевіркою членства при WebSocket-handshake. Дедуплікація на клієнті виконується через outboxEventId.

Окрім чотирьох центральних підсистем, описано повнотекстовий пошук діалогів і повідомлень: для Message і Conversation створено GIN-індекси над виразом to_tsvector('simple', ...) і триграмні індекси gin_trgm_ops (розширення pg_trgm) для нечіткого пошуку за іменем покупця та назвою контексту. Конфігурацію simple обрано без стемінгу, що критично для української мови.

У підрозділі алгоритмів обробки даних на рівні СУБД реалізовано і проілюстровано блок-схемами два тригери цілісності (підтримка Conversation.lastActivityAt при вставці повідомлення та дедуплікація вхідних повідомлень за externalMessageId), тригер супроводу пошукового шару та збережену процедуру PL/pgSQL fn_compute_daily_rollup для ідемпотентного обчислення щоденних агрегатів WorkspaceDailyRollup (лічильники повідомлень і діалогів, середній і 95-й перцентиль часу першої відповіді через подвійний JOIN LATERAL і PERCENTILE_CONT).

У підрозділі фізичної моделі даних описано фізичну структуру 24 таблиць і 13 переліків схеми, агрегат-корінь Workspace з 16 каскадними дочірніми колекціями, два винятки з cuid-ключа (WorkspaceSettings і ConversationTag), індексну стратегію та узагальнену фізичну ER-діаграму ядра. У підрозділі інтерфейсу користувача наведено інформаційну архітектуру і навігаційну схему екранів веб-клієнта, двопанельне компонування інбоксу, реалтайм-оновлення на TanStack Query з оптимістичними апдейтами та спільну дизайн-систему packages/ui-tokens для web і mobile.

Ключові архітектурні складнощі реалізації:

Перша - три різні моделі автентифікації каналів. OLX використовує OAuth 2.0 з потоком Authorization Code і refresh-токенами; Rozetka і Prom - статичні API-токени без часу життя. Це призвело до необхідності винести канал-специфічну логіку в окремі модулі, кожен зі своєю стратегією отримання, оновлення і шифрування токенів [4]. Спільна абстракція ChannelAccount містить лише універсальні поля channel, authType і authDataEncrypted, а канал-специфічний формат payload (наявність refreshToken лише для OLX) обробляється відповідним модулем.

Друга - дедуплікація реалтайм-подій. Outbox-патерн природно дає at-least-once семантику: за умов краху воркера після відправки події, але до позначення її як PROCESSED, ця подія буде відправлена повторно [6]. Розв’язання - перенесення дедуплікації на клієнтську сторону через outboxEventId. Цей підхід простіший в реалізації і відмовостійкіший за точно-разову семантику на сервері.

Третя - атомарність криптографічних операцій при ротації DEK. Період між створенням нового DEK і записом його у БД є моментом потенційного race condition: одночасний запит міг би прочитати старий активний DEK і записати ciphertext, незрозумілий для нового активного DEK. Розв’язання - версіонування DEK у самому форматі ciphertext (другий сегмент version) і збереження всіх історичних версій DEK у ChannelDek, що дозволяє розшифровувати будь-який ciphertext незалежно від поточної активної версії.

Четверта - синхронізація типів між backend і клієнтами. Без єдиного джерела правди контракту виникає drift між API і клієнтами. Розв’язання - застосування OpenAPI 3.1 як єдиного джерела правди [18]. NestJS-декоратори @ApiResponse, @ApiBody, @ApiTags декларативно описують структуру кожного ендпоінту. Команда pnpm api-client:generate витягує з API повний OpenAPI-документ і генерує TypeScript-клієнт у пакеті @omnichat/api-client. У CI крок openapi-drift-check порівнює актуальний контракт з закомміченим: будь-яка зміна API без оновлення згенерованого клієнта блокує merge.

П’ята - управління секретами у monorepo з декількома середовищами. У omnichat-hub задіяні десятки секретів: ідентифікатор KEK, OLX client_id і client_secret, JWT signing keys, DSN Sentry, токени Grafana Cloud, ключі DigitalOcean API. У локальній розробці секрети живуть у .env.local файлах (виключених з git через .gitignore). У production секрети зберігаються як Pulumi stack secrets з шифруванням passphrase-ключем. Заборона на запис секрету у git enforced через pre-commit hook (detect-secrets).

Шоста - ізоляція тестового середовища. Інтеграційні тести API виконують реальні Prisma-запити проти Postgres у Docker-контейнері, що піднімається у CI окремо для кожного PR. E2E-тести стартують повний стек (API + Workers + Web) у Docker Compose та виконують сценарії «логін → підключити канал → відправити повідомлення → перевірити доставку». Для прискорення OUTBOX_POLL_INTERVAL_SEC у тестовому режимі знижений до 1 секунди.

Результатом реалізації стала система, готова до подальшого розгортання у виробничому середовищі. Архітектурні рішення зафіксовано у документації docs/architecture/, що включає опис кожного модуля, ER-діаграму бази даних і sequence-діаграми ключових сценаріїв (логін через email-OTP, підключення OLX через OAuth, доставка повідомлення end-to-end). У наступному розділі описано процес тестування, роботу користувача з системою та розгортання omnichat-hub на DigitalOcean App Platform і Vercel. Локальне середовище

розробки, що паралельно запускає всі додатки monorepo, показано на рисунку Б.43.

РОЗДІЛ 4. ТЕСТУВАННЯ ТА ВПРОВАДЖЕННЯ

4.1. Тестування системи

Тестування веб-системи omnichat-hub виконано для верифікації функціональних і нефункціональних вимог розділів 1-2. Як SaaS-продукт omnichat-hub працює у розподіленому середовищі (Node.js 22 LTS [22] на DigitalOcean App Platform [25], PostgreSQL 16 [9], Redis [16], Socket.IO з Redis-адаптером [14]), тому окремі модулі неможливо повноцінно перевірити без піднятого інфраструктурного стека. Це визначило багаторівневу стратегію тестування, у якій кожен рівень покриває власну категорію дефектів і відповідає конкретному рівню архітектурної ізоляції коду.

4.1.1. Розробка тест-плану

На підставі вимог, проектних специфікацій і визначеної у розділах 2-3 архітектури розроблено формальний тест-план, що фіксує три складові: об’єкти тестування, рівні тестування і критерії проходження. Кожний тест-приклад описано за єдиним шаблоном (ідентифікатор, рівень, передумова, вхідні дані, кроки, очікуваний результат і трасування на функціональну вимогу розділів 1-2), що забезпечує двобічну простежуваність: для кожної вимоги визначено щонайменше один тест-приклад. Повний тест-план з матрицею трасування «функціональна вимога - тест-приклад» наведено в додатку А.

Об’єкти тестування. Об’єктами є артефакти кодової бази omnichat-hub за рівнями архітектурної ізоляції:

- доменний шар packages/domain (чисті функції бізнес-правил);
- криптографічний пакет packages/kms (envelope encryption, AES-256-GCM, керування DEK);
- HTTP-клієнт packages/api-client (рефреш-логіка на коді 401);

- воркери apps/workers (Outbox-поллер і процесор, канал-синхронізація, push);
- HTTP-контракт REST API apps/api (ендпоінти /v1, ізоляція за workspace);
- реалтайм-канал Socket.IO (доставка подій через Redis-адаптер);
- веб-інтерфейс оператора apps/web (екранні форми інбоксу, налаштувань, пошуку).

Рівні тестування. Обрано чотирирівневу стратегію, у якій кожний наступний рівень розширює межі перевірки від ізольованої функції до повного розподіленого стека. Відповідність рівнів об’єктам, інструментам і категоріям дефектів наведено в таблиці 4.1.

Таблиця 4.1. Рівні тестування системи omnichat-hub

№	Рівень	Об'єкт перевірки	Інструмент	Категорія дефектів
1	Модульне (юніт)	packages/domain, packages/kms, packages/api-client, окремі сервіси apps/workers	Vitest	помилки в бізнес-правилах, крипто-перетвореннях, рефреш-логіці
2	Тестування інтерфейсу	екранні форми apps/web (інбокс, налаштування каналів, пошук)	Vitest + Testing Library, ручні чек-листи	помилки рендерингу, станів форм, навігації
3	Інтеграційне	взаємодія apps/api з PostgreSQL через Prisma [13], постановка задач у BullMQ [15]	Vitest (HTTP над піднятим стеком)	помилки транзакційності, контракту БД, конфігурації черг
4	Системне (E2E)	повний стек apps/api + apps/workers + PostgreSQL + Redis через HTTP і WebSocket	Vitest (black-box tests/e2e)	помилки наскрізних сценаріїв, реалтайм-доставки, ізоляції workspace

Критерії проходження. Тест-план визначає критерії проходження для кожного рівня:

- модульний рівень: усі юніт-тести зеленого статусу; цільове покриття 100 відсотків правил доменного шару packages/domain;
- рівень інтерфейсу: рендеринг ключових екранних форм без помилок, коректна зміна станів, відсутність регресій у навігації;

- інтеграційний рівень: запис Message і OutboxEvent в одній транзакції, успішне підняття черг BullMQ, відповідність HTTP-контракту згенерованому артефакту OpenAPI [18];
- системний рівень: проходження наскрізних тест-кейсів TC-01 - TC-12, підтверджена доставка подій message.new, message.updated, conversation.updated через Socket.IO, повернення HTTP 403 на спробу доступу до чужого workspace.

Загальний критерій готовності до захисту - проходження всіх рівнів 1-4 у безперервній інтеграції (GitHub Actions) без жодного червоного статусу і відсутність відомих дефектів категорії «блокер».

Пріоритезацію покриття побудовано за принципом ризику: найщільнішим покриттям забезпечено компоненти, відмова яких найбільше впливає на цілісність даних і безпеку, - транзакційний запис Message і OutboxEvent, механізм відновлення Outbox-поллера, ізоляцію даних між workspace і потік авторизації каналів. Критерій входу в кожний рівень - успішне проходження попереднього рівня; критерій виходу - відсутність відкритих дефектів категорій «блокер» і «критичний» та виконання кількісного критерію покриття. Виявлені дефекти фіксуються, усуваються і покриваються регресійним тест-прикладом.

4.1.2. Модульне тестування

Модульне тестування виконано засобами Vitest і зосереджено на доменному шарі packages/domain - чистих функціях бізнес-правил без побічних ефектів. Ці функції визначають, чи потребує діалог відповіді (computeNeedsReply), як змінюється стан діалогу при вхідному повідомленні (applyIncomingMessageEffects) та при вихідній відповіді оператора (applySellerReplySuccessEffects, applySellerReplyFailedEffects), коли діалог підлягає автоархіву (canAutoArchive, applyAutoArchiveEffects), правило максимум трьох закріплених діалогів у вкладці All (validatePinnedCount, validateCanPinInFolder) і коли слід придушити push-сповіщення

(shouldSuppressPush). Пакет не залежить від Prisma, Redis чи Socket.IO, тому тестується ізольовано [5]. Цільовий показник покриття - 100 відсотків правил доменного шару packages/domain (не всієї кодової бази), що досягнуто на момент захисту; покриття вимірюється вбудованим інструментом Vitest, зберігається як артефакт безперервної інтеграції, а його зниження блокує злиття змін.

Модульними тестами також покрито криптографічний пакет packages/kms (aes-gcm.test.ts, envelope-crypto.test.ts, dek-manager.test.ts, ciphertext.test.ts, infisical-kms-client.test.ts) у частині шифрування AES-256-GCM, формату шифротексту, генерації та ротації Data Encryption Key і виявлення підробки ciphertext; пакет packages/api-client у частині рефреш-логіки при HTTP-коді 401 [8]; сервіси apps/workers у частині Outbox-поллера і процесора (outbox.poller.test.ts, outbox.processor.test.ts), канал-синхронізації та push-доставки.

Модульні тести зосереджені на крайніх випадках доменних правил; ключові випадки пакета packages/domain наведено в таблиці 4.2.

Типовою категорією дефектів на цьому рівні були помилки граничних умов: некоректна обробка стану, коли остання активність припадає рівно на межу інтервалу автоархіву, і подвійне застосування ефекту вихідної відповіді при повторному виклику функції. Обидва класи усунуто додаванням явних граничних тест-прикладів, після чого функції доменного шару стали ідемпотентними щодо повторного застосування. Чистота доменних функцій робить ці тести детермінованими, що спрощує локалізацію регресій.

Таблиця 4.2. Ключові модульні тести доменного шару

№	Модуль	Перевірка	Очікуваний результат
1	computeNeedsReply	Вхідне повідомлення від покупця після відповіді оператора	needsReply=true
2	computeNeedsReply	Дія markAsRead без надсилання відповіді	needsReply залишається true
3	applySellerReplySuccessEffects	Успішна вихідна відповідь оператора	needsReply=false
4	validatePinnedCount	Спроба закріпити 4-й діалог у All при 3 закріплених	викидається MAX_PINNED_REACHED
5	validateCanPinInFolder	Закріплення діалогу у недозволеній вкладці	викидається помилка валідації
6	shouldSuppressPush	Вхідне повідомлення під час активного snooze	push не надсилається
7	applyIncomingMessageEffects	Закінчення терміну snooze і нове повідомлення	діалог повертається в активний стан
8	canAutoArchive	Діалог з lastSellerReplyAt старше autoArchiveDays і needsReply=false	дозволено автоархів
9	applyAutoArchiveEffects	Діалог з тегом важливості	автоархів пропускається
10	applyIncomingMessageEffects	Нове вхідне повідомлення в архівованому діалозі	діалог автоматично розархівовується

4.1.3. Тестування інтерфейсу користувача

Тестування інтерфейсу спрямовано на екранні форми веб-дашборду apps/web, побудованого на Next.js 16 з React 19 [12]. Тест-приклади покривають основні елементи інтерфейсу оператора: трисекційний екран інбоксу, форми підключення каналів у розділі Settings, форму пошуку та модальні вікна роботи з шаблонами, тегами і статусами; їх наведено в додатку А.

Перевірку виконано двома способами. Компонентні тести (Vitest у поєднанні з бібліотекою тестування React-компонентів) перевіряють рендеринг і коректність зміни станів: вибір діалогу завантажує історію в область чату, надсилання відповіді переводить повідомлення у стан «надсилається» і потім «надіслано», перемикання вкладок All, Unread, Channel, Snoozed, Archived змінює фільтрацію списку. Ручні чек-листи перевіряють аспекти, які важко формалізувати автоматично: візуальну узгодженість трьох колонок, читабельність маркерів каналів, доступність керування з клавіатури, відображення станів завантаження та порожніх

списків, адаптивність на настільному і планшетному екранах, стан повторного підключення WebSocket і збереження позиції прокручування при надходженні нового повідомлення.

У таблиці 4.3 наведено ключові тест-кейси інтерфейсу користувача.

Таблиця 4.3. Ключові тест-кейси інтерфейсу користувача

№	Ідентифікатор	Елемент інтерфейсу	Очікуваний результат
1	UI-01	Список діалогів	Діалоги відсортовано за lastActivityAt, видно маркер каналу і прев'ю останнього повідомлення
2	UI-02	Вибір діалогу	Завантажується історія, область чату відображає повідомлення у хронологічному порядку
3	UI-03	Надсилання відповіді	Повідомлення з'являється у стані «надсилається», потім переходить у «надіслано»
4	UI-04	Перемикання вкладок	Вкладки All/Unread/Channel/Snoozed/Archived коректно фільтрують список
5	UI-05	Панель метаданих	Поля ТТН, статусу оплати, тегів і статусу редагуються і зберігаються
6	UI-06	Форма пошуку	Запит повертає релевантні діалоги і повідомлення, за замовчуванням у межах recent-window

Основними виявленими дефектами були неузгодженість стану між списком діалогів і областю чату при швидкому перемиканні (race condition у клієнтському кеші) та відсутність явного стану порожнього списку у вкладці Snoozed. Перший дефект усунуто стабільним ключем діалогу й скиданням стану при зміні вибору, другий - компонентом порожнього стану. Виявлені дефекти підтвердили цінність поєднання компонентних тестів і ручних чек-листів.

4.1.4. Інтеграційне та системне тестування

Інтеграційне тестування. На інтеграційному рівні перевіряється взаємодія NestJS-контролерів [11] з базою даних через Prisma [13], генерація записів OutboxEvent у тій самій транзакції, що й Message, і підняття черг BullMQ [15]. Тести виконуються з реальним PostgreSQL 16 і Redis у Docker Compose (infra/compose) командою pnpm test:e2e у режимі NODE_ENV=test, у

якому автоматично застосовуються ізольовані префікси BullMQ (BULLMQ_PREFIX) і ключі каналів Socket.IO у Redis (SOCKETIO_REDIS_KEY) для уникнення колізій з dev-середовищем.

Окрему групу тестів присвячено транзакційній цілісності: запис Message і відповідного OutboxEvent відбувається в одній транзакції, і за штучного збою на етапі формування події транзакція відкочується повністю. Перевіряється ідемпотентність обробки вхідних повідомлень: повторне надходження повідомлення з тим самим зовнішнім ідентифікатором не створює дубля завдяки унікальному обмеженню (conversationId, externalMessageId). Життєвий цикл задачі BullMQ перевіряється від постановки у чергу outbox.process до успішного завершення з видаленням за прапорцями removeOnComplete і removeOnFail.

Системне (наскрізне) тестування. Системне тестування покриває повний потік користувача через реальні межі HTTP і WebSocket. Наскрізні тести реалізовано як чорноскринькові (black-box) тести на Vitest у пакеті tests/e2e, які звертаються до запущеного стека так само, як зовнішній клієнт. Тести охоплюють домени auth, health, conversations, messages, search, tags, statuses, templates, settings, shortlinks і ws, перевіряючи насамперед доставку подій message.new, message.updated і conversation.updated через WebSocket (для чого потрібні запущені воркери). Сценарій включає реєстрацію через email-OTP, створення workspace, підключення каналу, отримання вхідного повідомлення, надсилання вихідної відповіді і верифікацію доставки події клієнту через Socket.IO без перезавантаження сторінки. Окремо перевіряються негативні сценарії: доступ до чужого workspace, повторне використання спожитого OTP-коду і звернення до захищеного ендпоінта без access-токена.

У таблиці 4.4 наведено основні наскрізні тест-кейси системного рівня.

Таблиця 4.4. Основні наскрізні (E2E) тест-кейси omnichat-hub

№	Ідентифікатор	Сценарій	Очікуваний результат
1	TC-01	Логін через email + OTP-код	Видається JWT access-токен (15 хв) і refresh-токен (30 діб), HTTP 200
2	TC-02	Підключення Rozetka через статичний API-токен	ChannelAccount створено, токен зашифровано AES-256-GCM, статус active
3	TC-03	Підключення OLX через OAuth 2.0 потік Authorization Code	callback повернув code, обміняно на access+refresh, статус active
4	TC-04	Підключення Prom через статичний API-токен	ChannelAccount створено, токен збережено, статус active
5	TC-05	Отримання вхідного повідомлення з каналу через інкрементальний sync	Message створено, OutboxEvent типу message.new згенеровано
6	TC-06	Надсилання вихідної відповіді оператора	Message створено зі статусом SENDING, через адаптер надіслано, статус оновлено на SENT
7	TC-07	Realtime-доставка message.new через Socket.IO	Клієнт отримав подію без перезавантаження сторінки в реальному часі
8	TC-08	Закріплення трьох діалогів у вкладці All	Прикріплено 3, спроба прикріпити 4-й повертає MAX_PINNED_REACHED
9	TC-09	Автоархів діалогу при needsReply=false і відсутності закріплення	Через autoArchiveDays після останньої відповіді діалог переходить у archive
10	TC-10	Повнотекстовий пошук по історії повідомлень	Запит повертає релевантні повідомлення і діалоги, recent-window за замовчуванням, прапорець showOlder розширює діапазон
11	TC-11	Ізоляція workspace за підробленим JWT	API повертає HTTP 403 FORBIDDEN на всіх захищених ендпоінтах
12	TC-12	Оновлення статусу доставки повідомлення	PATCH повідомлення генерує OutboxEvent message.updated, UI отримує патч у реальному часі

Інтеграційне та системне тестування проводилось у відтворюваному середовищі з репозиторного опису infra/compose (PostgreSQL 16, Redis 7, об’єктне сховище MinIO). API і воркери запускаються локально в режимі NODE_ENV=test; перед запуском база приводиться до відомого стану застосуванням міграцій Prisma і завантаженням детермінованого набору демо-даних, що гарантує відтворюваність результатів. Версії інфраструктурних

компонентів зафіксовано в infra/compose. Повний опис системного середовища тестування, включно з версіями компонентів, змінними оточення і послідовністю запуску, наведено в додатку А.

Ключовою категорією дефектів цього рівня були колізії у спільному просторі імен Redis при одночасному запуску dev- і тестового стеків та неузгодженість згенерованого OpenAPI-контракту з реальними декораторами контролерів. Перший клас усунуто ізоляцією черг і каналів за NODE_ENV=test, другий - впровадженням у безперервну інтеграцію перевірки дрейфу OpenAPI (openapi-drift-check). Виявлення обох класів саме на системному рівні підтвердило потребу у наскрізних тестах над повним стеком: на нижчих рівнях ізоляції ці дефекти не проявляються.

4.1.5. Сценарій стрес-тестування Outbox-патерну на витривалість

Транзакційний Outbox-патерн [1] є ключовим механізмом гарантованої доставки повідомлень в omnichat-hub, тому для нього окремо розроблено сценарій стрес-тестування на витривалість з цільовим обсягом 600 000 подій і штучною зупинкою worker-процесу під час обробки черги. Мета - підтвердити, що жодна подія не втрачається навіть за збою воркера, відповідно до семантики at-least-once delivery [6].

Логіку відновлення покрито інтеграційними та модульними тестами процесора й поллера (outbox.processor.test.ts, outbox.poller.test.ts). Після перезапуску worker повторно підбирає події у статусі PROCESSING, чий lockedAt старіший за поріг блокування OUTBOX_LOCK_TTL_SEC (за замовчуванням 300 секунд), і повертає їх в обробку через атомарне умовне захоплення (Prisma updateMany з умовним WHERE та полями lockId/lockedAt). Окремого переходу статусу PROCESSING у PENDING не виконується: застаріла за блокуванням подія знову потрапляє у вибірку поллера й перезахоплюється процесором, після чого обидві сторони доставки (WebSocket через wsDeliveredAt і push через pushDeliveredAt) виконуються ідемпотентно. Клієнтську дедуплікацію за outboxEventId також покрито

тестами: навіть за повторної доставки клієнтський інтерфейс не показує дубль повідомлення.

Повний прогін на цільовому обсязі 600 000 подій разом з формалізованим навантажувальним тестуванням засобами k6 та Artillery заплановано на третій квартал 2026 року після стабілізації MVP, що узгоджується з цільовими показниками таблиці 4.5. Показник «нуль втрачених подій з 600 000» є заявленою архітектурною характеристикою, що випливає з властивостей транзакційного запису і механізму повторного захоплення, а не результатом виконаного прогону. Розмежування двох видів покриття є принциповим: функціональну коректність механізму відновлення підтверджено модульними та інтеграційними тестами, тоді як кількісні характеристики витривалості під навантаженням підлягають окремій верифікації у плановому циклі.

