

ДЕРЖАВНИЙ УНІВЕРСИТЕТ
ІНФОРМАЦІЙНО-КОМУНІКАЦІЙНИХ ТЕХНОЛОГІЙ
НАВЧАЛЬНО-НАУКОВИЙ ІНСТИТУТ
ІНФОРМАЦІЙНИХ ТЕХНОЛОГІЙ
КАФЕДРА ТЕХНОЛОГІЙ ЦИФРОВОГО РОЗВИТКУ

КВАЛІФІКАЦІЙНА РОБОТА

на тему: «Web додаток для знайомств за допомогою мови
програмування TypeScript та Go»

на здобуття освітнього ступеня бакалавра
зі спеціальності 121 Інженерія програмного забезпечення
освітньо-професійної програми «Технології цифрового розвитку»

*Кваліфікаційна робота містить результати власних досліджень. Використання
ідей, результатів і текстів інших авторів мають посилання
на відповідне джерело*

(підпис)

Дмитро ІВАНОВ

Виконав: здобувач вищої освіти групи ТЦР-44

Дмитро ІВАНОВ

Керівник:

Вадим ЧИТУЛЯН

Рецензент:

**ДЕРЖАВНИЙ УНІВЕРСИТЕТ
ІНФОРМАЦІЙНО-КОМУНІКАЦІЙНИХ ТЕХНОЛОГІЙ**

Навчально-науковий інститут інформаційних технологій

Кафедра Технологій цифрового розвитку

Ступінь вищої освіти Бакалавр

Спеціальність 121 Інженерія програмного забезпечення

Освітньо-професійна програма «Технології цифрового розвитку»

ЗАТВЕРДЖУЮ

Завідувач кафедри

Технологій цифрового розвитку

_____ Вікторія ЖЕБКА

« ____ » _____ 2026 р.

**ЗАВДАННЯ
НА КВАЛІФІКАЦІЙНУ РОБОТУ**

Іванову Дмитру Андрійовичу

1. Тема кваліфікаційної роботи: «Web додаток для знайомств за допомогою мови програмування TypeScript та Go»

керівник кваліфікаційної роботи викладач кафедри ТЦР Вадим ЧИТУЛЯН,
затверджені наказом Державного університету інформаційно-комунікаційних технологій від «20» лютого 2026 року №51.

2. Строк подання кваліфікаційної роботи «18» травня 2026 року.

3. Вихідні дані до кваліфікаційної роботи: теоретичні відомості про мікросервісну архітектуру, доменно-орієнтоване проєктування (DDD) та шаблон Clean Architecture; матеріали з геопросторових обчислень (тип POINT, SRID 4326, R-Tree-індекси у MariaDB) та real-time-комунікації на базі gRPC bidirectional streaming і Socket.IO; технічна документація середовищ розробки серверних сервісів (Go 1.26, фреймворк Gin, ORM GORM) та клієнтських додатків (TypeScript 5, Next.js 16, React 19, Redux Toolkit); документація розподілених сховищ (MariaDB з MaxScale, DragonflyDB, ClickHouse, MinIO); документація платформи Telegram Mini Apps та Telegram Bot API; документація технологій контейнеризації (Docker, BuildKit, OCI) та GitOps-розгортання у Kubernetes (FluxCD, Harbor, Trivy).

4. Зміст розрахунково-пояснювальної записки (перелік питань, які потрібно розробити)

1. Огляд та аналіз предметної галузі онлайн-знайомств, існуючих програмних продуктів (Tinder, Bumble, Hinge, Badoo) та сучасних технологічних засобів побудови масштабованих real-time-систем.
2. Проектування мікросервісної архітектури, моделі бази даних із геопросторовою підтримкою, програмного інтерфейсу REST/gRPC та інтерфейсу користувача вебзастосунку для знайомств у форматі Telegram Mini App.
3. Програмна реалізація серверних сервісів мовою Go, проксі-сервісу чатів на базі gRPC bidirectional streaming, клієнтських додатків мовою TypeScript, конвеєра CI/CD з власним кешуванням у MinIO та GitOps-розгортання у Kubernetes-кластері.
4. Тестування розробленої системи (статичний аналіз, юніт- та інтеграційне тестування, CVE-сканування контейнерних образів).