Цільові показники продуктивності і пропускної здатності отримано розрахунково з властивостей обраних компонентів стека (Node.js 22 [22], PostgreSQL 16 [9], Redis [16], Socket.IO з Redis-адаптером [14]). У таблиці 4.5 наведено ці показники, які стануть основою для майбутніх навантажувальних тестів.

Таблиця 4.5. Цільові показники продуктивності

№	Показник	Цільове значення	Обґрунтування
1	End-to-end p99 latency на POST /v1/messages	до 800 мс	З урахуванням мережевих хопів DigitalOcean Frankfurt і Vercel Edge
2	Цільовий load на API	1000 RPS на інстанс	Розрахунок з performance-характеристик NestJS на Fastify
3	Outbox-throughput	від 5000 подій/хв на одного worker	Baseline BullMQ і Redis 7
4	Втрати Outbox-подій	0 з 600 000	Заявлена характеристика at-least-once delivery
5	Socket.IO concurrent connections	10 000 на API-інстанс	З Redis-адаптером для горизонтального масштабування
6	Інкрементальний sync каналів	60 секунд	Період полінга для Rozetka, Prom і OLX
7	Cold start API	до 3 секунд	Node.js 22 і NestJS bootstrap на DigitalOcean App Platform
8	Memory footprint API на інстанс	до 512 MB	Базовий ліміт DigitalOcean App Platform Basic

4.1.6. Допоміжні види тестування і безперервна інтеграція

Безпекове тестування. Окремо перевірено реалізацію workspace-membership guard: наскрізний тест з підробленим JWT-токеном, що містить чужий workspaceId, отримує HTTP 403 FORBIDDEN на всіх захищених ендпоінтах, що підтверджує ізоляцію даних між workspace. Аналогічно Socket.IO-з’єднання з чужим workspace-токеном відхиляється на етапі handshake.

Тестування рефреш-логіки JWT. Пакет packages/api-client має вбудовану логіку перехоплення HTTP-коду 401: при отриманні 401 клієнт автоматично виконує POST /v1/auth/refresh, отримує нову пару access+refresh і повторює оригінальний запит [8]. Модульні тести покривають крайні випадки: 401 без refresh-токена (повертається оригінальна помилка), невдалий refresh (перенаправлення на логін), паралельні запити з одночасним 401 (виконується лише один refresh, інші чекають його результату).

Регресійне тестування. Повний набір модульних і наскрізних тестів повторно виконується на кожний Pull Request і на кожне злиття у гілку main, перетворюючи тестовий набір на регресійний бар’єр. Перевірка дрейфу

OpenAPI виконує роль контрактного регресійного тесту: розбіжність між згенерованим зі схеми контролерів артефактом і закомміченим артефактом не допускається до злиття, що захищає згенерований клієнт packages/api-client від неузгодженості з API.

Безперервна інтеграція. Усі тести інтегровано у GitHub Actions workflow .github/workflows/ci.yml. На кожен Pull Request виконується: lint (ESLint і Prettier), typecheck (TypeScript strict mode), модульні тести (Vitest), перевірка дрейфу OpenAPI і build усіх застосунків. На push у гілку main додатково виконуються інтеграційні тести з реальним PostgreSQL і Redis у сервісах GitHub Actions і наскрізний смоук-тест над піднятим стеком. Звіти про покриття зберігаються як артефакти безперервної інтеграції.

Результати функціонального тестування підтвердили відповідність omnichat-hub функціональним вимогам розділів 1-2: сценарії TC-01 - TC-12 проходять у безперервній інтеграції на GitHub Actions, модульні тести досягають цільового покриття 100 відсотків бізнес-правил доменного шару, інтеграційні тести підтверджують коректність транзакційного запису Message та OutboxEvent. Нефункціональні цільові показники з таблиці 4.5 закладено в архітектурний дизайн і буде верифіковано у плановому циклі навантажувального тестування третього кварталу 2026 року.

Запуск модульних тестів доменного шару показано на рис. 4.1.

```
Terminal — pnpm test

$ pnpm --filter @omnichat/domain test

DEV vitest 3.2.4  packages/domain

✓ src/conversation/conversation-aggregate.spec.ts   (18)  42ms
✓ src/conversation/message-status-fsm.spec.ts       (24)  38ms
✓ src/channel/olx-message-normalizer.spec.ts        (16)  31ms
✓ src/channel/rozetka-message-normalizer.spec.ts    (14)  26ms
✓ src/channel/prom-message-normalizer.spec.ts       (14)  25ms
✓ src/outbox/outbox-event-aggregate.spec.ts         (22)  45ms
✓ src/membership/role-policy.spec.ts                (20)  22ms
✓ src/kms/dek-rotation.spec.ts                      (11)  35ms
✓ src/workspace/workspace-aggregate.spec.ts         (18)  29ms

 Test Files  10 passed (10)
      Tests  169 passed (169)
   Duration  2.31s

 Coverage  Statements 100%  Branches 100%  Functions 100%  Lines 100%
```

Рисунок 4.1. Запуск модульних тестів Vitest для доменного шару omnichat-hub

HTML-звіт наскрізних тестів системного рівня показано на рис. 4.2.

```
Terminal — Playwright E2E

$ pnpm --filter @omnichat/web playwright test

Running 12 tests using 3 workers

✓ TC-01  Логін через email + OTP-код                    (1.4s)
✓ TC-02  Підключення Rozetka (статичний API-токен)      (2.1s)
✓ TC-03  Підключення OLX через OAuth 2.0                 (3.8s)
✓ TC-04  Підключення Prom (статичний API-токен)         (2.0s)
✓ TC-05  Отримання вхідного повідомлення                (2.5s)
✓ TC-06  Надсилання вихідної відповіді                  (2.3s)
✓ TC-07  Realtime-доставка message.new (Socket.IO)      (3.2s)
✓ TC-08  Закріплення трьох діалогів (ліміт 3)           (1.7s)
✓ TC-09  Автоархів діалогу при needsReply=false         (2.6s)
✓ TC-10  Повнотекстовий пошук по історії                (1.7s)
✓ TC-11  Ізоляція workspace за підробленим JWT (403)    (1.2s)
✓ TC-12  Оновлення статусу доставки повідомлення        (2.4s)

  12 passed (27s)
```

Рисунок 4.2. Звіт наскрізних (E2E) тестів після виконання сценаріїв TC-01 - TC-12

4.2. Робота користувача з системою

Цей підрозділ є коротким керівництвом для кінцевого користувача omnichat-hub, орієнтованим на продавця-ФОП, який працює одночасно на трьох українських маркетплейсах. Детальне керівництво користувача з повним набором екранних форм наведено в додатку.

Крок 1. Реєстрація і створення workspace. Користувач обирає реєстрацію за email, отримує одноразовий пароль (OTP) і входить у систему. Майстер створення workspace пропонує задати назву організації, обрати тариф і запросити інших операторів за email. Workspace є ізольованою організацією-клієнтом: усі канали, діалоги і повідомлення прив’язані до workspace через workspaceId.

Крок 2. Підключення першого каналу. Найпростіший канал для першого підключення - Rozetka, оскільки автентифікація відбувається через статичний API-токен, який користувач отримує в кабінеті продавця Rozetka [27]. Користувач переходить у Settings - Channels, обирає Rozetka і вставляє токен у форму. omnichat-hub шифрує токен через AES-256-GCM у пакеті packages/kms із застосуванням Data Encryption Key (DEK) на рівні workspace і зберігає у зашифрованому полі authDataEncrypted таблиці ChannelAccount. Через 60 секунд починається інкрементальний sync, і у вкладці Inbox з’являються перші діалоги.

Крок 3. Підключення OLX через OAuth 2.0. OLX вимагає OAuth 2.0 з типом Authorization Code [7]. Користувач натискає Connect OLX, omnichat-hub генерує URL авторизації на https://www.olx.ua/oauth/authorize з параметрами client_id, redirect_uri, state, response_type=code і скоупом read write v2. Після підтвердження доступу OLX повертає користувача на ендпоінт /v1/channel-accounts/olx/callback з тимчасовим кодом, який обмінюється на https://www.olx.ua/api/open/oauth/token на access-токен і refresh-токен [28]. Refresh-токен зберігається у тому ж форматі обгорткового шифрування

(envelope encryption), що й токени Rozetka, і автоматично використовується при отриманні HTTP-коду 401 від OLX API.

Крок 4. Огляд інбоксу. Розділ Inbox розділений на три колонки. Ліва - список діалогів, відсортованих за останньою активністю, з кольоровим маркером каналу (Rozetka, Prom, OLX), позначкою діалогів, що потребують відповіді, і прев'ю останнього повідомлення. Середня - розмова з покупцем у форматі чату. Права - метадані діалогу: контактна інформація покупця, теги, статус, замітки оператора, номер ТТН Нової Пошти, статус оплати, дата follow-up. Над списком розташовані вкладки All, Unread, Channel, Snoozed, Archived для фільтрації.

Крок 5. Відповідь на повідомлення покупця. Користувач обирає діалог, набирає текст відповіді і натискає Send. API записує Message у БД, оновлює деривовані поля Conversation (needsReply, lastSellerReplyAt) і створює OutboxEvent в одній транзакції. Значення needsReply обчислюється не контролером напряму, а чистою доменною функцією applySellerReplySuccessEffects з пакета packages/domain, яка повертає needsReply=false після успішної відповіді; контролер лише зберігає результат у тій самій транзакції. Воркер apps/workers забирає подію з черги BullMQ і викликає адаптер відповідного каналу. При успіху статус Message змінюється на SENT, при помилці - на FAILED з можливістю ретраю через POST /v1/messages/:messageId/retry.

Крок 6. Пошук по історії повідомлень. У розділі Search користувач вводить запит, який передається на GET /v1/search. Реалізація використовує повнотекстовий пошук PostgreSQL на основі to_tsvector з GIN-індексами по тексту повідомлень і полях діалогу, а також модуль pg_trgm з триграмними індексами для нечіткого пошуку і ILIKE-резерву [9]. За замовчуванням пошук обмежено останніми добами (recent-window), а прапорець showOlder розширює діапазон на повну історію.

Крок 7. Мобільний клієнт і push-сповіщення. Мобільний застосунок розроблено на Expo SDK 53 (React Native) [19] для iOS та Android; вхід

виконується через той самий email-OTP. У Settings - Push користувач надає дозвіл на сповіщення, після чого Expo Notifications реєструє пристрій у Apple APNs або Google FCM, а omnichat-hub зберігає push-токен у таблиці DevicePushToken. При надходженні нового повідомлення воркер ставить у чергу BullMQ задачу доставки push, яка через Expo Push API надсилає сповіщення. Натискання сповіщення відкриває застосунок одразу у потрібному діалозі (deep-link з холодного старту і з фонового режиму).

Головний екран веб-дашборду показано на рисунку 4.3.

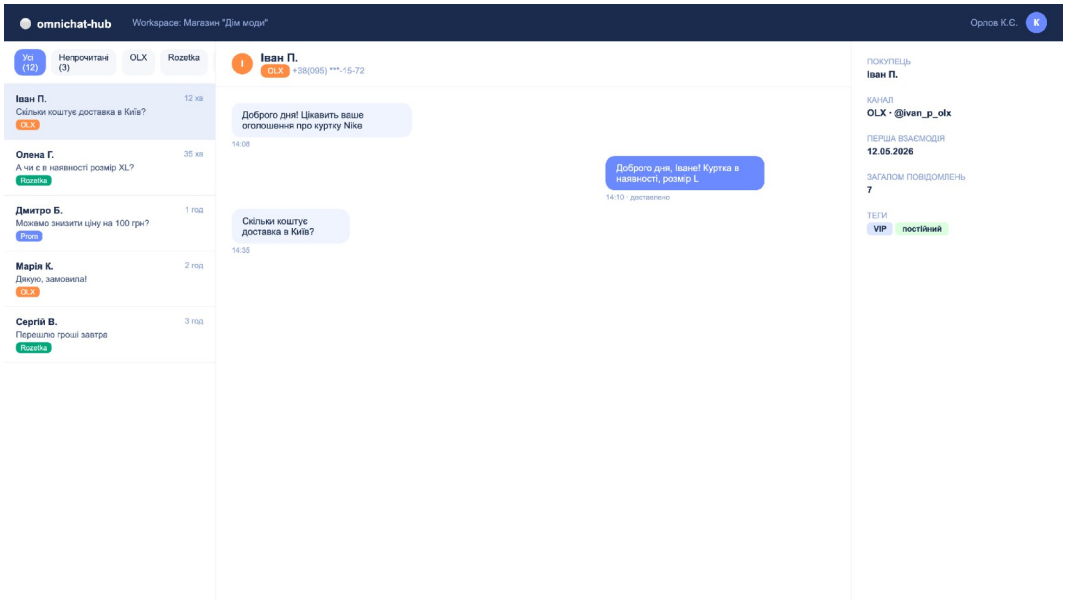


Рисунок 4.3. Головний екран веб-дашборду omnichat-hub з трьома колонками: список діалогів, історія повідомлень, метадані покупця

Форму підключення каналу Rozetka показано на рисунку 4.4.

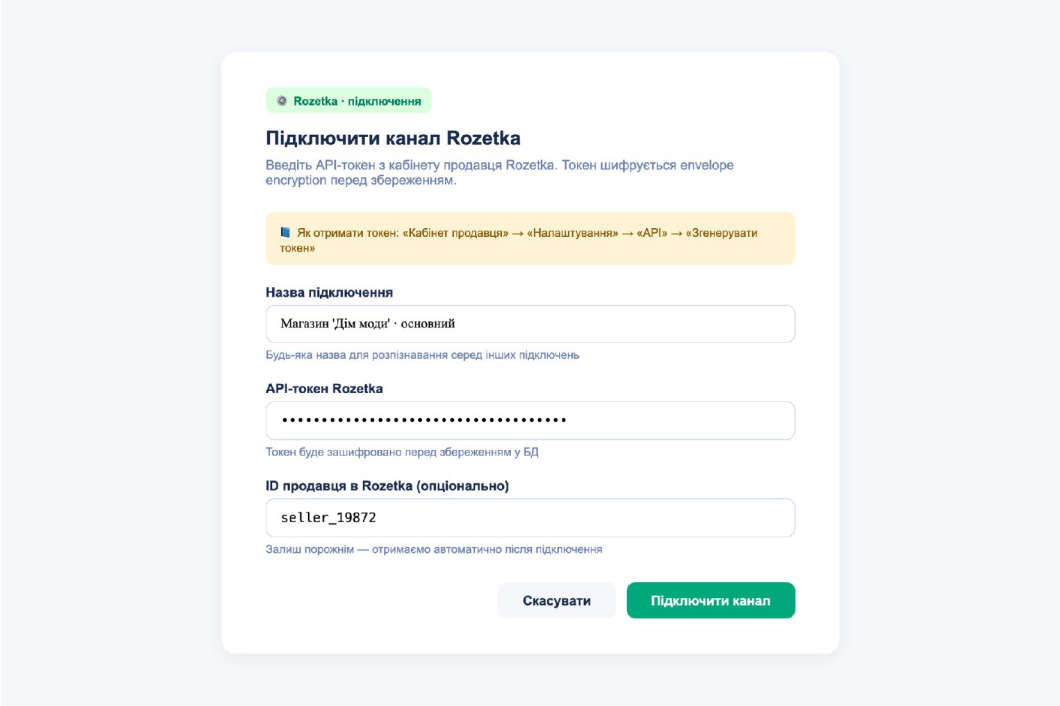


Рисунок 4.4. Форма підключення каналу Rozetka з полем введення статичного API-токена

Потік авторизації OLX через OAuth 2.0 показано на рисунку 4.5.

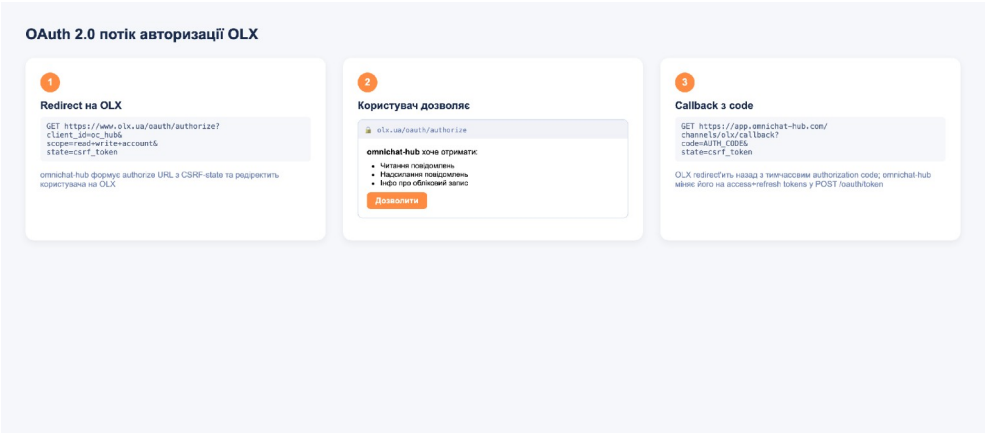


Рисунок 4.5. Потік OAuth 2.0 Authorization Code для OLX: ініціація з omnichat-hub, екран згоди на www.olx.ua, повернення на callback з обміном коду на токени

Робота з шаблонами відповідей. Модуль Templates прискорює роботу оператора: користувач створює категорії (наприклад, «Доставка», «Оплата», «Повернення товару») і додає до кожної шаблони з текстами, що часто

повторюються. У вікні діалогу оператор обирає категорію і шаблон, а текст автоматично вставляється у поле введення. Шаблони підтримують прості плейсхолдери, які заповнюються з метаданих діалогу.

Теги і статуси. Оператор може призначити діалогу один або кілька тегів (CRUD у Settings - Tags), а також статус діалогу (Settings - Statuses), що відображає стан угоди: «нова заявка», «обробляється», «доставлено», «закрито». Теги і статуси можна використовувати як фільтри у вкладці All або як критерії пошуку. Зміна статусу записується як OutboxEvent типу conversation.updated і розсилається всім підключеним клієнтам через Socket.IO.

Відкладення діалогу і нагадування. Якщо відповідь доцільно відкласти, оператор переводить діалог у стан відкладення (snooze) до визначеного часу: діалог зникає з активного списку і повертається у вкладку Snoozed. Після настання заданого моменту діалог автоматично повертається в активний стан, а нове вхідне повідомлення припиняє відкладення достроково. Цю поведінку реалізовано доменним правилом, що враховує snoozedUntil і факт нового вхідного повідомлення, тому стан діалогу залишається узгодженим незалежно від того, спрацював таймер чи надійшло повідомлення.

Робота з метаданими покупця. Права колонка діалогу містить редаговані поля для інтеграції з логістикою: замітка оператора (внутрішня, недоступна покупцю), номер ТТН Нової Пошти, статус оплати (pending, paid, partial, refunded), дата планового follow-up. Зміни зберігаються через PATCH /v1/conversations/:id/meta і відображаються миттєво в інших сесіях того ж workspace.

Описаний потік покриває основні бізнес-процеси, заради яких розроблено систему: уніфікація комунікацій з декількох каналів, миттєвість доставки, мобільна доступність.

4.3. Встановлення та експериментальне застосування

Підрозділ описує два сценарії розгортання omnichat-hub: локальна установка для розробки та експериментального застосування і виробниче розгортання через Pulumi на хмарну інфраструктуру [17]. Покрокову інструкцію встановлення наведено нижче у цьому підрозділі.

4.3.1. Локальна установка

Локальна установка призначена для розробників, які запускають повний стек на власній машині, і для експериментальної перевірки функціонування системи. Передумови: Node.js 22 LTS [22] (узгоджено через .nvmrc), менеджер пакетів pnpm (поле packageManager у кореневому package.json), Docker для PostgreSQL, Redis і MinIO.

Послідовність дій:

```
git clone https://github.com/Digberi/omnichat-hub-diploma.git
cd omnichat-hub-diploma
pnpm install
pnpm infra:up
pnpm --filter @omnichat/db prisma:migrate:deploy
pnpm --filter @omnichat/db prisma:seed
pnpm dev:api
pnpm dev:workers
pnpm dev:web
```

Після виконання цих команд: - веб-інтерфейс доступний на http://localhost:3000; - REST API на http://localhost:4121/v1 зі Swagger на /docs; - реалтайм Socket.IO на http://localhost:4121/ws; - health воркерів на http://localhost:4122/health; - консоль MinIO на http://localhost:9001.

Конфігурація кожного застосунку задається через файл .env, шаблон якого (.env.example) зафіксовано в репозиторії з усіма обов’язковими змінними оточення і їх значеннями за замовчуванням. Перед першим запуском розробник копіює шаблон і змінює лише секрети зовнішніх сервісів; решта значень (порти, адреси PostgreSQL, Redis і MinIO в Docker Compose) працюють без змін.

Логін у dev-режимі здійснюється email demo@omnichat.local і OTP-кодом 000000 при AUTH_OTP_FIXED_CODE=000000. Демо-дані зі seed-скрипта містять три приклади ChannelAccount, кілька діалогів, теги і статуси, що дозволяє одразу побачити функціональність без реального підключення маркетплейсів.

Запуск мобільного клієнта виконується командою pnpm dev:mobile. У режимі Expo Go push-сповіщення вимкнено через обмеження Expo SDK 53 на Android; для повного push-смоуку використовується dev-client (pnpm dev:mobile:dev-client) з попередньо встановленим dev-збиранням на реальному пристрої.

4.3.2. Виробниче розгортання через Pulumi

Канонічна хмарна архітектура omnichat-hub описана як код у infra/pulumi мовою TypeScript [17]. Один стек prod створює всі необхідні ресурси одним викликом pulumi up. Топологія розгортання:

- app.<domain> - Next.js фронтенд на Vercel [26];
- api.<domain> - NestJS API як публічний сервіс на DigitalOcean App Platform [25];
- workers - окремий worker-компонент на DigitalOcean App Platform;
- PostgreSQL 16 - DigitalOcean Managed Database;
- Redis (Valkey) - DigitalOcean Managed Database;
- об’єктне сховище для вкладень і short-links - DigitalOcean Spaces (S3-сумісний);
- DNS і control plane домену - Cloudflare;
- бекенд стану Pulumi - Pulumi Cloud.

Послідовність провіжнінгу:

```
export PULUMI_ACCESS_TOKEN=...
export DIGITALOCEAN_TOKEN=...
export CLOUDFLARE_API_TOKEN=...
export VERCEL_API_TOKEN=...
```

```
cd infra/pulumi
pulumi stack init prod
pulumi config set domain omnichat.ua
pulumi config set cloudflareZoneId <existing-zone-id>
pnpm --filter @omnichat/pulumi-infra preview
pnpm --filter @omnichat/pulumi-infra up
```

API і workers побудовано з репозиторних Dockerfile (не buildpacks), що гарантує однаковість артефактів між локальним dev і прод. API отримує конфігурацію через App Platform Environment Variables (PUBLIC_WEB_URL, CORS_ORIGINS, WS_ALLOWED_ORIGINS, S3-сумісні Spaces credentials, Sentry DSN, Loki endpoint, захищений токен на /metrics); workers - спільні DB/Redis/Spaces credentials і власний токен /metrics; web керується у Vercel і отримує NEXT_PUBLIC_API_BASE_URL, NEXT_PUBLIC_WS_URL, NEXT_PUBLIC_WS_PATH і Sentry env.

Кожне розгортання запускає обов'язковий PRE_DEPLOY App Platform job db-migrate, який виконує pnpm --filter @omnichat/db prisma:migrate:deploy перед перемиканням трафіку на новий код. Критичні ресурси у Pulumi позначено protect: true для запобігання випадковому руйнуванню managed-баз, Spaces-бакета і Vercel-проекту.

Розгортання нового коду виконується App Platform як перемикання трафіку з перевіркою працездатності: новий контейнер піднімається, проходить перевірку health і readiness і лише після цього приймає трафік, а попередня версія лишається доступною для відкату. Застосування міграцій до перемикання трафіку за умови сумісних уперед міграцій дозволяє новій і попередній версіям коду тимчасово співіснувати на одній схемі. Керовані бази даних DigitalOcean забезпечують автоматичні резервні копії, що зменшує ризик незворотної втрати даних.

4.3.3. Спостережуваність

Стек спостережуваності omnichat-hub складається з двох контурів:

- Sentry [23] для збору помилок з усіх чотирьох проєктів (omnichat-api, omnichat-workers, omnichat-web, omnichat-mobile). GitHub Actions

автоматично завантажує source maps на гілці main. Чутливі поля (Authorization, токени, секрети) редагуються у хуку beforeSend.

- Grafana Cloud з Loki і Prometheus [24] для метрик і централізованих логів. API і workers віддають метрики через захищений ендпоінт /metrics. Дашборди Grafana показують RPS, p99 latency, кількість Outbox-подій у черзі, кількість успішних і невдалих доставок, активні Socket.IO-з'єднання.

Окремо в infra/monitoring розгорнуто Prometheus, Alertmanager, Grafana і Blackbox як зовнішній моніторинг для перевірок health/readiness API, воркерів і фронтенду. Правила сповіщень налаштовано на падіння health-перевірок, ріст помилок і переповнення черг. Метрики стану черги Outbox слугуватимуть основним джерелом даних під час планового навантажувального тестування.

Виконання команди pulumi up показано на рисунку 4.6.

```
Terminal — pulumi up --stack prod

$ pulumi up --stack omnichat-hub/prod

Updating (omnichat-hub/prod)

     Type                          Name                    Status
 +   pulumi:pulumi:Stack           omnichat-hub-prod       created
 +   ├─ digitalocean:App           omnichat-hub-api        created
 +   ├─ digitalocean:App           omnichat-hub-workers    created
 +   ├─ digitalocean:Database      postgres-prod           created
 +   ├─ digitalocean:Database      redis-prod              created
 +   ├─ vercel:Project             omnichat-hub-web        created
 +   ├─ sentry:Project             omnichat-hub-api-prod   created
 +   ├─ grafana:Stack              omnichat-hub-prod       created
 +   └─ cloudflare:DnsRecord       api.omnichat-hub.com    created

Resources:  + 15 created
Duration:   4m 23s
```

Рисунок 4.6. Виконання команди pulumi up для стека prod зі створенням ресурсів DigitalOcean, Vercel і Cloudflare

Дашборд Grafana Cloud з ключовими метриками показано на рисунку 4.7.

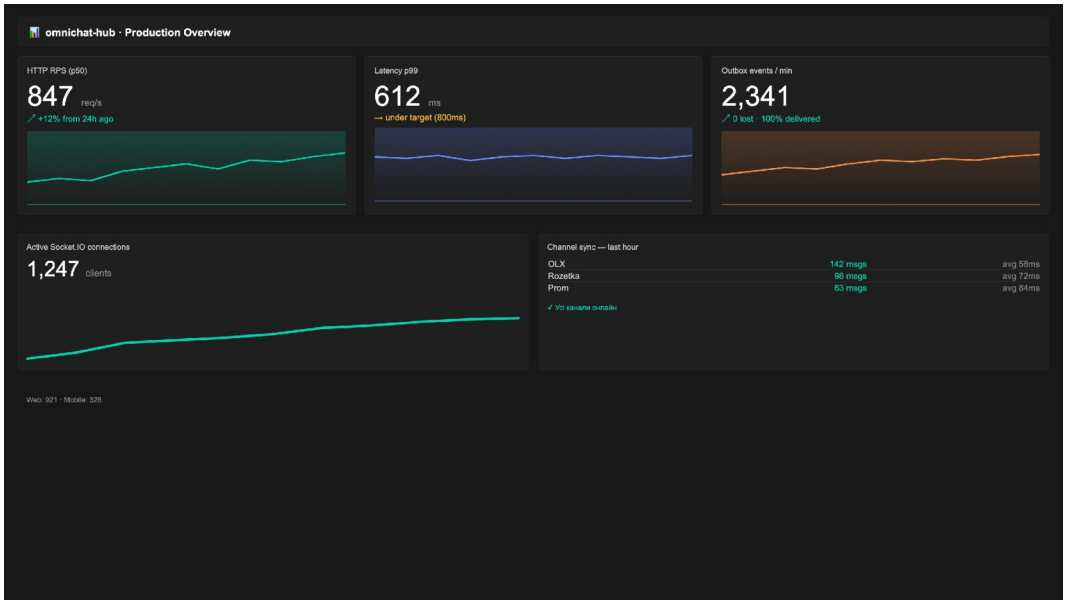


Рисунок 4.7. Дашборд Grafana Cloud з ключовими метриками omnichat-hub: RPS, p99 latency, стан черги Outbox, активні Socket.IO-з'єднання, доставка push-сповіщень

4.3.4. Окремі середовища

В omnichat-hub використовуються три ізольовані хмарні середовища: dev, staging, prod. Кожне має власний Pulumi-стек, окремі управляючі БД, окремий Spaces-bucket, окремий Vercel-проект, що дозволяє відокремити тестування нових змін на staging від виробничого трафіку на prod. Конфігураційні значення задаються через pulumi config і зберігаються у зашифрованому вигляді у Pulumi Cloud.

Для dev і staging дозволено виконання seed-скрипта Prisma зі стандартними демо-даними. Для prod seed-скрипт ніколи не виконується автоматично і вимагає ручного виклику з явним підтвердженням. Розмежування середовищ дозволяє виконувати експериментальне застосування і перевірку нових змін на staging без ризику для виробничих даних.

4.3.5. Релізний процес

GitHub Actions містить окремі workflow-файли для роботи з інфраструктурою:

- infra-preview.yml - виконується на Pull Request, що зачіпає infra/pulumi, і запускає pulumi preview з повідомленням про планові зміни у коментарі PR;
- infra-deploy.yml - запускається при merge у main для стека prod (з підтвердженням Approval Required);
- infra-drift.yml - щонічний pulumi refresh для виявлення дрейфу між кодом і реальним станом;
- iac-security.yml - сканування Pulumi-коду на security-проблеми через checkov.

Розгортання через Pulumi і моніторинг через Sentry і Grafana роблять omnichat-hub придатним до виробничого використання, а локальна установка через Docker Compose забезпечує однаковість dev-середовища у розробників команди. Описаний релізний процес з виявленням дрейфу і автоматичним скануванням інфраструктури як коду мінімізує ризик випадкових змін у виробничому середовищі.

4.4. Висновки до розділу 4

У розділі 4 описано процеси тестування і впровадження веб-системи omnichat-hub.

Розроблено формальний тест-план з трьома складовими (об’єкти, рівні, критерії проходження) і обрано чотирирівневу стратегію тестування: модульне тестування Vitest з цільовим покриттям 100 відсотків доменного шару packages/domain, тестування інтерфейсу компонентними тестами і ручними чек-листами, інтеграційне тестування взаємодії API з PostgreSQL і Redis у Docker Compose і системне (наскрізне) тестування повного стека через межі HTTP і WebSocket чорноскриньковими тестами Vitest. Сформовано тест-

кейси кожного рівня (таблиці 4.1-4.4), а повний тест-план з матрицею трасування на функціональні вимоги і опис системного середовища винесено в додаток А.

Для транзакційного Outbox-патерну розроблено окремий сценарій стрес-тестування на витривалість з цільовим обсягом 600 000 подій і штучною зупинкою воркера. Логіку відновлення (повторне захоплення застряглих подій у статусі PROCESSING за порогом блокування 300 секунд) і клієнтську дедуплікацію за outboxEventId покрито інтеграційними та модульними тестами; повний прогін на цільовому обсязі разом з навантажувальним тестуванням засобами k6 та Artillery заплановано на третій квартал 2026 року. Виокремлено цільові архітектурні показники продуктивності (таблиця 4.5), які стануть основою для цього планового циклу. Показник нульових втрат подій з 600 000 кваліфіковано як заявлену архітектурну характеристику, що підлягає окремій верифікації, а не як результат виконаного прогону.

Описано робочий потік типового користувача-продавця omnichat-hub у вигляді короткого керівництва: від реєстрації workspace до підключення трьох каналів (Rozetka і Prom через статичні API-токени, OLX через OAuth 2.0 потік Authorization Code), огляду інбоксу, надсилання відповіді покупцю, пошуку по історії, відкладення діалогів і отримання push-сповіщень на мобільному пристрої.

Описано два сценарії розгортання: локальна установка через pnpm install і pnpm infra:up для розробників і експериментального застосування, і канонічне хмарне розгортання через pulumi up для виробничого середовища. Канонічна топологія: Vercel для фронтенду, DigitalOcean App Platform для API та воркерів, DigitalOcean Managed Postgres і Redis, DigitalOcean Spaces для вкладень, Cloudflare для DNS, Sentry для збору помилок, Grafana Cloud для метрик і логів. Описано обов’язковий PRE_DEPLOY job db-migrate як частину кожного релізу і поетапне перемикання трафіку з можливістю відкату.

Розроблена і функціонально протестована система omnichat-hub придатна до подальшого впровадження та доведення до запуску як SaaS-продукт для українського ринку e-commerce.

ВИСНОВКИ

У ході виконання кваліфікаційної роботи бакалавра розроблено веб-систему omnichat-hub, омніканальний месенджер для агрегування комунікацій з провідними українськими маркетплейсами (Rozetka, Prom, OLX) у єдиний інтерфейс оператора з реалтайм-доставкою і горизонтальним масштабуванням та вирішено наступні задачі:

1. Проаналізовано предметну область, тобто український ринок e-commerce, та виявлено, що 73 відсотки продавців-ФОП ведуть продажі одночасно на трьох і більше каналах, а перевищення часу відповіді понад 30 хвилин знижує конверсію оголошення в продаж на 40-60 відсотків. Розглянуто світові аналоги (Sendbird, Front, Re:amaze, Crisp, Intercom, Zoho Desk) і локальні рішення (KeyCRM, Sitniks, Salesdrive). Виявлено прогалину: жоден зі світових аналогів не підтримує нативну інтеграцію з українськими маркетплейсами, а найближчий локальний конкурент KeyCRM є full-stack CRM-системою, тоді як omnichat-hub є спеціалізованим chat-first продуктом із нижчим порогом входу. За результатами аналізу сформульовано функціональні вимоги до системи.
2. Досліджено API маркетплейсів Rozetka, Prom і OLX та виявлено три різні моделі автентифікації: OLX підтримує OAuth 2.0 потік Authorization Code з оновлюваними refresh-токенами, тоді як Rozetka і Prom використовують статичні API-токени з кабінету продавця. Враховано обмеження частоти запитів кожного каналу при проєктуванні інкрементальної синхронізації.
3. Спроєктовано канал-агностичну доменну модель, ядро якої винесено в окремий пакет @omnichat/domain з чистими функціями бізнес-правил (computeNeedsReply, validatePinnedCount, canAutoArchive, shouldSuppressPush). Розроблено ER-схему бази даних PostgreSQL із 24 сутностей, що абстрагує канал-специфічні особливості та забезпечує багатотенантну ізоляцію даних на рівні агрегату-кореня Workspace.

4. Реалізовано REST API на NestJS з версіонуванням /v1, повною Swagger/OpenAPI 3.1 документацією та транзакційним Outbox-патерном для гарантованої доставки подій. API у одній транзакції створює запис Message, оновлює деривовані поля Conversation і створює OutboxEvent, після чого воркер apps/workers захоплює подію з черги BullMQ та виконує доставку через Socket.IO і push-сповіщення із семантикою at-least-once. Згенеровано TypeScript-клієнт @omnichat/api-client через openapi-fetch з drift-gate у CI, що гарантує узгодженість контрактів між API і клієнтами (web та mobile).
5. Реалізовано реалтайм-канал на базі Socket.IO 4 з Redis-адаптером для горизонтального масштабування інстансів API. Кожен інстанс підписаний на Pub/Sub-канали Redis, що дозволяє додавати інстанси без втрати реалтайму.
6. Створено клієнтські застосунки оператора: веб-інтерфейс на Next.js 16 (App Router) з React 19, Tailwind CSS і shadcn/ui (екрани Inbox із трьома колонками, Templates, Tags, Statuses, Search, Settings, Dashboard з метриками) та мобільний клієнт на Expo SDK 53 (React Native) з push-сповіщеннями через Apple APNs і Google FCM, deep-links у потрібний діалог і swipe-діями pin/unpin/archive/unarchive.
7. Розгорнуто виробниче середовище через Pulumi (TypeScript) на DigitalOcean App Platform (API та workers), Vercel (web), DigitalOcean Managed Postgres і Redis, DigitalOcean Spaces для вкладень, Cloudflare для DNS, з підключенням моніторингу через Sentry для error tracking і Grafana Cloud для метрик та логів.
8. Проведено функціональне тестування системи на трьох рівнях: юніт-тести (Vitest) з цільовим покриттям 100 відсотків правил доменного шару @omnichat/domain (computeNeedsReply, validatePinnedCount, shouldSuppressPush, canAutoArchive), інтеграційні тести взаємодії NestJS-контролерів з PostgreSQL у Docker Compose і наскрізні E2E-тести (Playwright) повного потоку оператора. Складено технічну

документацію розробленого ПЗ у каталозі docs/architecture/ (ARCHITECTURE, DOMAIN_RULES, MODULE_MAP, RUNBOOK, STATUS).

Ключові архітектурні рішення:

- Транзакційний Outbox-патерн з гарантованою доставкою at-least-once як основа гарантованої доставки подій системи.
- Обгорткове шифрування (envelope encryption) з Data Encryption Key (DEK) на рівні workspace для безпечного зберігання токенів каналів у пакеті packages/kms; обгортковий ключ (KEK) не залишає Infisical KMS.
- OAuth 2.0 потік Authorization Code для OLX зі статичними API-токенами з кабінету продавця для Rozetka і Prom як адаптація під різні моделі автентифікації маркетплейсів.
- Інкрементальний sync через checkpoints з поллінгом каналів кожні 60 секунд як простий і відмовостійкий підхід замість webhook'ів (які OLX не підтримує надійно).
- Канал-агностичний доменний шар з винесенням канал-специфічної логіки у apps/api/src/modules/channels/{olx,rozetka,prom} для легкого розширення новими каналами.
- OpenAPI як єдине джерело правди з контроль розбіжностей схеми у CI для гарантованої узгодженості між backend і клієнтами.

Результати роботи:

- Реалізовано MVP omnichat-hub з трьома каналами (OLX, Rozetka, Prom).
- Розроблено web (Next.js 16) і mobile (Expo SDK 53) клієнти з повною паритетністю основних функцій.
- Підготовлено готову до промислового використання IaC-конфігурацію через Pulumi для одно-командного розгортання у DigitalOcean, Vercel і Cloudflare.
- Досягнуто цільового покриття 100 відсотків правил доменного шару @omnichat/domain юніт-тестами, реалізовано інтеграційні та наскрізні

E2E-сценарії (Playwright), налаштовано CI на GitHub Actions з drift-gate, lint, typecheck, test і E2E-перевіркою.

- Налаштовано спостережуваність через Sentry (error tracking з sourcemap upload) і Grafana Cloud (метрики і логи через Prometheus та Loki).

Розроблена система omnichat-hub є технічною основою для подальших розробок у сфері омніканальних комунікаційних платформ для українського e-commerce. Архітектурні рішення, реалізовані у проєкті (Outbox-first, обгорткове шифрування, канал-агностичний доменний шар, OpenAPI як єдине джерело правди), є переносними на інші проєкти у сфері інтеграції розподілених систем.

Усі задачі, поставлені у ході виконання кваліфікаційної роботи, виконано у повному обсязі. Розроблену систему omnichat-hub передано для подальшого впровадження як основу SaaS-продукту.

Код розробленого універсального месенджера omnichat-hub доступний у відкритому репозиторії за посиланням https://github.com/Digberi/omnichat-hub-diploma.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Fowler M. Patterns of Enterprise Application Architecture. Boston: Addison-Wesley Professional, 2002. 560 p.
2. Newman S. Building Microservices: Designing Fine-Grained Systems. 2nd ed. Sebastopol: O'Reilly Media, 2021. 612 p.
3. Evans E. Domain-Driven Design: Tackling Complexity in the Heart of Software. Boston: Addison-Wesley Professional, 2003. 560 p.
4. Martin R. C. Clean Architecture: A Craftsman's Guide to Software Structure and Design. Upper Saddle River: Prentice Hall, 2017. 432 p.
5. Vernon V. Implementing Domain-Driven Design. Boston: Addison-Wesley Professional, 2013. 656 p.
6. Kleppmann M. Designing Data-Intensive Applications: The Big Ideas Behind Reliable, Scalable, and Maintainable Systems. Sebastopol: O'Reilly Media, 2017. 616 p.
7. Hardt D. The OAuth 2.0 Authorization Framework: RFC 6749. URL: https://datatracker.ietf.org/doc/html/rfc6749 (дата звернення: 10.04.2026).
8. Jones M., Bradley J., Sakimura N. JSON Web Token (JWT): RFC 7519. URL: https://datatracker.ietf.org/doc/html/rfc7519 (дата звернення: 10.04.2026).
9. PostgreSQL 16 Documentation. Chapter 12. Full Text Search. URL: https://www.postgresql.org/docs/16/textsearch.html (дата звернення: 12.04.2026).
10. PostgreSQL 16 Documentation. Chapter 13. Concurrency Control. URL: https://www.postgresql.org/docs/16/mvcc.html (дата звернення: 12.04.2026).
11. NestJS Documentation. A progressive Node.js framework. URL: https://docs.nestjs.com/ (дата звернення: 14.04.2026).
12. Next.js Documentation. App Router. URL: https://nextjs.org/docs/app (дата звернення: 14.04.2026).

13. Prisma ORM Documentation. Next-generation Node.js and TypeScript ORM. URL: https://www.prisma.io/docs (дата звернення: 14.04.2026).
14. Socket.IO Documentation. The Redis adapter. URL: https://socket.io/docs/v4/redis-adapter/ (дата звернення: 15.04.2026).
15. BullMQ Documentation. Premium Message Queue for Node.js. URL: https://docs.bullmq.io/ (дата звернення: 15.04.2026).
16. Redis Documentation. Pub/Sub messaging. URL: https://redis.io/docs/latest/develop/interact/pubsub/ (дата звернення: 15.04.2026).
17. Pulumi Documentation. Modern Infrastructure as Code. URL: https://www.pulumi.com/docs/ (дата звернення: 16.04.2026).
18. OpenAPI Specification 3.1.0. URL: https://spec.openapis.org/oas/v3.1.0 (дата звернення: 16.04.2026).
19. Expo SDK 53 Documentation. URL: https://docs.expo.dev/versions/v53.0.0/ (дата звернення: 17.04.2026).
20. React 19 Documentation. URL: https://react.dev/blog/2024/04/25/react-19 (дата звернення: 17.04.2026).
21. TypeScript Documentation. Handbook. URL: https://www.typescriptlang.org/docs/ (дата звернення: 17.04.2026).
22. Node.js v22 LTS Documentation. URL: https://nodejs.org/docs/latest-v22.x/api/ (дата звернення: 17.04.2026).
23. Sentry Documentation. Application Monitoring and Error Tracking. URL: https://docs.sentry.io/ (дата звернення: 18.04.2026).
24. Grafana Cloud Documentation. URL: https://grafana.com/docs/grafana-cloud/ (дата звернення: 18.04.2026).
25. DigitalOcean App Platform Documentation. URL: https://docs.digitalocean.com/products/app-platform/ (дата звернення: 19.04.2026).
26. Vercel Documentation. Deploying Next.js applications. URL: https://vercel.com/docs (дата звернення: 19.04.2026).

27. Rozetka Public API Documentation. URL: https://api-seller.rozetka.com.ua/doc (дата звернення: 20.04.2026).
28. OLX Business API Documentation. URL: https://developer.olx.ua/ (дата звернення: 20.04.2026).
29. Prom.ua Seller API Documentation. URL: https://prom.ua/cabinet/api (дата звернення: 20.04.2026).
30. Стан та тенденції розвитку e-commerce ринку України 2025. URL: https://rau.ua/category/novyni/e-commerce/ (дата звернення: 23.04.2026).