5. Перелік графічного матеріалу: *презентація*

1. Аналіз недоліків існуючих рішень (аналогів) та обґрунтування актуальності розробки.
2. Концепція та переваги поєднання dating- і party-режимів у форматі Telegram Mini App.
3. Програмні засоби реалізації, п'ятишаровий стек технологій та архітектура розгортання у Kubernetes.
4. Моделювання вимог (Use Case діаграма).
5. Архітектура реляційної бази даних (ER-діаграма).
6. Алгоритм роботи системи валідаційних rolling-хешів та шардування real-time-чатів через Socket.IO та gRPC.
7. Екранні форми Telegram Mini App (стрічка матчмейкінгу, чат, профіль).
8. Апробація результатів дослідження.

6. Дата видачі завдання «20» лютого 2026 року.

КАЛЕНДАРНИЙ ПЛАН

№ з/п	Назва етапів кваліфікаційної роботи	Строк виконання етапів роботи	Примітка
1	Підбір та аналіз науково-технічної літератури	20.02.2026 – 22.02.2026	Виконано
2	Аналіз та дослідження існуючих аналогів (dating-сервісів та Telegram Mini Apps)	23.02.2026 – 28.02.2026	Виконано
3	Огляд технологій мікросервісної архітектури, gRPC, Socket.IO, геопросторових індексів та контейнерної безпеки	01.03.2026 – 12.03.2026	Виконано
4	Проектування мікросервісної архітектури, моделі бази даних та UX/UI інтерфейсу Telegram Mini App	13.03.2026 – 26.03.2026	Виконано
5	Програмна реалізація (Frontend, Backend, ChatsProxy) та налаштування CI/CD і GitOps-розгортання	27.03.2026 – 30.04.2026	Виконано
6	Тестування системи (статичний аналіз, юніт-, інтеграційне тестування, Trivy-сканування)	01.05.2026 – 10.05.2026	Виконано
7	Оформлення роботи: вступ, висновки, реферат	11.05.2026 – 13.05.2026	Виконано
8	Розробка демонстраційних матеріалів (презентація)	14.05.2026 – 15.05.2026	Виконано
9	Попередній захист роботи	16.05.2026 – 18.05.2026	Виконано

Здобувач вищої освіти _____
(підпис)

Дмитро ІВАНОВ

Керівник
кваліфікаційної роботи _____
(підпис)

Вадим ЧИТУЛЯН

РЕФЕРАТ

Текстова частина кваліфікаційної роботи на здобуття освітнього ступеня бакалавра: 91 сторінка, 33 рисунки, 1 таблиця, 28 джерел.

Мета роботи – спрощення та прискорення процесу пошуку партнерів для спілкування і знайомств шляхом розробки масштабованого, відмовостійкого web-додатку з геолокаційним пошуком, обміном повідомленнями у реальному часі та системою заходів (party-events), реалізованого засобами мов TypeScript і Go з використанням мікросервісної архітектури.

Об'єкт дослідження – процес забезпечення комунікації між користувачами сучасних соціальних сервісів знайомств та інформаційно-аналітичної підтримки рішень про вибір партнера.

Предмет дослідження – програмне забезпечення для організації знайомств, побудоване на базі мікросервісної архітектури з використанням мов програмування TypeScript і Go, технології gRPC з потоковою передачею даних, розподілених сховищ MariaDB, DragonflyDB, ClickHouse, MinIO та оркестрування у Kubernetes-кластері.

Методи дослідження – системний аналіз предметної галузі, порівняльний аналіз технологій, математичне моделювання навантаження, методи проектування програмного забезпечення (UML), статичний аналіз, юніт- та інтеграційне тестування, методи доказової інженерії безпеки контейнерних образів.

Короткий зміст роботи: проаналізовано предметну галузь онлайн-знайомств та існуючі технологічні рішення; визначено об'єкт, предмет, мету та задачі роботи; спроектовано та реалізовано тринадцятисервісну архітектуру з повним покриттям циклу від реєстрації через Telegram Mini App до обміну повідомленнями у реальному часі та проведення групових заходів; розроблено власну систему кешування CI з MinIO та конвеєр GitOps-розгортання у Kubernetes на базі FluxCD; реалізовано контейнерні образи з нульовою поверхнею атаки (0 CVE) для всіх серверних сервісів та власний захищений ~600 КБ образ nginx з env-injection для фронтенду; виконано статичний аналіз 41

лінтером, юніт-, інтеграційне та ручне тестування; обґрунтовано та апробовано результати, опубліковано тези на двох наукових конференціях.