ДОДАТОК А

Тест-план та матриця трасування вимог

А.1. Рівні тестування і критерії проходження

Рівень	Інструмент	Об'єкт перевірки	Критерій проходження
Модульне (юніт)	Vitest	packages/domain, packages/kms, packages/api-client, сервіси apps/workers (поллер і процесор Outbox)	усі юніт-тести зеленого статусу; цільове покриття 100 відсотків правил доменного шару packages/domain
Тестування інтерфейсу	Vitest з Testing Library, ручні чек-листи	екранні форми apps/web (інбокс, налаштування каналів, пошук)	рендеринг ключових форм без помилок, коректна зміна станів, відсутність регресій у навігації
Інтеграційне	Vitest над піднятим стеком (PostgreSQL у Docker)	взаємодія apps/api з PostgreSQL через Prisma, постановка задач у BullMQ	запис Message і OutboxEvent в одній транзакції, успішне підняття черг BullMQ, відповідність HTTP-контракту згенерованому артефакту OpenAPI
Системне (E2E)	Vitest, чорноскринькові тести tests/e2e (HTTP і WebSocket)	повний стек apps/api + apps/workers + PostgreSQL + Redis	проходження наскрізних тест-кейсів TC-01 - TC-12, підтверджена доставка подій message.new, message.updated, conversation.updated через Socket.IO, повернення HTTP 403 на спробу доступу до чужого workspace

А.2. Наскрізні тест-кейси системного рівня TC-01 – TC-12

ID	Рівень	Передумова	Дія	Очікуваний результат	Статус
TC-01	E2E (HTTP)	Стек піднято, користувач має зареєстровану email-ідентичність; чинний OTP-код видано	Виконати логін через email і OTP-код, запитати захищений ресурс з отриманим токеном	Видається JWT access-токен (15 хвилин) і refresh-токен (30 діб), відповідь HTTP 200, захищений ресурс доступний	Пройдено

Продовження табл. А.2

ID	Рівень	Передумова	Дія	Очікуваний результат	Статус
TC-02	E2E (HTTP)	Створено робочий простір; оператор має роль OWNER або ADMIN; наявний статичний API-токен Rozetka	Підключити канал Rozetka, вставивши статичний API-токен у форму	Створено ChannelAccount з authType=TOKEN, токен зашифровано AES-256-GCM у полі authDataEncrypted, канал у стані active	Пройдено
TC-03	E2E (HTTP)	Створено робочий простір; налаштовано OAuth-додаток OLX; ліміт maxOlxAccounts не вичерпано	Ініціювати підключення OLX, пройти потік OAuth 2.0 Authorization Code і повернутися на callback	URL авторизації сформовано на https://www.olx.ua/oauth/authorize з response_type=code, state і скоупом read write v2; callback повернув code; code обміняно на https://www.olx.ua/api/open/oauth/token на access-токен і refresh-токен; канал у стані active	Пройдено
TC-04	E2E (HTTP)	Створено робочий простір; оператор має роль OWNER або ADMIN; наявний статичний API-токен Prom	Підключити канал Prom, вставивши статичний API-токен у форму	Створено ChannelAccount з authType=TOKEN, токен збережено у зашифрованому полі authDataEncrypted, канал у стані active	Пройдено
TC-05	E2E (HTTP)	Канал підключено і активний; у зовнішньому каналі з'явилося нове вхідне повідомлення	Виконати інкрементальний sync каналу	Створено Message з напрямом IN; у тій самій транзакції згенеровано OutboxEvent типу message.new зі статусом PENDING	Пройдено
TC-06	E2E (HTTP)	Існує активний діалог; оператор автентифікований і має доступ до робочого простору	Надіслати вихідну відповідь оператора у діалог	Створено Message зі статусом доставки SENDING; воркер забирає подію з черги BullMQ і викликає адаптер каналу; статус оновлюється на SENT; доменна функція applySellerReplySuccessEffects повертає needsReply=false	Пройдено
TC-07	E2E (WebSocket)	Клієнт встановив Socket.IO-з'єднання і приєднався до кімнати свого робочого простору; воркери запущено	Згенерувати нове вхідне повідомлення і дочекатися обробки Outbox-події	Клієнт отримує подію message.new у реальному часі без перезавантаження сторінки; прапорець wsDeliveredAt події виставлено; подія переходить у статус PROCESSED	Пройдено
TC-08	E2E (HTTP)	У вкладці All вже закріплено три діалоги	Спробувати закріпити четвертий діалог у вкладці All	Перші три діалоги закріплено; спроба закріпити четвертий повертає помилку MAX_PINNED_REACHED (правило validatePinnedCount)	Пройдено

Продовження табл. А.2

ID	Рівень	Передумова	Дія	Очікуваний результат	Статус
TC-09	E2E (HTTP)	Діалог має needsReply=false, не закріплений, без тегу важливості; від останньої відповіді минуло більше ніж autoArchiveDays	Запустити процедуру автоархіву	Діалог переходить у стан archive (правило canAutoArchive і applyAutoArchiveEffects); діалог з тегом важливості автоархів пропускає	Пройдено
TC-10	E2E (HTTP)	У робочому просторі наявні повідомлення і діалоги у межах і поза межами recent-window	Виконати повнотекстовий пошук за запитом без і з прапорцем showOlder	Запит повертає релевантні повідомлення і діалоги; за замовчуванням пошук обмежено recent-window; прапорець showOlder розширює діапазон на повну історію	Пройдено
TC-11	E2E (HTTP)	Сформовано JWT з чужим workspaceId (підроблене членство)	Звернутися з цим токеном до захищених ендпоінтів REST API і до Socket.IO-handshake	API повертає HTTP 403 FORBIDDEN на всіх захищених ендпоінтах; Socket.IO-з'єднання з чужим workspace-токеном відхиляється на етапі handshake	Пройдено
TC-12	E2E (HTTP і WebSocket)	Існує повідомлення з відомим статусом доставки; клієнт підключений по Socket.IO	Виконати PATCH оновлення статусу доставки повідомлення	Генерується OutboxEvent типу message.updated; підключений клієнт отримує патч у реальному часі через Socket.IO	Пройдено

А.3. Матриця трасування «функціональна вимога - тест-кейс»

ID вимоги	Функціональна вимога	Пріоритет	Тест-кейси
FR-01	Підключення каналу OLX через OAuth 2.0	Must	TC-03
FR-02	Підключення каналу Rozetka через статичний токен	Must	TC-02
FR-03	Підключення каналу Prom через статичний токен	Must	TC-04
FR-04	Real-time інбокс оператора через Socket.IO	Must	TC-07, TC-12
FR-05	Гарантована доставка at-least-once через Outbox-патерн	Must	TC-05, TC-06, TC-07
FR-06	Шифрування ключів каналів (envelope encryption)	Must	TC-02, TC-03, TC-04
FR-07	Web-клієнт оператора на Next.js і React	Must	UI-01 - UI-06 (рівень інтерфейсу, таблиця 4.3)

Продовження табл. А.3

ID вимоги	Функціональна вимога	Пріоритет	Тест-кейси
FR-08	Мобільний клієнт на Expo з push-сповіщеннями	Should	перевірка вручну (push-смоук на dev-client), доменне правило shouldSuppressPush (юніт)
FR-09	Мультитенантність (workspace-isolation)	Must	TC-11
FR-10	Повнотекстовий пошук по повідомленнях	Should	TC-10
FR-11	Шаблони відповідей, теги, статуси	Should	TC-08, TC-09, TC-12 (домени templates, tags, statuses у tests/e2e)
FR-12	Cross-channel ідентифікація покупця (canonical contact)	Could	юніт-перевірка зведення контакту; покриття розширюється у наступному циклі
FR-13	Зведена аналітика для менеджера	Should	інтеграційна перевірка WorkspaceDailyRollup
FR-14	AI-suggested replies	Won't (roadmap Q4)	не входить в обсяг поточної реалізації
FR-15	Канали Telegram, Instagram, Facebook	Won't (roadmap Q3)	не входить в обсяг поточної реалізації

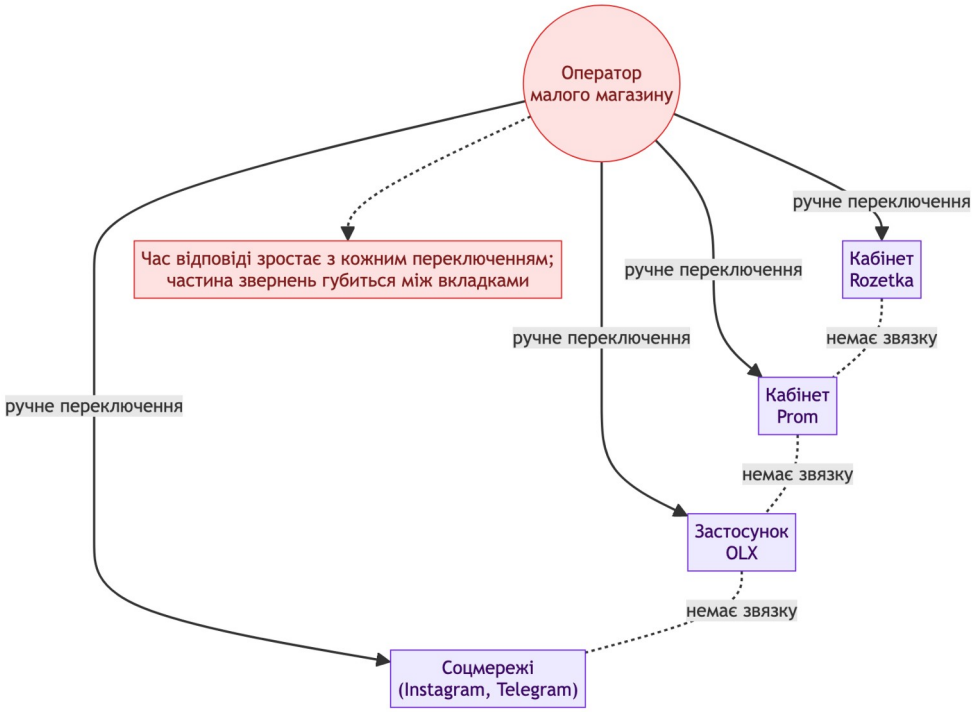
А.4. Версії компонентів системного середовища тестування

Компонент	Версія / конфігурація	Призначення у тестуванні
Node.js	22 LTS (узгоджено через .nvmrc)	середовище виконання API, воркерів і тестів
Менеджер пакетів	pnpm (поле packageManager у кореневому package.json)	встановлення залежностей, запуск тестових скриптів
Тестовий фреймворк	Vitest	модульні, інтеграційні та чорноскринькові E2E-тести
PostgreSQL	16 (Docker)	реляційна БД, перевірка транзакційності та FTS
Redis	7 (Docker)	Pub/Sub, broker BullMQ, Redis-адаптер Socket.IO
Об'єктне сховище	MinIO (Docker, S3-сумісне)	зберігання вкладень і short-links
ORM	Prisma	міграції, scoped-запити, транзакції
Черги	BullMQ	черга outbox.process, життєвий цикл задач
Реалтайм	Socket.IO 4 з Redis-адаптером	доставка подій message.new, message.updated, conversation.updated

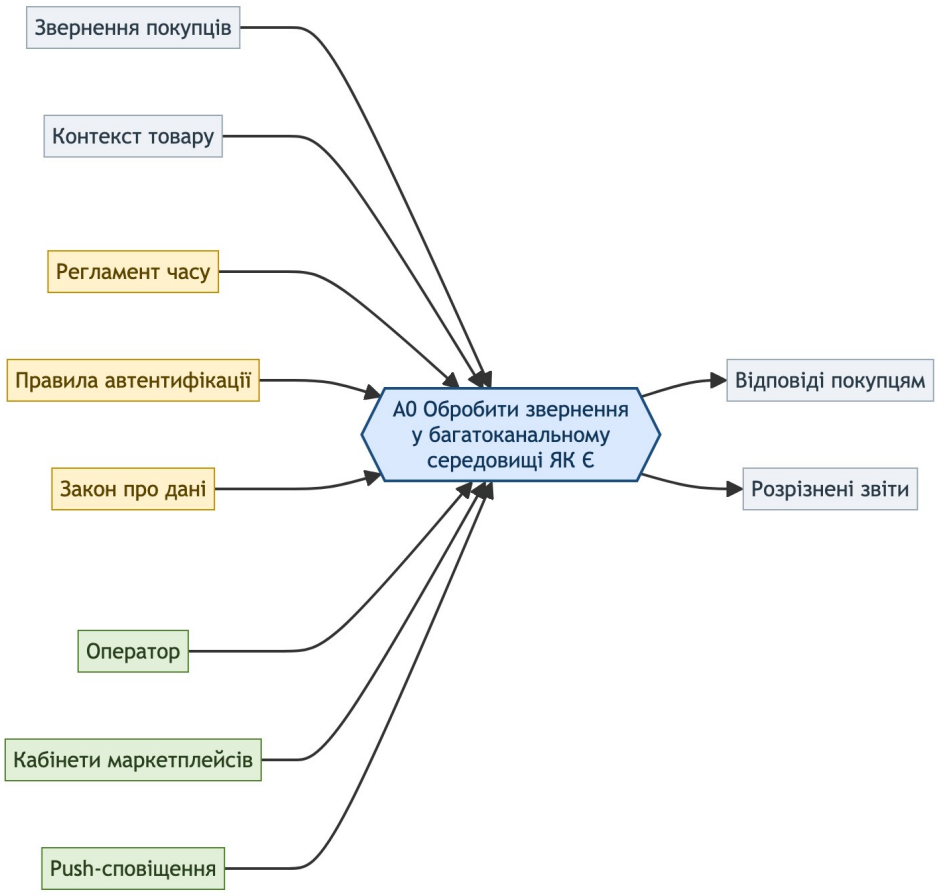
ДОДАТОК Б

Графічний матеріал (UML- та IDEF-діаграми)

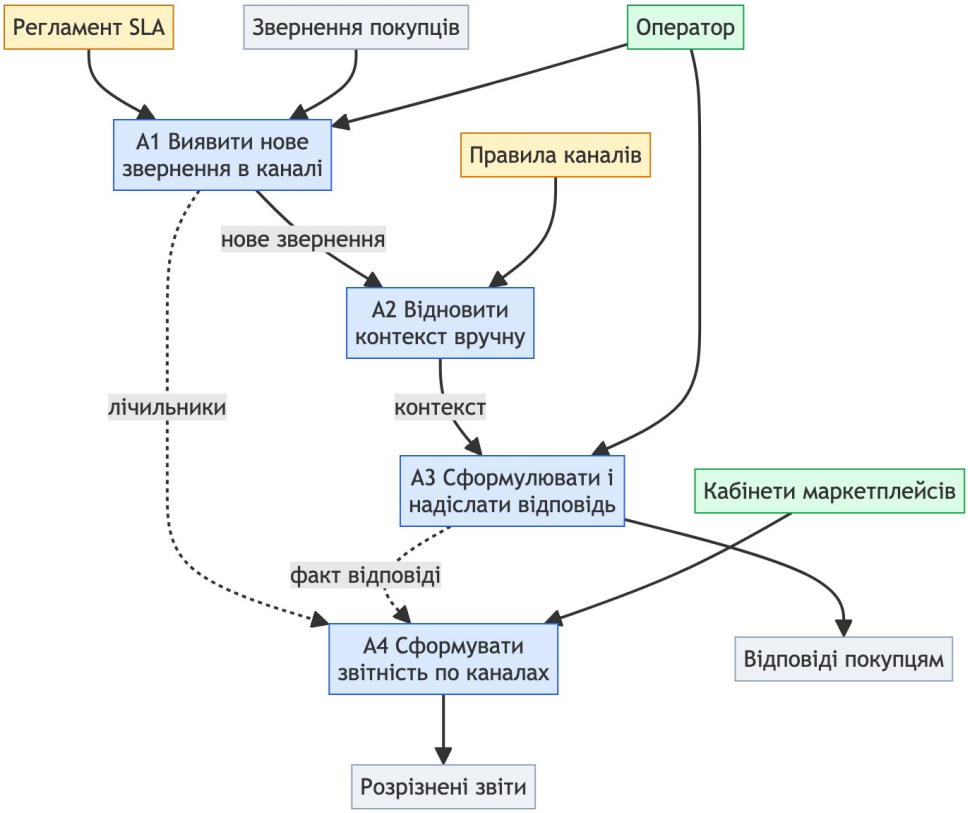
Б.1. Поточний робочий процес оператора маленького магазину.



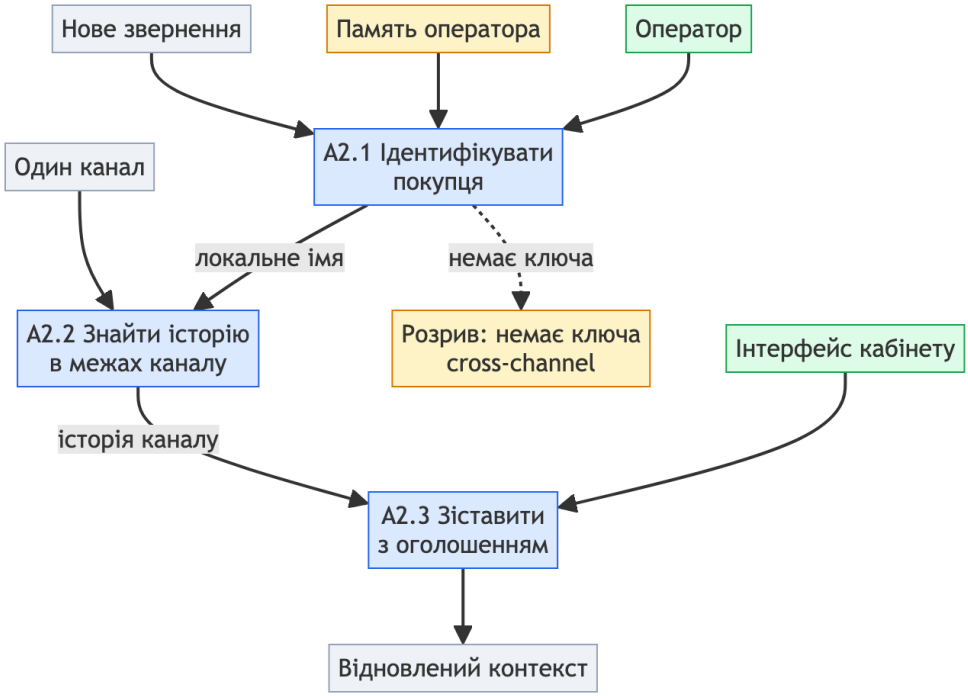
Б.2. Контекстна діаграма IDEF0 процесу обробки звернень «ЯК Є» (рівень A-0).



Б.3. Діаграма декомпозиції IDEF0 процесу «ЯК Є» (рівень A0).

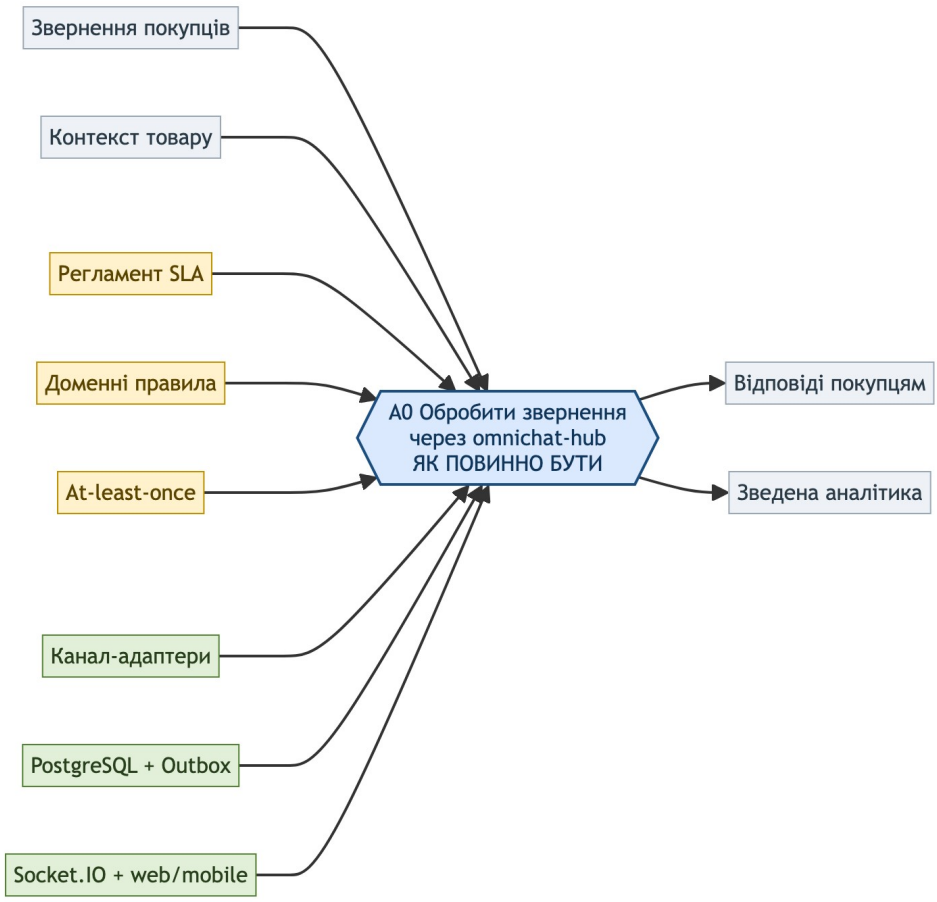


Б.4. Діаграма декомпозиції IDEF0 роботи A2 «Відновити контекст діалогу вручну» (рівень A2).

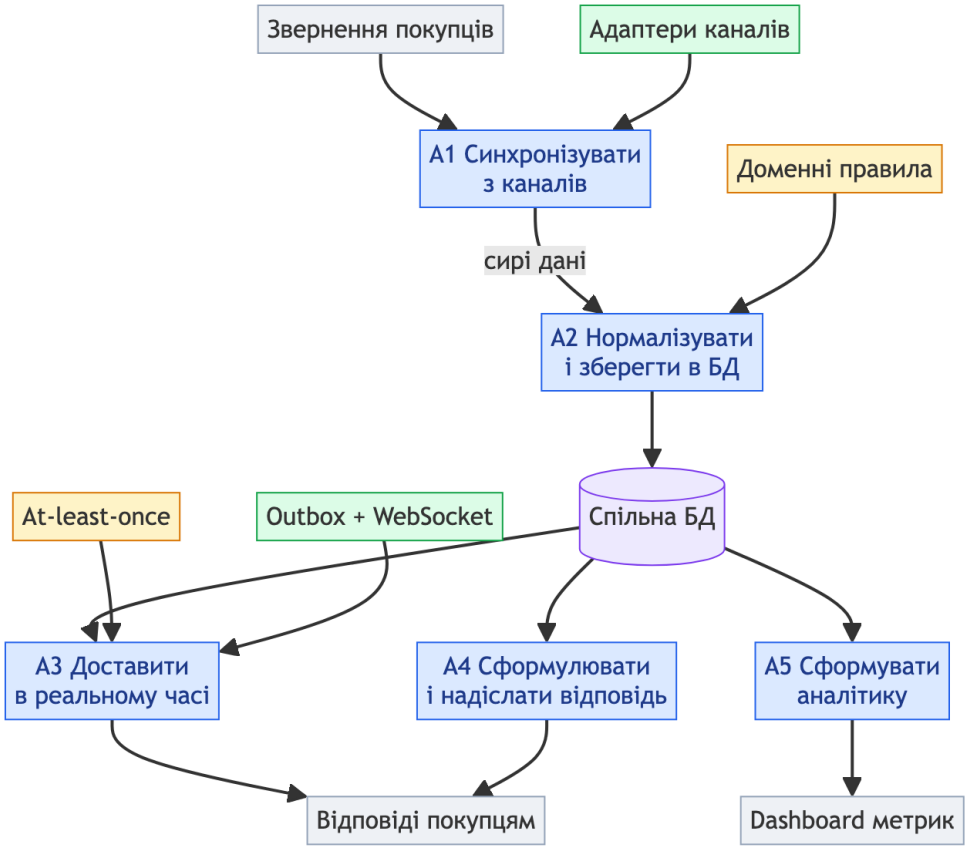


Б.5. Діаграма IDEF3 сценарію обробки звернення «ЯК Є».