Галузь застосування – інформаційні системи соціальних сервісів, web-платформи знайомств, Telegram Mini Apps; підходи й програмні артефакти роботи можуть бути застосовані в інших галузях, де необхідна горизонтально масштабована real-time-комунікація на базі gRPC та Socket.IO.

КЛЮЧОВІ СЛОВА: ВЕБ-ДОДАТОК, МІКРОСЕРВІСИ, GO, TYPESCRIPT, GRPC, SOCKET.IO, KUBERNETES, GITOPS, MARIADB, DRAGONFLYDB, ТЕЛЕГРАМ MINI APP, КОНТЕЙНЕРНА БЕЗПЕКА, ПРОТОКОЛИ ОБМІНУ ПОВІДОМЛЕННЯМИ.

ЗМІСТ

ПЕРЕЛІК УМОВНИХ ПОЗНАЧЕНЬ.....	10
ВСТУП.....	11
1 АНАЛІЗ ПРЕДМЕТНОЇ ГАЛУЗІ ТА ЗАСОБІВ РОЗРОБКИ.....	15
1.1 Опис задачі та предметної галузі.....	15
1.2 Аналіз існуючих програмних продуктів.....	21
1.3 Огляд технологій, фреймворків та програмних засобів.....	23
1.4 Об'єкт, предмет, мета та задачі роботи.....	30
2 ПРОЕКТУВАННЯ СИСТЕМИ.....	32
2.1 Архітектура системи.....	32
2.2 Моделювання вимог.....	41
2.3 Проектування бази даних.....	49
2.4 Проектування програмного інтерфейсу.....	54
2.5 Проектування інтерфейсу користувача.....	58
3 РЕАЛІЗАЦІЯ ТА ТЕСТУВАННЯ СИСТЕМИ.....	62
3.1 Структура проекту й організація коду.....	62
3.2 Реалізація серверної частини.....	64
3.3 Реалізація проксі-сервісу чатів та інших сервісів.....	70
3.4 Реалізація клієнтської частини.....	75
3.5 Контейнерна безпека та CI/CD.....	79
3.6 Тестування системи.....	85
ВИСНОВКИ.....	94
ПЕРЕЛІК ПОСИЛАНЬ.....	99
ДОДАТОК А. ДЕМОНСТРАЦІЙНІ МАТЕРІАЛИ.....	102

ПЕРЕЛІК УМОВНИХ ПОЗНАЧЕНЬ

API – Application Programming Interface
CI/CD – Continuous Integration / Continuous Delivery
CVE – Common Vulnerabilities and Exposures
DTO – Data Transfer Object
FluxCD – GitOps-контролер для Kubernetes
GIS – Geographic Information System
gRPC – Google Remote Procedure Call
HMAC – Hash-based Message Authentication Code
HPA – Horizontal Pod Autoscaler
JWT – JSON Web Token
MTProto – Mobile Transport Protocol (Telegram)
OCI – Open Container Initiative
ORM – Object-Relational Mapping
OSM – OpenStreetMap
REST – Representational State Transfer
RBAC – Role-Based Access Control
SDK – Software Development Kit
SemVer – Semantic Versioning
SPA – Single-Page Application
SRID – Spatial Reference System Identifier
SRV – Service Resource Record (DNS)
SSR – Server-Side Rendering
WebP – Web Picture Format
WSS – WebSocket Secure
WKT – Well-Known Text (формат геоданих)

ВСТУП

Цифрова трансформація сфери соціальної взаємодії докорінно змінила способи знайомства людей. Якщо ще 15 років тому встановлення нових особистих зв'язків відбувалося переважно у фізичному просторі – навчальних закладах, спільнотах за інтересами, місцях відпочинку, – то у 2025 році переважна частина перших контактів зароджується в межах цифрових платформ. За оцінками аналітичних агенцій, обсяг світового ринку онлайн-знайомств досягає одинадцяти мільярдів доларів США щорічно та зростає у середньому на 11% на рік (рисунки 1.1), а кількість активних користувачів подібних сервісів перевищує триста мільйонів осіб.

Сучасний користувач формулює до платформи знайомств доволі суворий перелік вимог: миттєвий обмін повідомленнями без помітних затримок, точний геолокаційний пошук, висока приватність персональних даних, прозорі правила монетизації, доступність як на смартфоні, так і у вбудованому в улюблений месенджер форматі, що підтверджується популярністю таких рішень. Окрему підгрупу складають так звані «події» (party-events) – групові спільні зустрічі, що формуються навколо конкретного інтересу, локації чи дати. Жоден класичний свайп-сервіс не дає повноцінної підтримки таких сценаріїв, тоді як їхня частка у соціальній активності молоді 16-35 років стрімко зростає.

Поряд з цим, технологічна сторона побудови подібної системи також зазнає змін. Перехід на хмарну (cloud-native) парадигму, поширення GitOps-практик, посилення вимог до безпеки контейнерних образів та зниження частки класичного REST на користь бінарних протоколів зі строгою типізацією роблять актуальним дослідження сучасних інженерних рішень у комплексі. Зокрема, мови Go та TypeScript у поєднанні з технологіями gRPC, Kubernetes, FluxCD дозволяють побудувати горизонтально масштабовані системи з мінімальною операційною складністю та низькою сукупною вартістю володіння. Однак на стику цих технологій залишається низка нерозв'язаних задач: як забезпечити сублінійну

затримку при доставці повідомлень мільйонам одночасних користувачів, як гарантувати нульову поверхню атаки контейнерів для всіх компонентів системи, як зменшити час повторної збірки конвеєру СІ у десятки разів за допомогою власного механізму кешування, як організувати безпечну та зручну авторизацію через Telegram Mini App без відмови від традиційних сценаріїв входу за email/паролем. Усі ці питання знаходяться у фокусі цієї роботи.

Крім того, попередні розділи показали, що спеціалізована робота над тематикою комбінованої dating-+-party-платформи, реалізованої як Telegram Mini App, відсутня у вітчизняному просторі наукових робіт у галузі програмної інженерії. Результати дослідження апробовані у вигляді тез на двох всеукраїнських науково-практичних конференціях [27; 28].

Мета роботи – спрощення та прискорення процесу пошуку партнерів для спілкування і знайомств шляхом розробки масштабованого, відмовостійкого web-додатку із геолокаційним пошуком, обміном повідомленнями у реальному часі та системою заходів, реалізованого засобами мов TypeScript і Go з використанням мікросервісної архітектури.

Досягнення мети передбачає розв'язання таких задач:

1. Виконати аналіз предметної галузі онлайн-знайомств, систематизувати функціональні та нефункціональні вимоги до системи такого класу.
2. Виконати порівняльний аналіз існуючих програмних продуктів та обґрунтувати актуальність розробки системи з обраним поєднанням властивостей.
3. Дослідити сучасні технології, фреймворки та програмні засоби, придатні для побудови масштабованої real-time-системи; обґрунтувати їхній вибір.
4. Спроекувати мікросервісну архітектуру системи з обґрунтуванням способу розмежування доменів та механізмів міжсервісної комунікації.
5. Спроекувати модель бази даних, що підтримує геолокаційний пошук, систему 1-to-1 та групових чатів і виконання транзакційних операцій з валютами.

6. Спроектувати програмний інтерфейс REST та gRPC, події системи real-time-обміну повідомленнями, інтерфейс мобільного клієнта.
7. Реалізувати ключові серверні сервіси мовою Go, проксі-сервіс чату на базі gRPC bidirectional streaming, клієнтський додаток на TypeScript.
8. Реалізувати конвеєр CI/CD з власним кешуванням у MinIO, ущільненими контейнерними образами на базі scratch та власним захищеним базовим образом nginx, налаштувати GitOps-розгортання у Kubernetes-кластері за допомогою FluxCD.
9. Виконати статичний аналіз, юніт- та інтеграційне тестування, оцінити покриття та безпекові показники системи.

Об'єкт дослідження – процес забезпечення комунікації між користувачами сучасних соціальних сервісів знайомств та інформаційно-аналітичної підтримки рішень про вибір партнера.