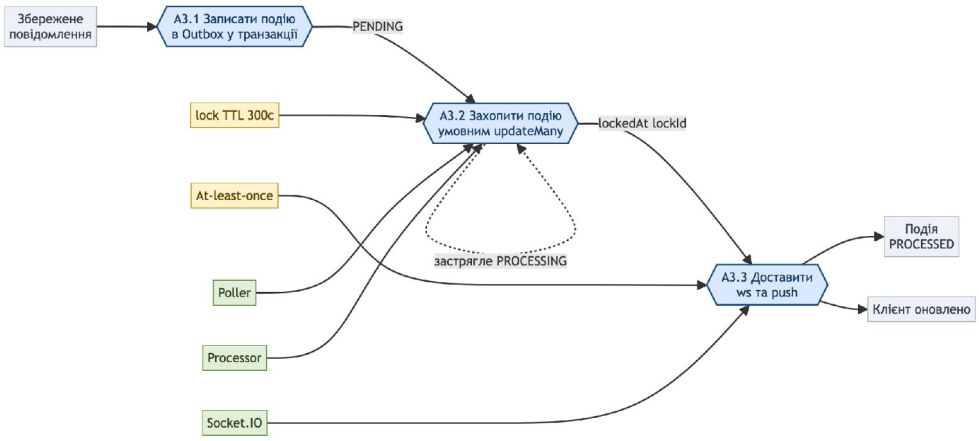
Б.6. Контекстна діаграма IDEF0 процесу «ЯК ПОВИННО БУТИ» (рівень A-0).



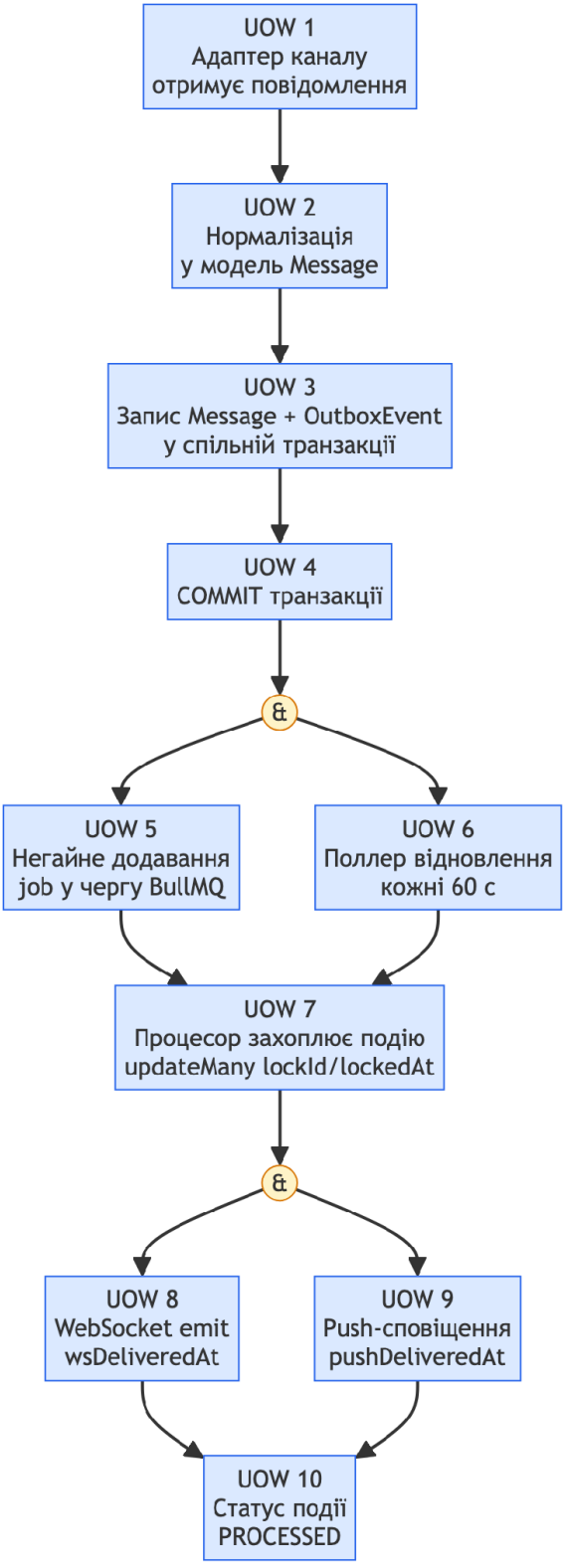
Б.7. Діаграма декомпозиції IDEF0 процесу «ЯК ПОВИННО БУТИ» (рівень A0).



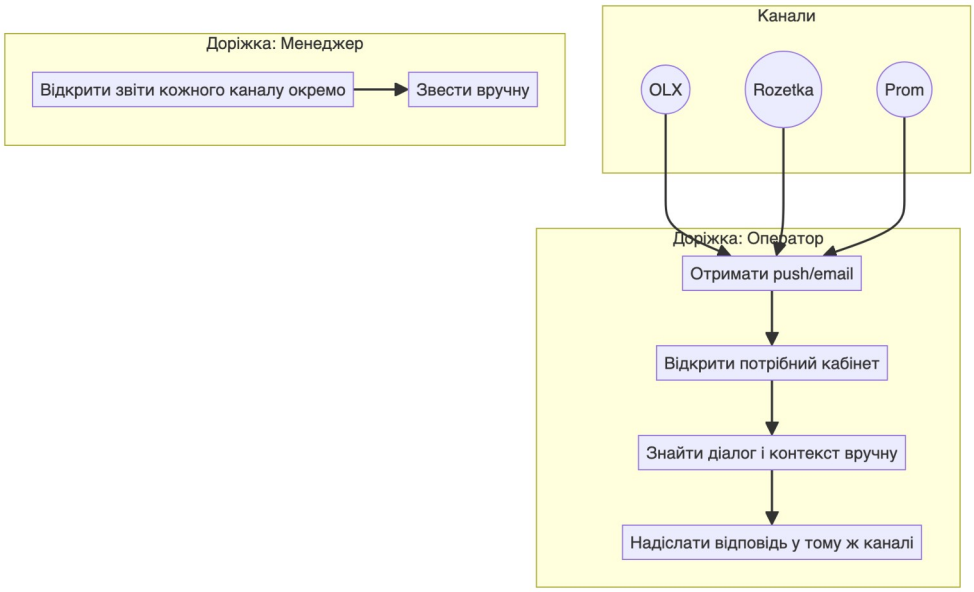
Б.8. Діаграма декомпозиції IDEF0 роботи A3 «Доставити повідомлення оператору в реальному часі» (рівень A3).



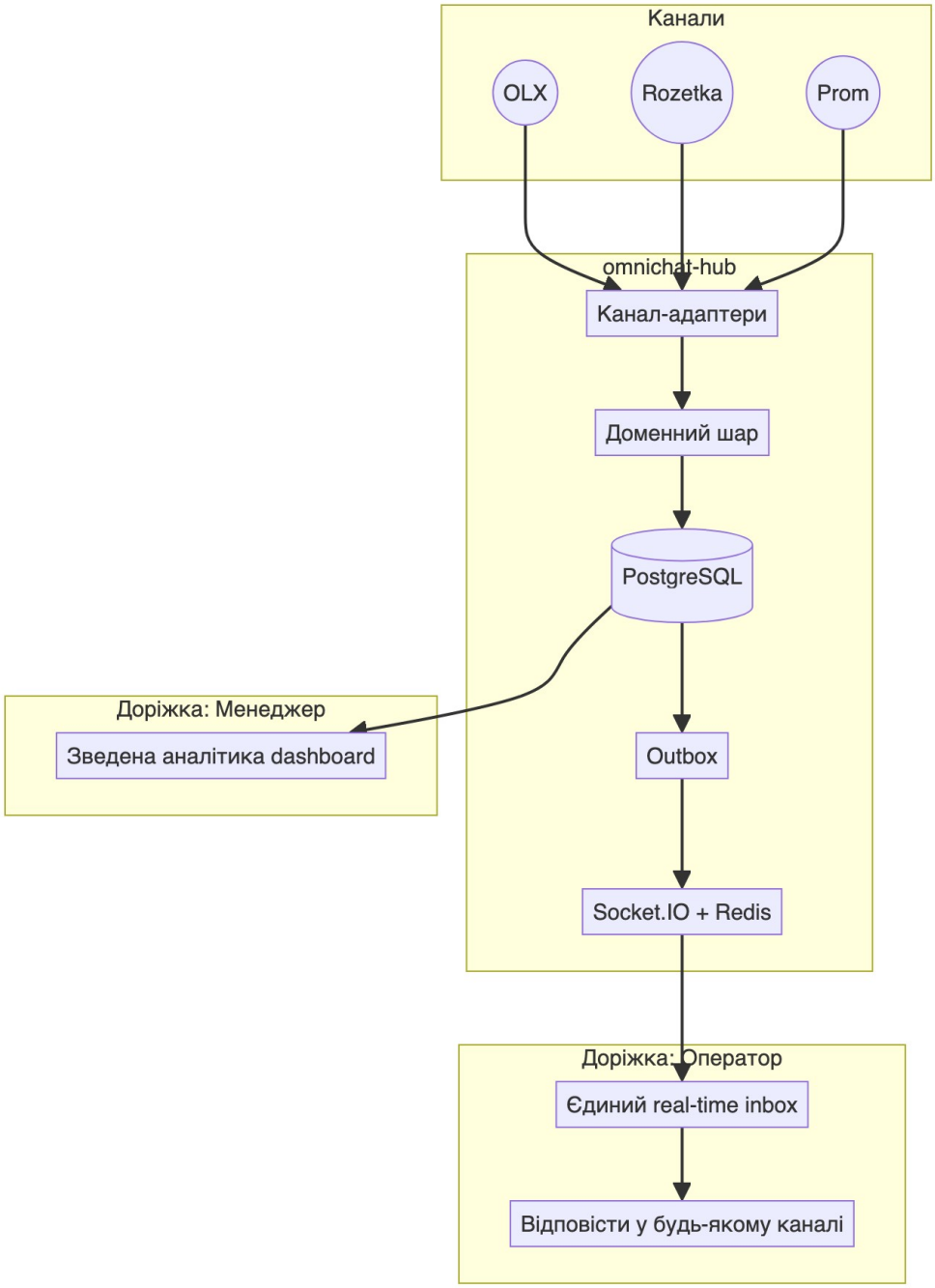
Б.9. Діаграма IDEF3 сценарію гарантованої доставки «ЯК ПОВИННО БУТИ».



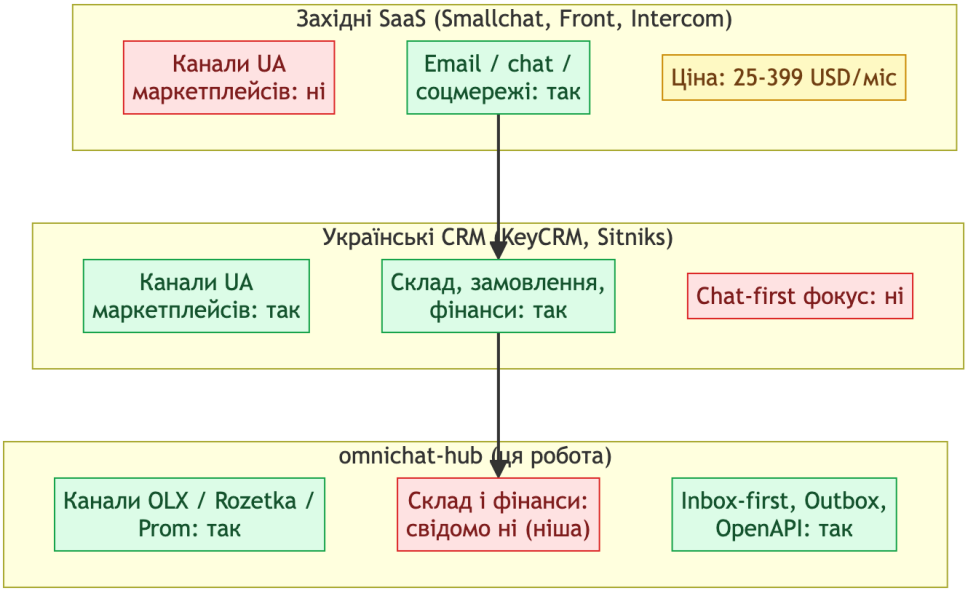
Б.10. Додаткова BPMN-модель бізнес-процесу «ЯК Є» (розрізнена обробка повідомлень).



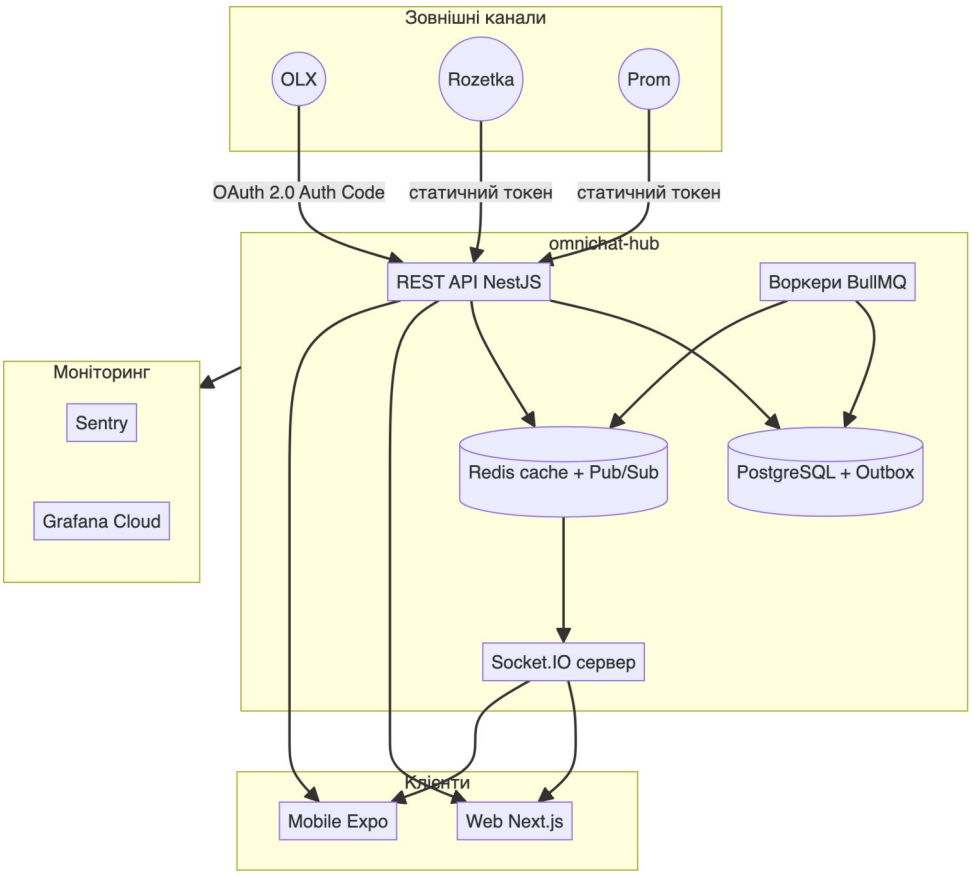
Б.11. Додаткова BPMN-модель бізнес-процесу «ЯК ПОВИННО БУТИ» (омніканальна агрегація через omnichat-hub).



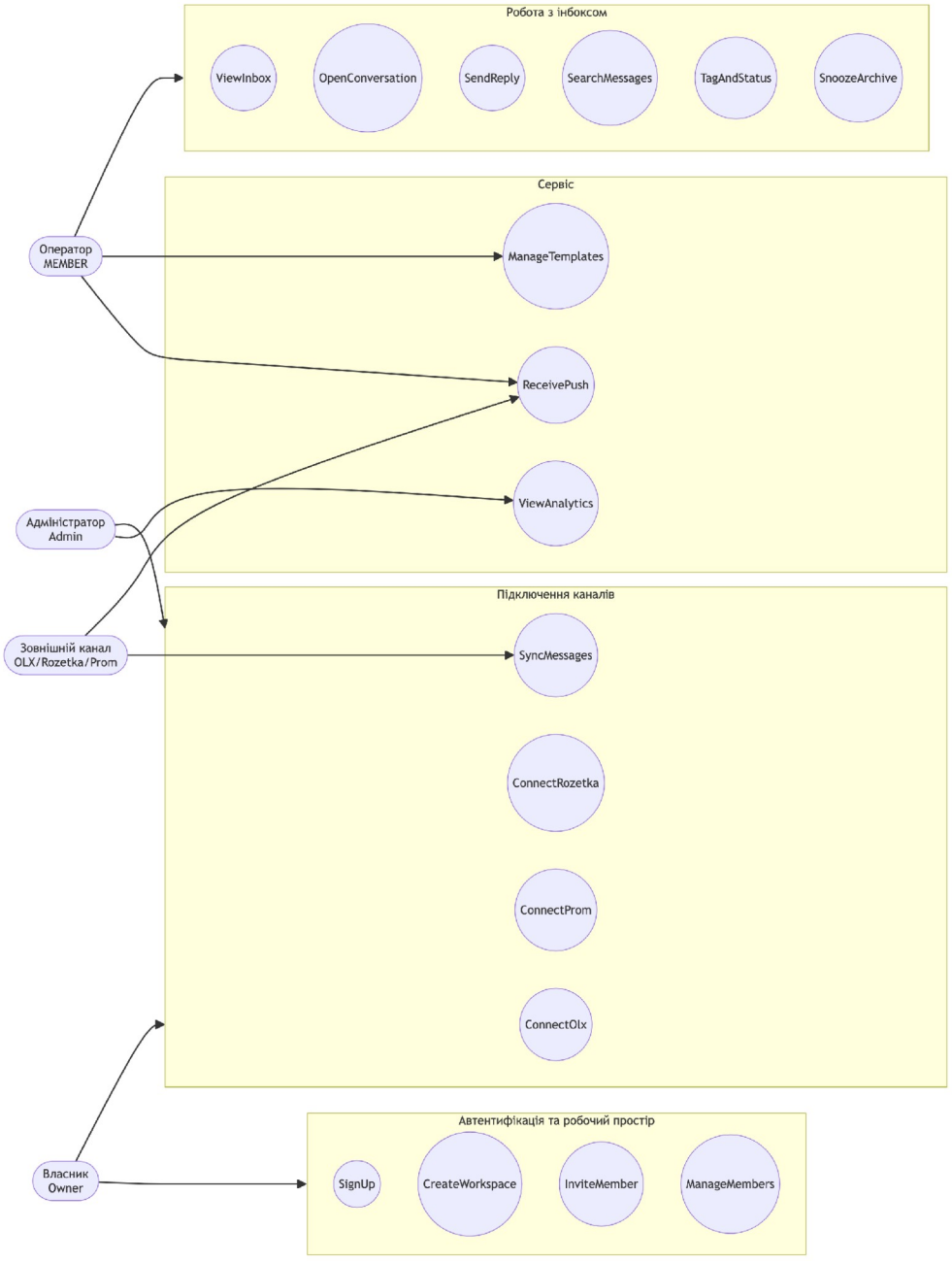
Б.12. Матриця функціонального порівняння omnichat-hub з конкурентами.



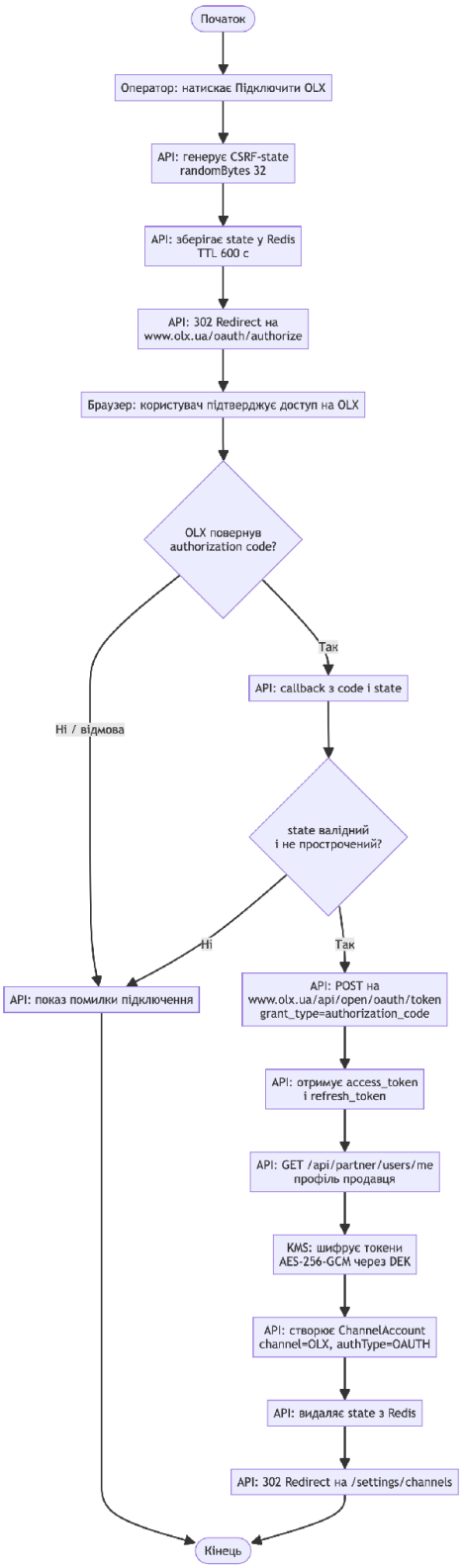
Б.13. Концептуальна архітектура omnichat-hub.



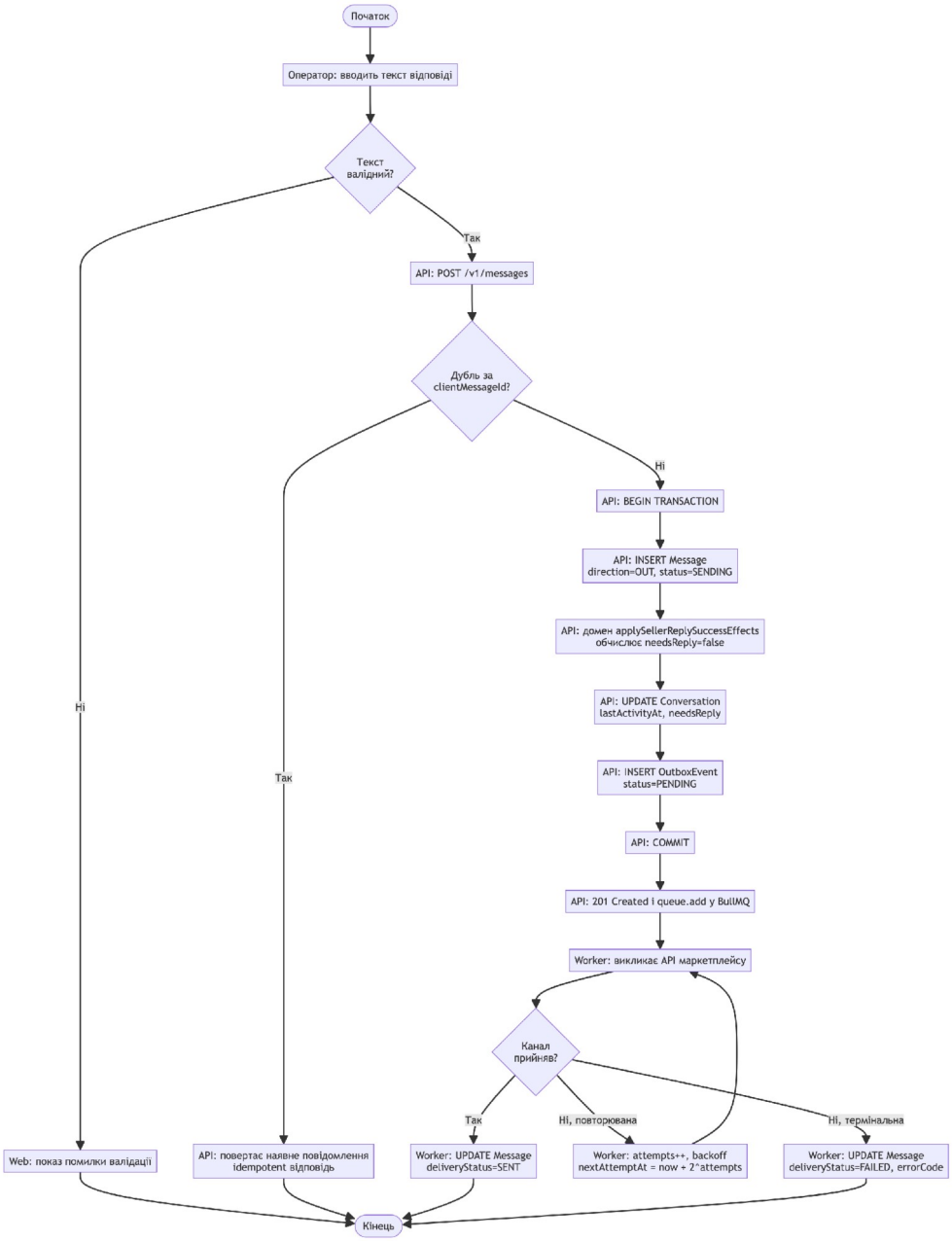
Б.14. Діаграма варіантів використання omnichat-hub.



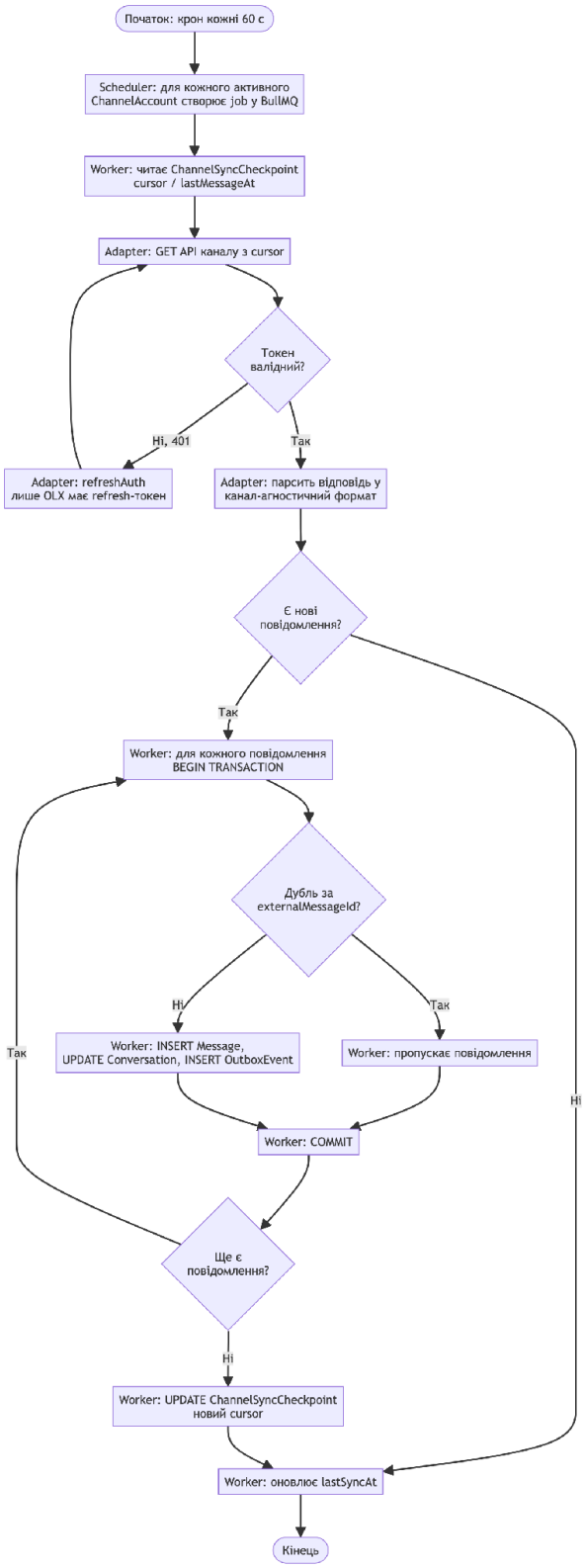
Б.15. Діаграма діяльності «Підключення каналу OLX».



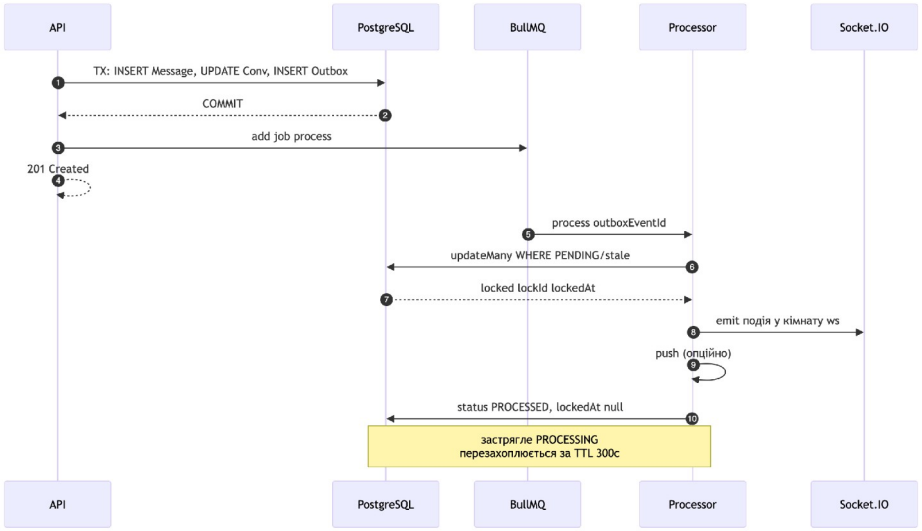
Б.16. Діаграма діяльності «Надсилання вихідного повідомлення».



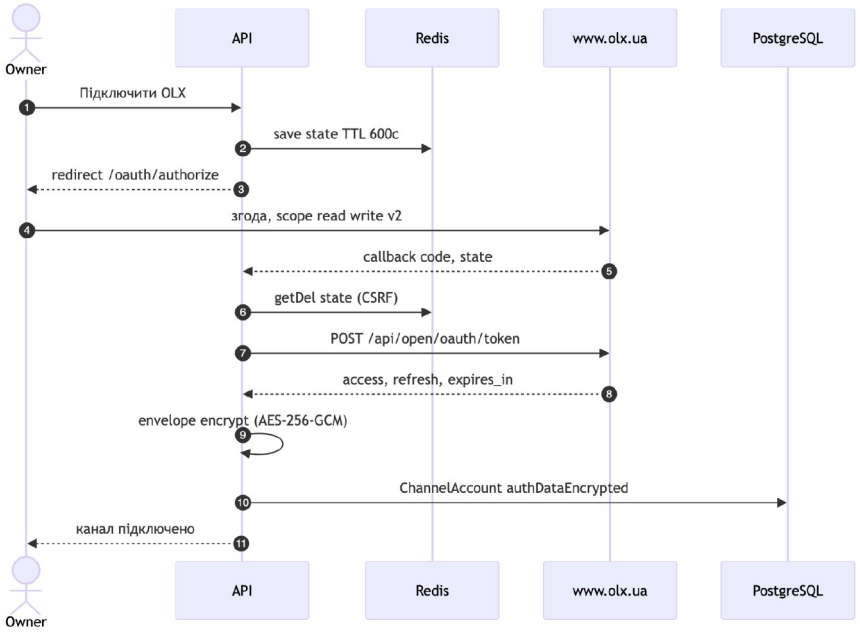
Б.17. Діаграма діяльності «Інкрементальна синхронізація каналу».



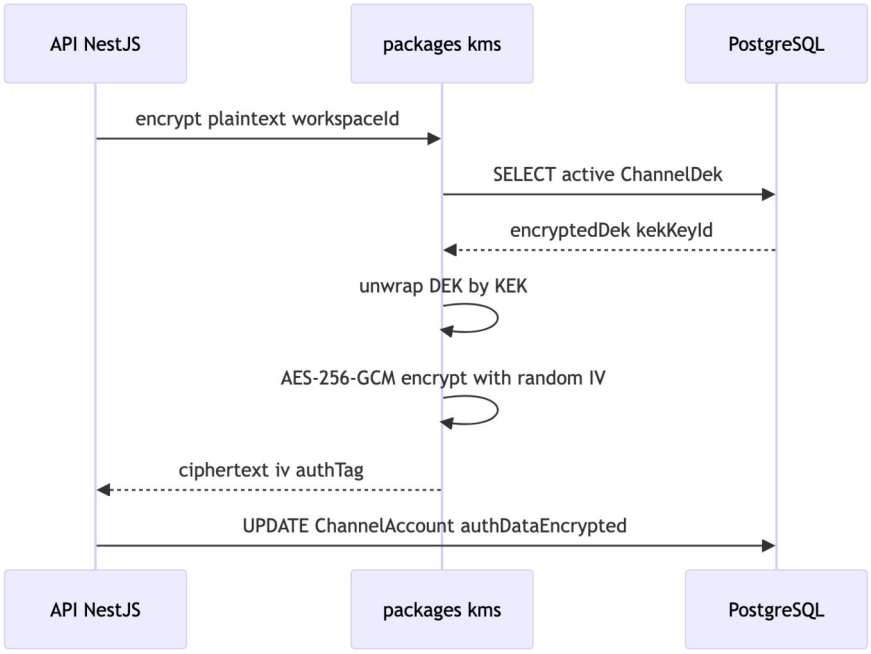
Б.18. Діаграма послідовності транзакційного Outbox-патерну.



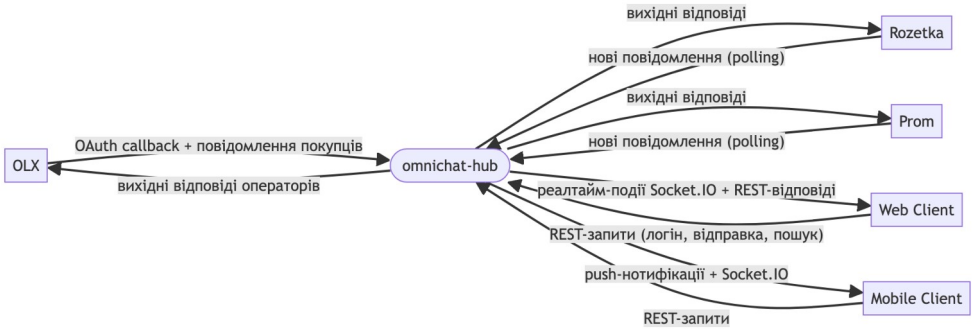
Б.19. Діаграма послідовності OAuth 2.0 потоку OLX.



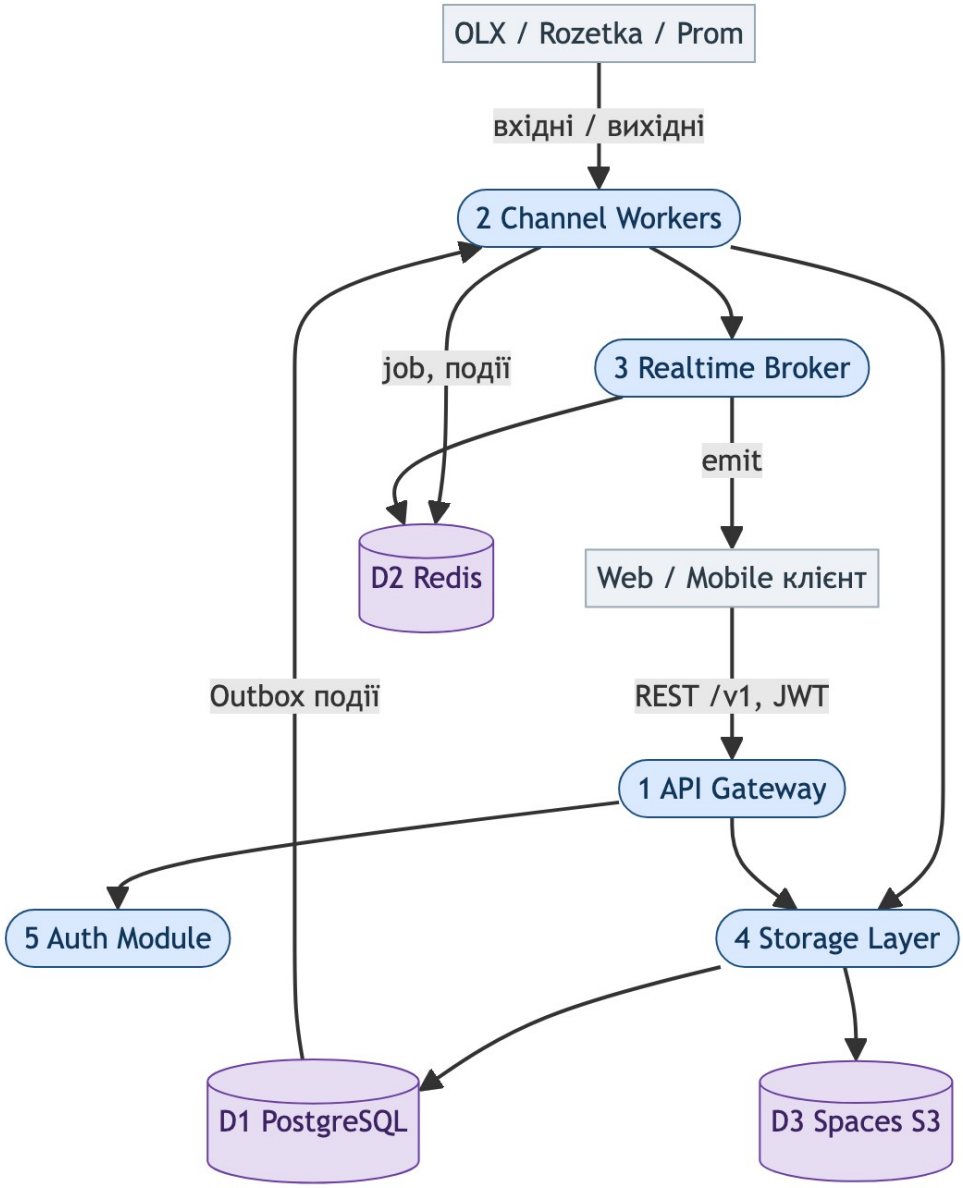
Б.20. Діаграма послідовності envelope encryption.



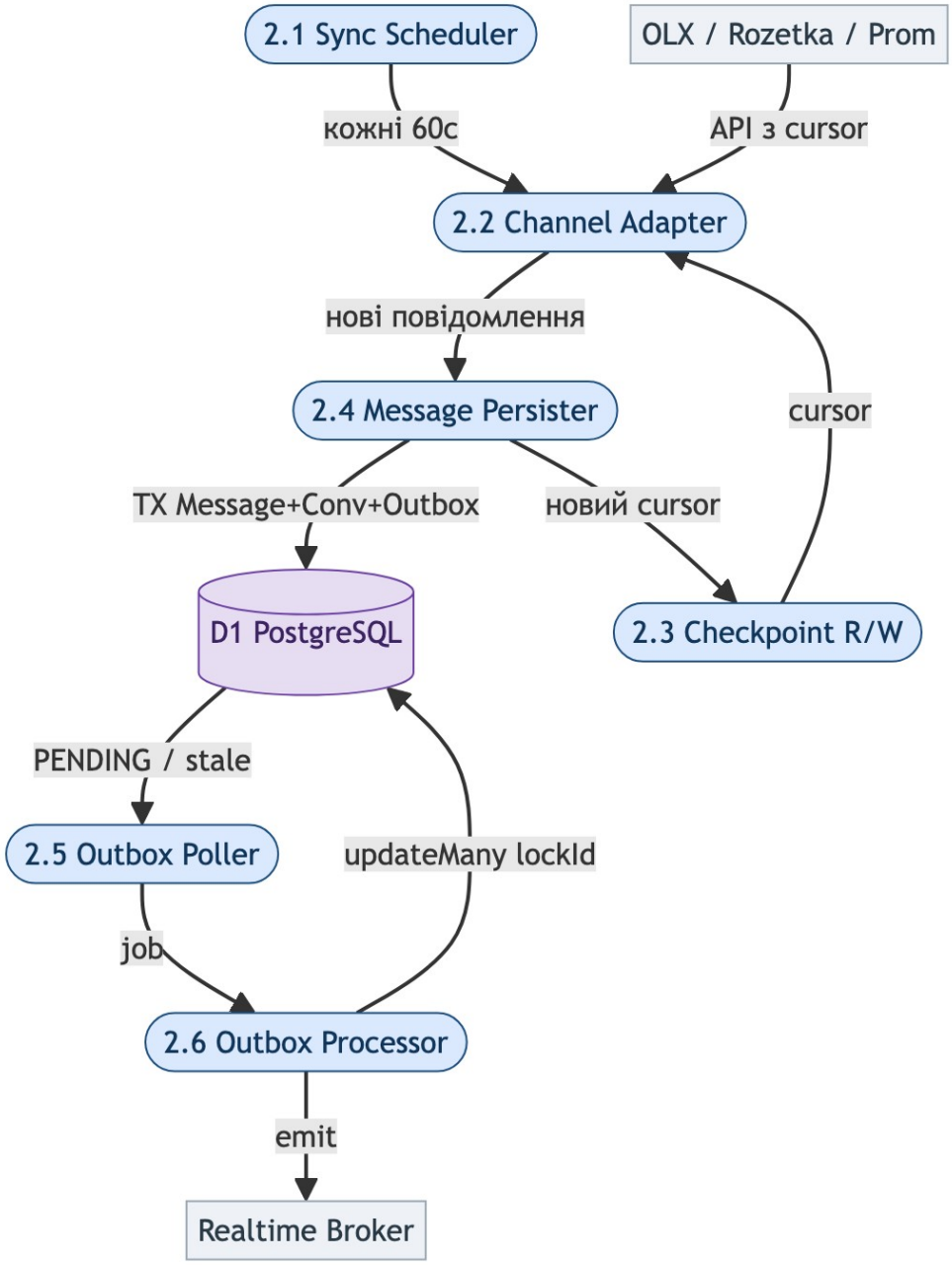
Б.21. DFD рівень 0, контекстна діаграма.



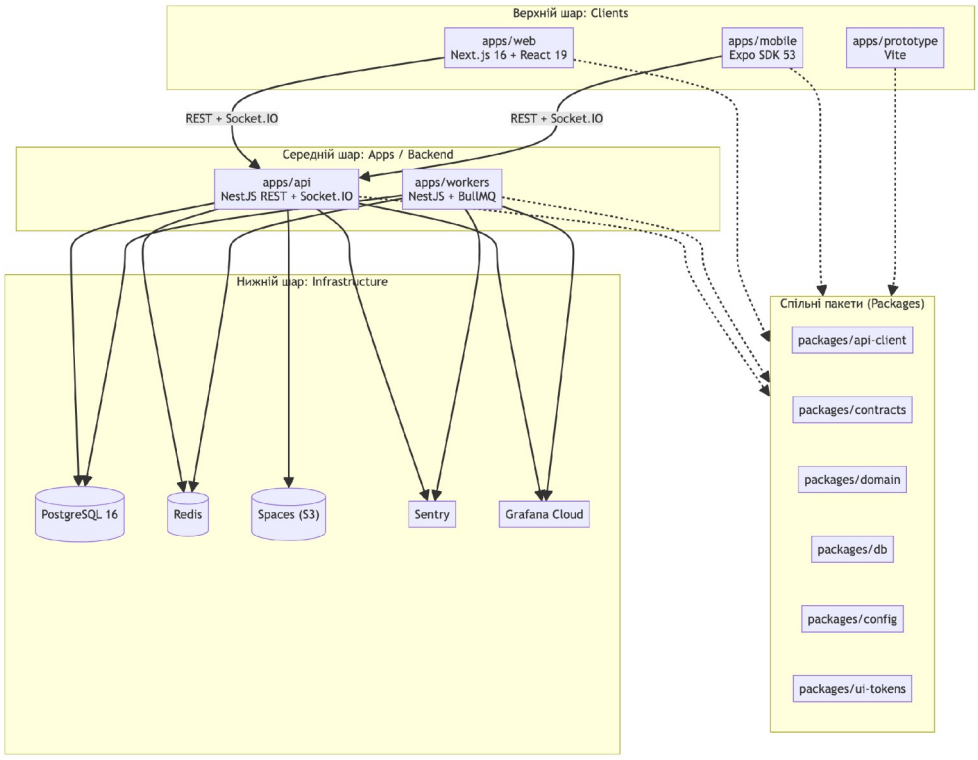
Б.22. DFD рівень 1, декомпозиція omnichat-hub.



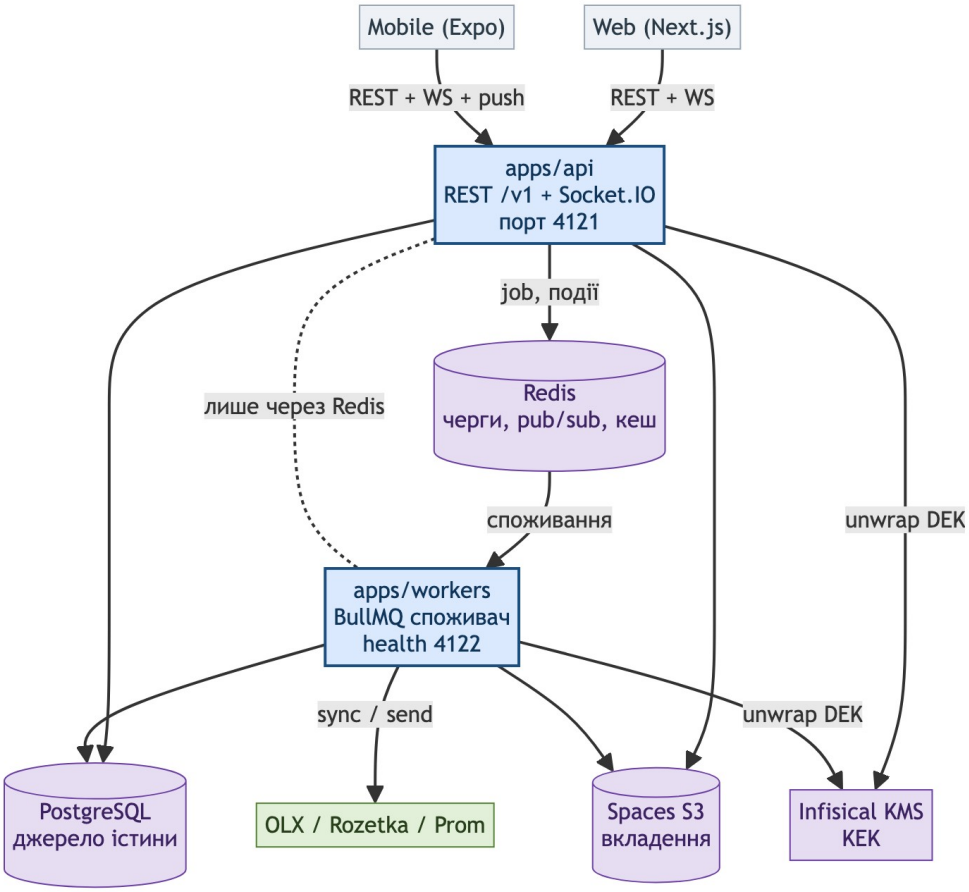
Б.23. DFD рівень 2, деталізація процесу Channel Sync.



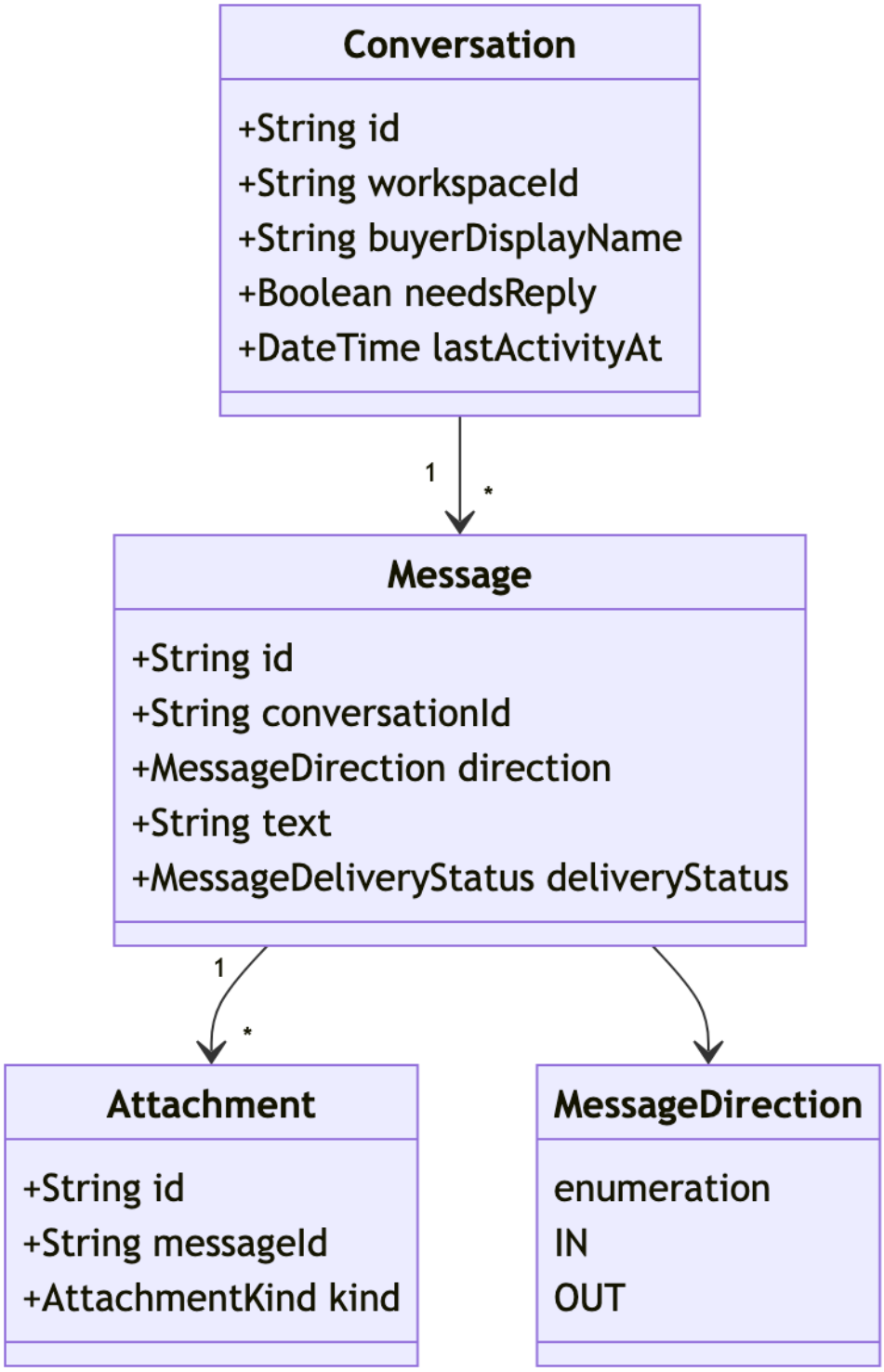
Б.24. Загальна архітектурна діаграма omnichat-hub.



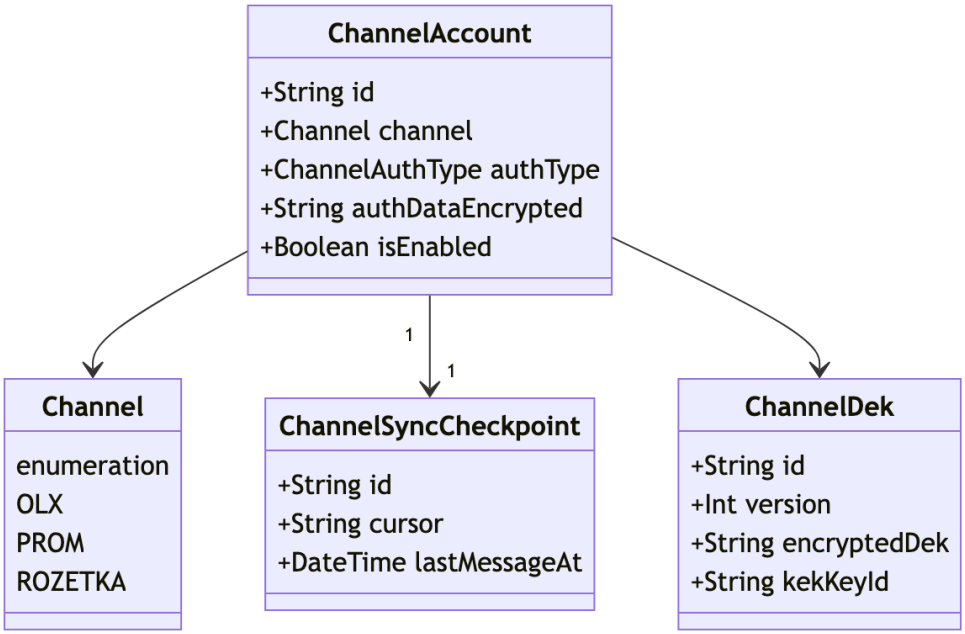
Б.25. Топологія сервісів omnichat-hub.



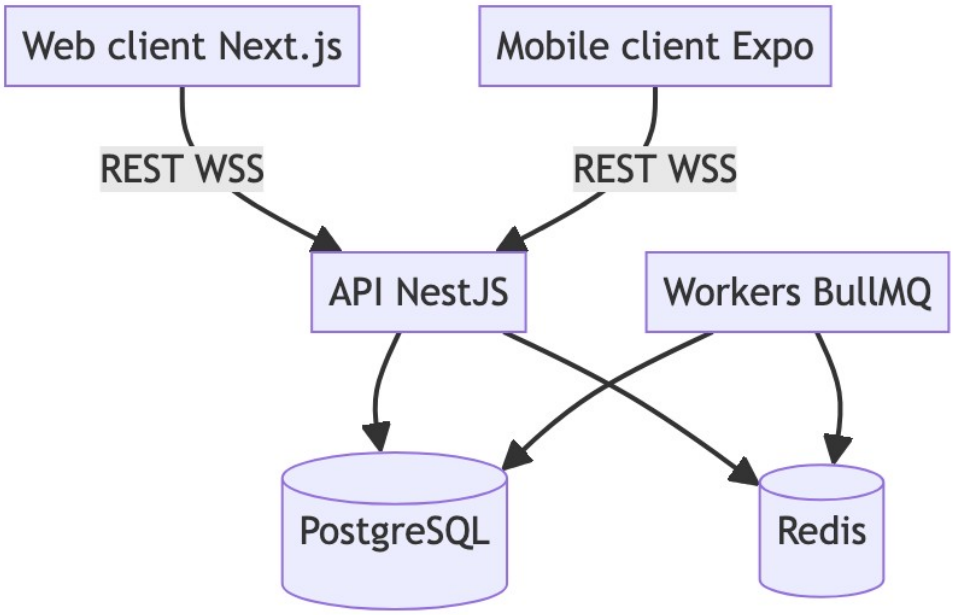
Б.26. Діаграма класів модуля Conversations.



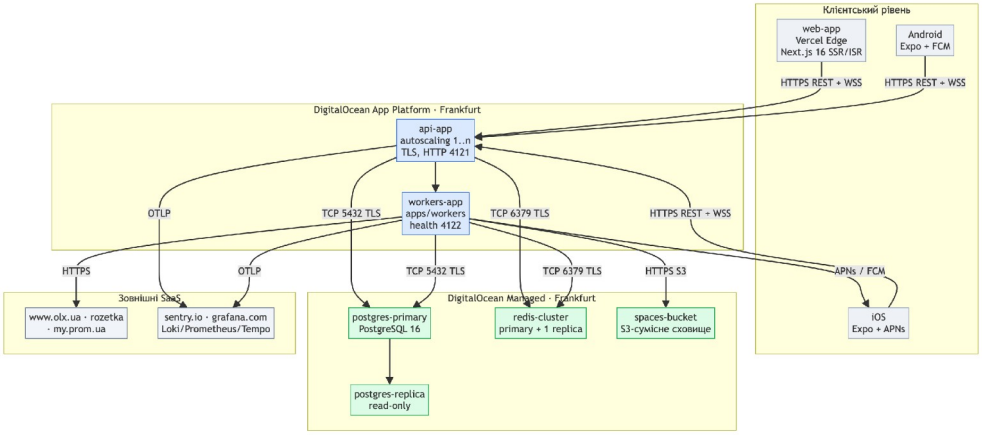
Б.27. Діаграма класів модуля Channels.



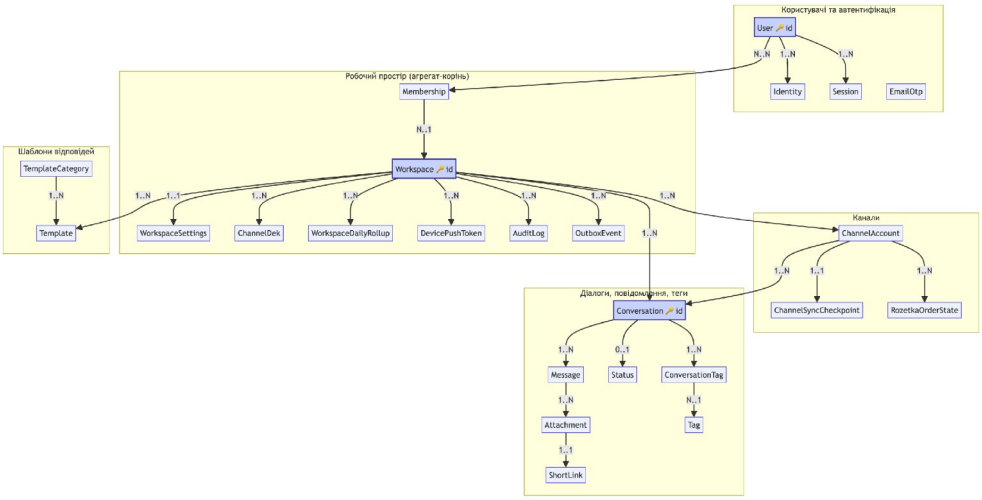
Б.28. Діаграма компонентів.



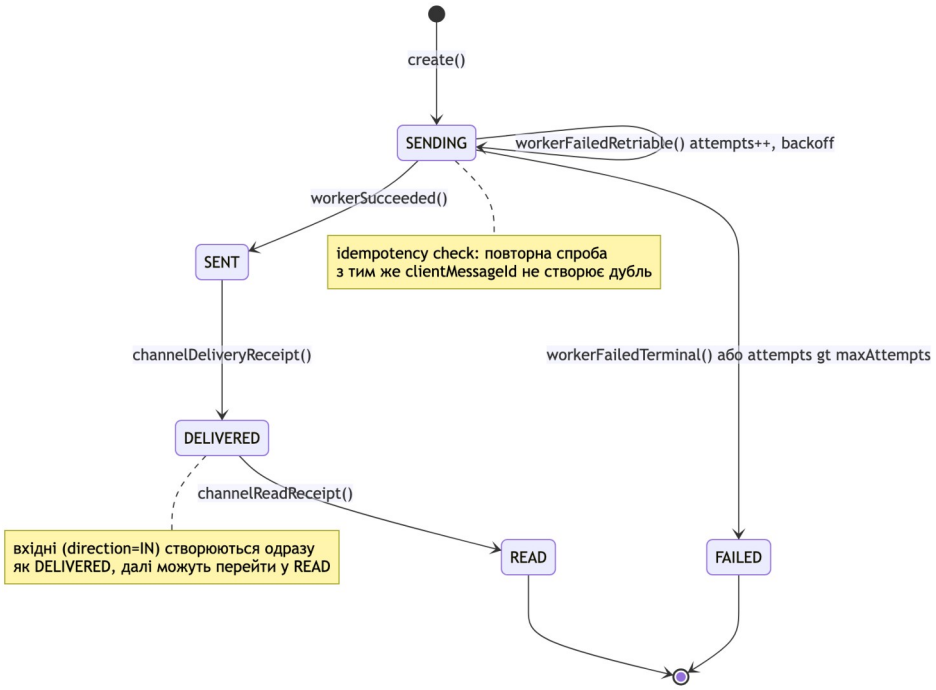
Б.29. Діаграма розгортання.



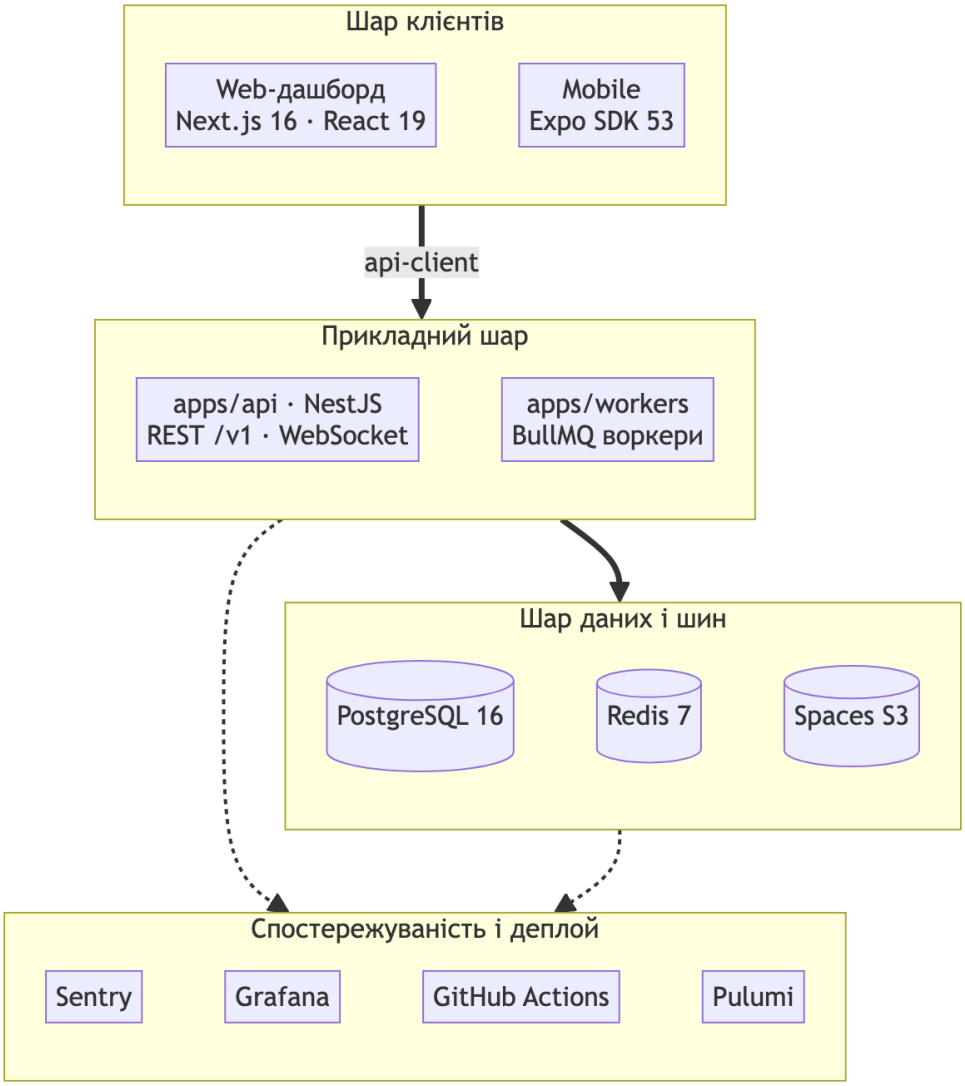
Б.30. Логічна модель даних omnichat-hub у нотації IDEF1X.



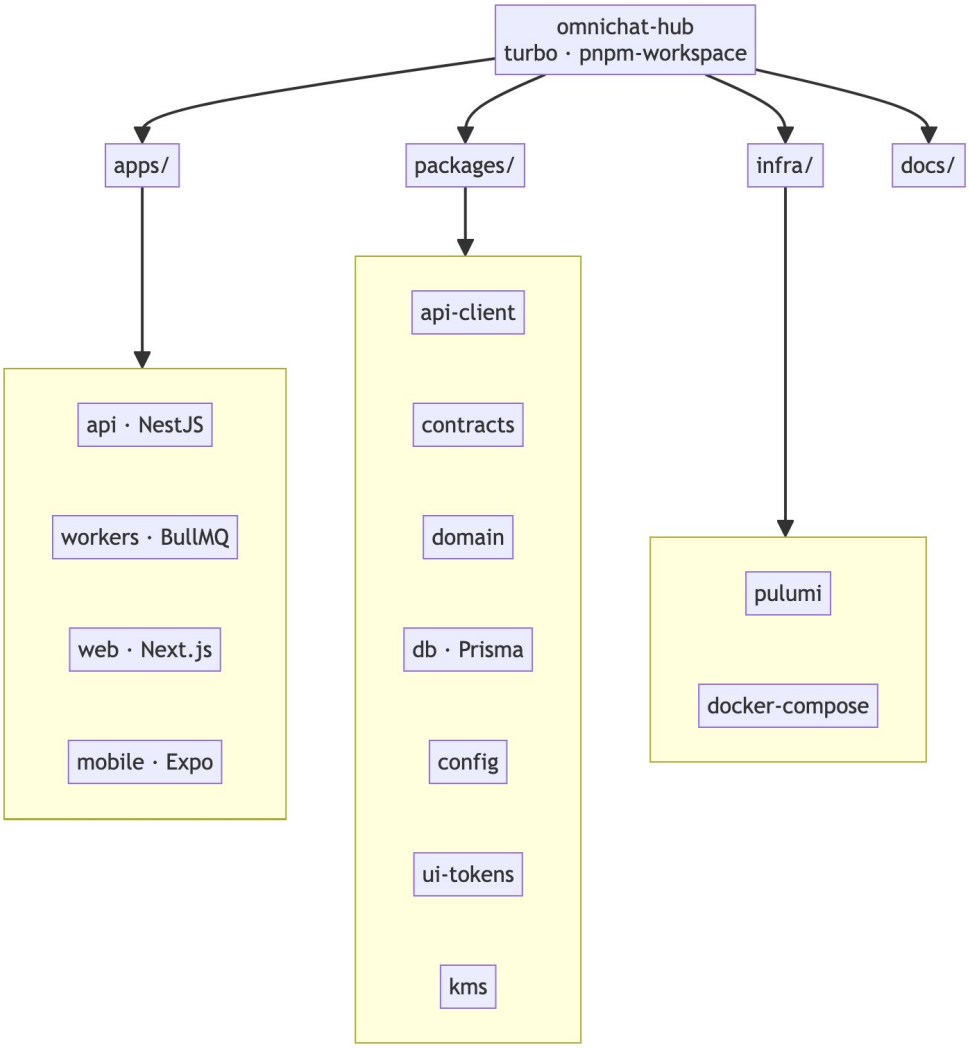
Б.31. Діаграма станів повідомлення.



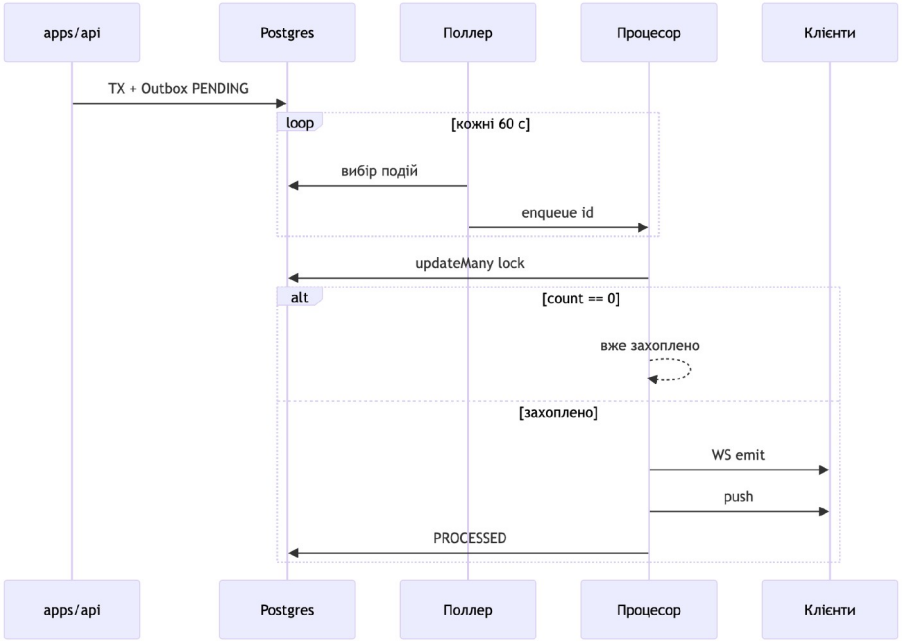
Б.32. Шарова структура технологічного стеку omnichat-hub.



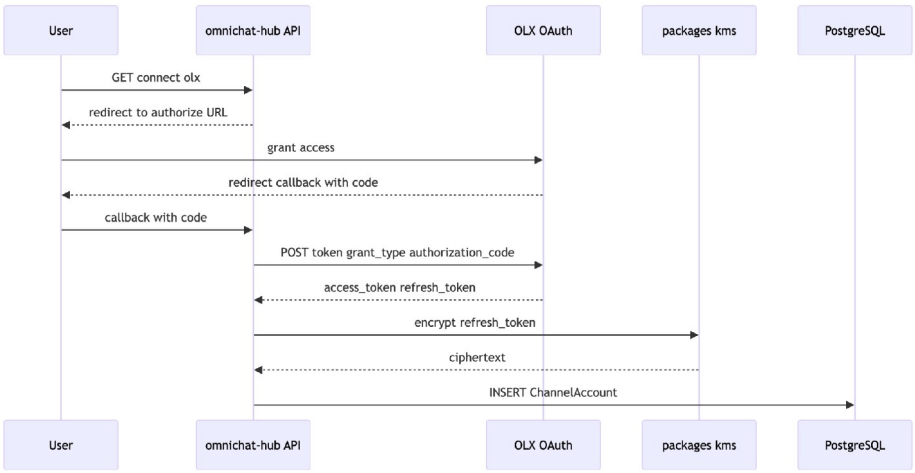
Б.33. Дерево директорій monorepo omnichat-hub.



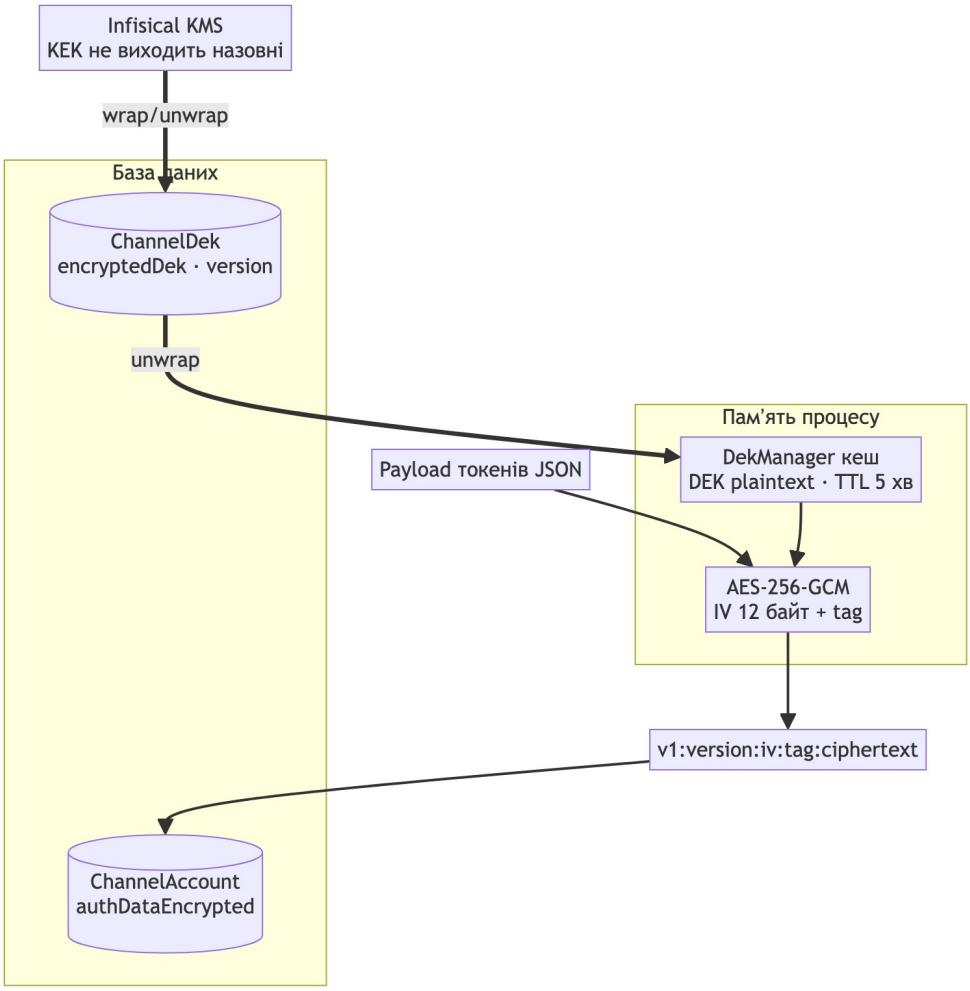
Б.34. Послідовність обробки події транзакційного Outbox.



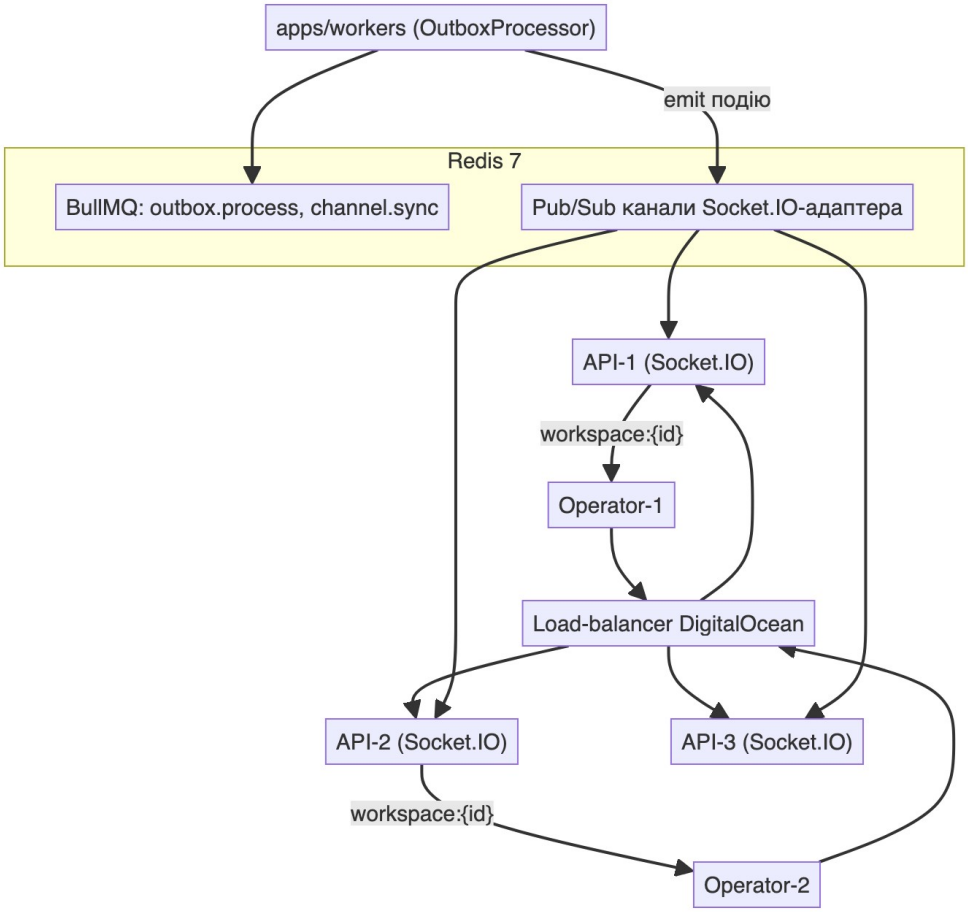
Б.35. Послідовність OAuth-обміну для OLX.



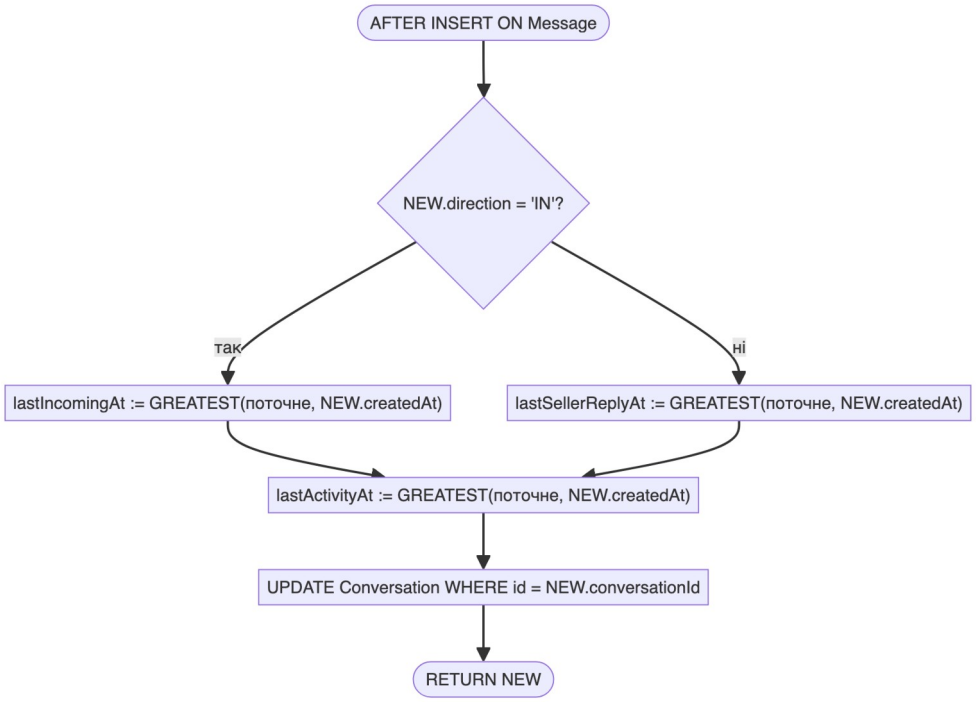
Б.36. Схема двошарового обгорткового шифрування KEK → DEK → payload.



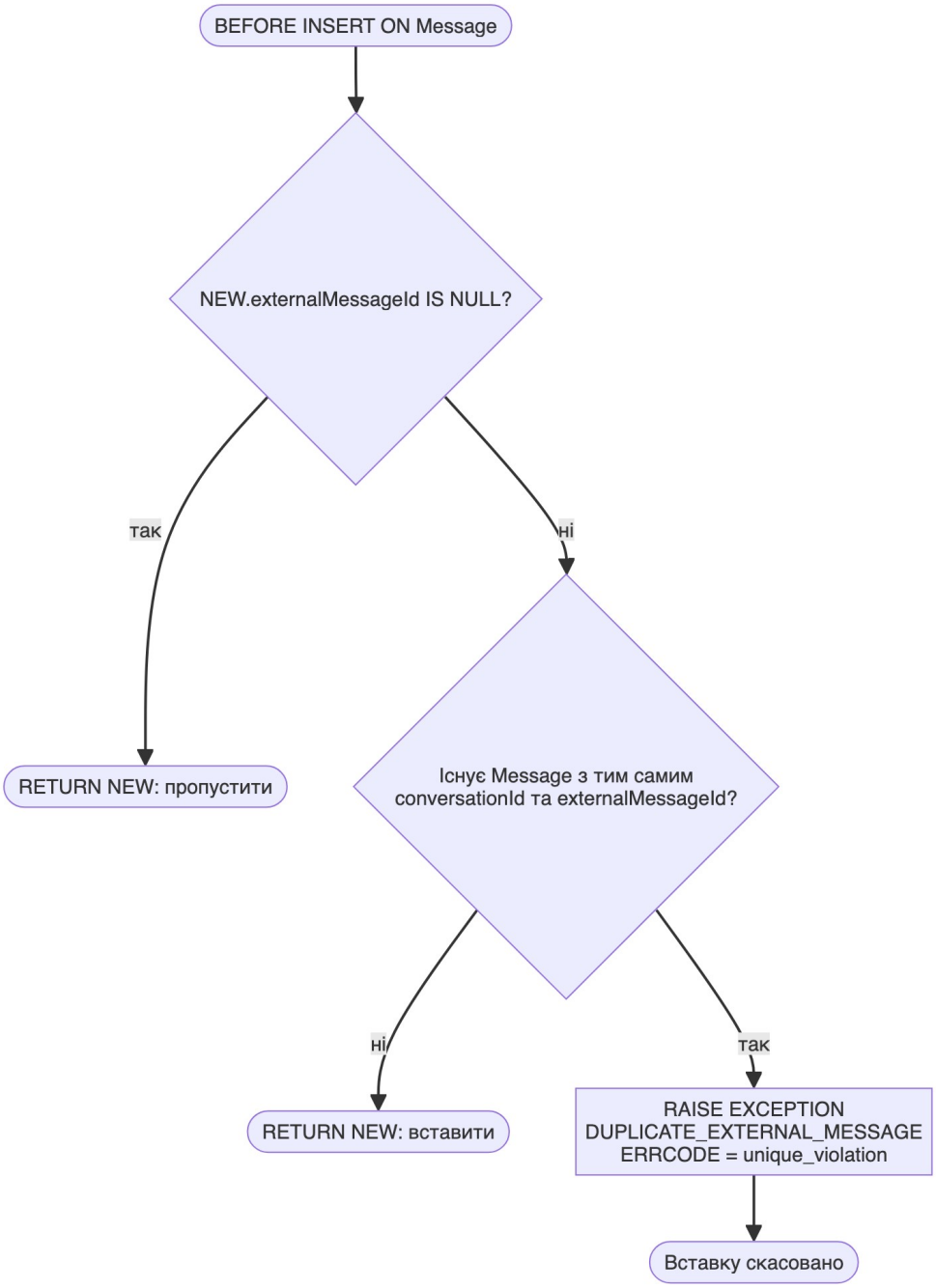
Б.37. Архітектура Socket.IO з Redis-адаптером для горизонтального масштабування.



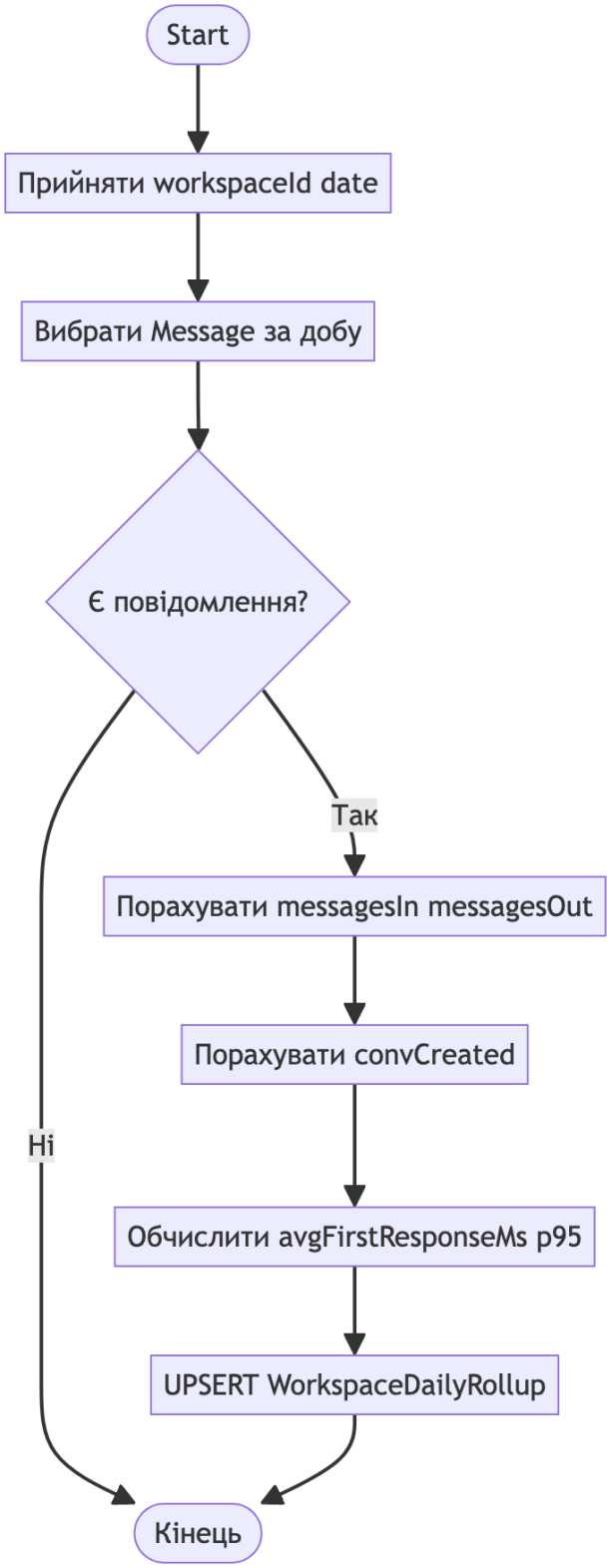
Б.38. Блок-схема алгоритму тригера підтримки lastActivityAt при вставці повідомлення.



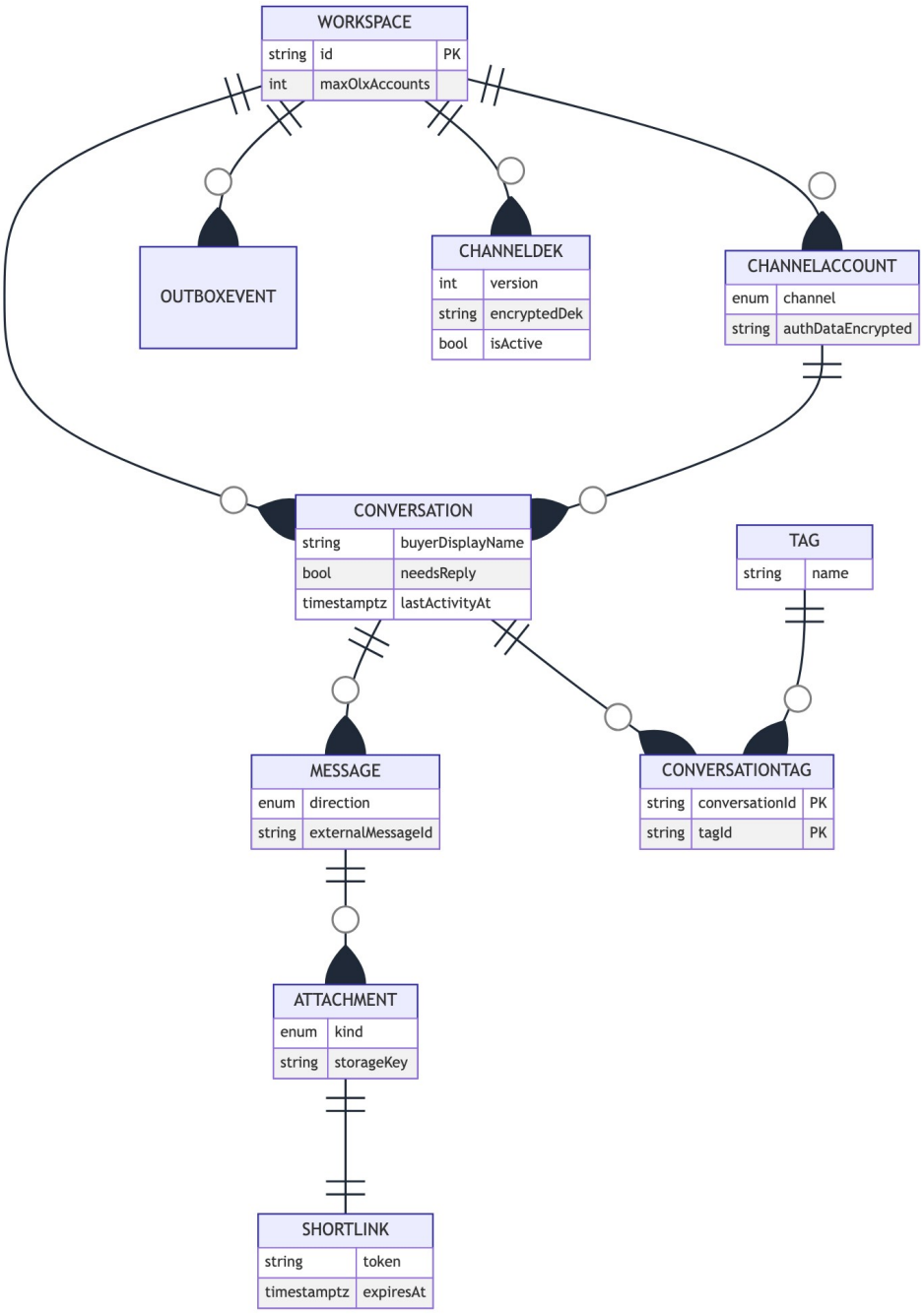
Б.39. Блок-схема алгоритму тригера дедуплікації вхідних повідомлень.



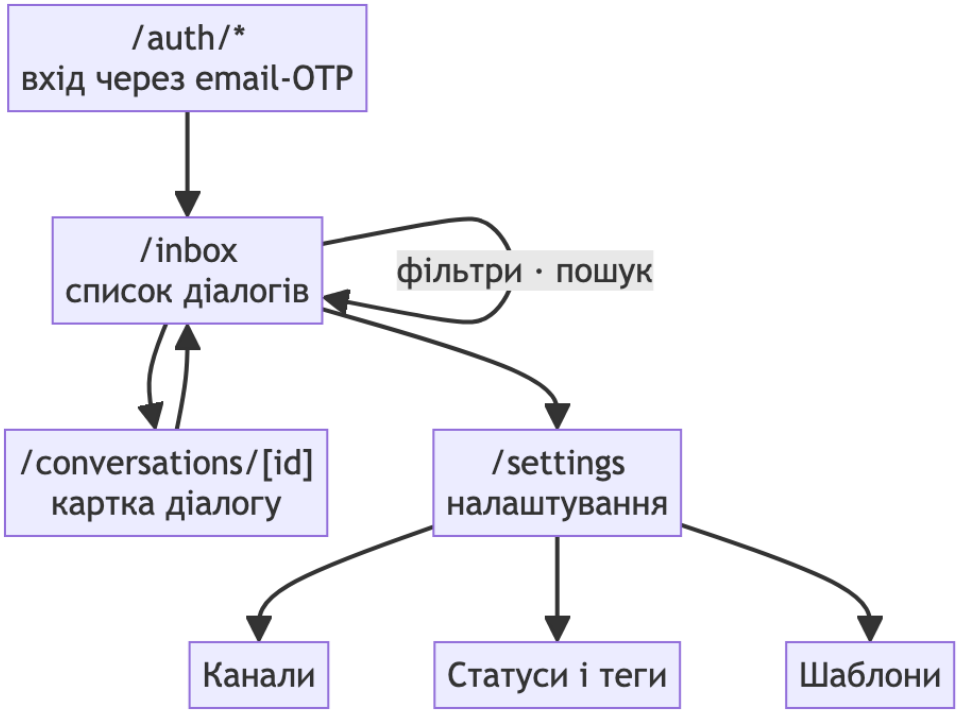
Б.40. Блок-схема алгоритму збереженої процедури fn_compute_daily_rollup.



Б.41. Фізична ER-діаграма ядра бази даних omnichat-hub.



Б.42. Навігаційна схема екранів веб-клієнта оператора.



Б.43. Запуск локального середовища розробки командою pnpm dev.