Предмет дослідження – програмне забезпечення для організації знайомств, побудоване на базі мікросервісної архітектури з використанням мов програмування TypeScript і Go, технології gRPC з потоковою передачею даних, розподілених сховищ MariaDB, DragonflyDB, ClickHouse, MinIO та оркестрування у Kubernetes-кластері.

Методи дослідження – системний аналіз предметної галузі, порівняльний аналіз технологій, методи об'єктно-орієнтованого та доменно-орієнтованого проектування, моделювання бази даних з використанням нотації «сутність-зв'язок», UML-моделювання вимог, методи горизонтального масштабування та шардування, методи інженерії безпеки контейнерних образів, статичний аналіз вихідного коду, юніт- та інтеграційне тестування, експериментальне вимірювання продуктивності та часу збирання конвеєрів CI/CD.

Розроблено та розгорнуто у Kubernetes-кластері повнофункціональний web-додаток для знайомств, що інтегрується з Telegram у формі Mini App та доступний через стандартний браузер. Спроектowano архітектурні шаблони, які можуть бути використані у інших соціальних сервісах: схема шардування real-time-чатів між горизонтально-масштабованими екземплярами через gRPC

bidirectional streaming; система валідаційних rolling-хешів з розмежуванням ключів за категоріями операцій; конвеєр CI/CD з кешуванням BuildKit-станів у MinIO, що забезпечує прискорення повторних збірок у 15-25 разів; шаблон захищеного базового образу nginx розміром близько 600 КБ з механізмом env-injection – придатний для розгортання SPA-фронтендів у будь-яких бізнес-доменах.

Інформаційні системи соціальних сервісів, web-платформи знайомств і спільнот, корпоративні real-time-комунікації, Telegram Mini Apps, мобільні клієнт-серверні системи з high-load-сценаріями. Окремі архітектурні складові – наприклад, схема шардування чатів та інженерія 0-CVE-контейнерів – мають універсальну галузь застосування і можуть бути використані поза тематикою знайомств.

Основні положення роботи пройшли апробацію у вигляді доповідей на:

1. XV Всеукраїнській науково-практичній конференції «Програмна інженерія та веб-технології» (Київ, 2026) – з тезами «Мікросервісна архітектура веб-додатку для знайомств з використанням Go, TypeScript та gRPC» [27].
2. Всеукраїнській науково-практичній конференції «Програмні системи та технології цифрового розвитку» (Київ, 2026) – з тезами «Застосування gRPC та DragonflyDB у розподіленій мікросервісній системі для цифрових знайомств» [28].

Кваліфікаційна робота складається зі вступу, трьох розділів, висновків та переліку посилань. У першому розділі проведено аналіз предметної галузі, існуючих програмних продуктів та обрано стек технологій. У другому розділі розроблено архітектуру системи, модель даних, програмний інтерфейс і прототипи інтерфейсу користувача. У третьому розділі описано реалізацію всіх компонентів, заходи з контейнерної безпеки, конвеєр CI/CD, а також обґрунтовано стратегію та результати тестування. У висновках узагальнено основні наукові й практичні результати роботи.

Загальний обсяг роботи – 80 сторінок основного тексту, 33 рисунки, 28 джерел використаної літератури.

1 АНАЛІЗ ПРЕДМЕТНОЇ ГАЛУЗІ ТА ЗАСОБІВ РОЗРОБКИ

1.1 Опис задачі та предметної галузі

Онлайн-знайомства є специфічною підгалуззю соціальних мереж, у якій основною метою користувача є встановлення двостороннього персонального контакту з іншими користувачами на підставі заздалегідь зазначених критеріїв. На відміну від класичних соціальних мереж, у dating-сервісах ключовою операцією є не публікація контенту, а двосторонній взаємний інтерес – так званий «метч». Технологічно це означає, що системи цього класу повинні витримувати високий обсяг короткочасних транзакцій (свайпи, лайки, повідомлення) і забезпечувати сублінійну затримку при доставленні повідомлень тисячам одночасних користувачів.

З економічної точки зору галузь становить близько 11 млрд доларів США щорічного доходу станом на 2025 рік, із прогнозом подальшого щорічного зростання у середньому на 11% (рисунок 1.1). Понад 60% доходу формується підписками, ще близько 25% – мікротранзакціями (купівля додаткових «лайків», «бустів», «розблокувань»), решта – рекламою.

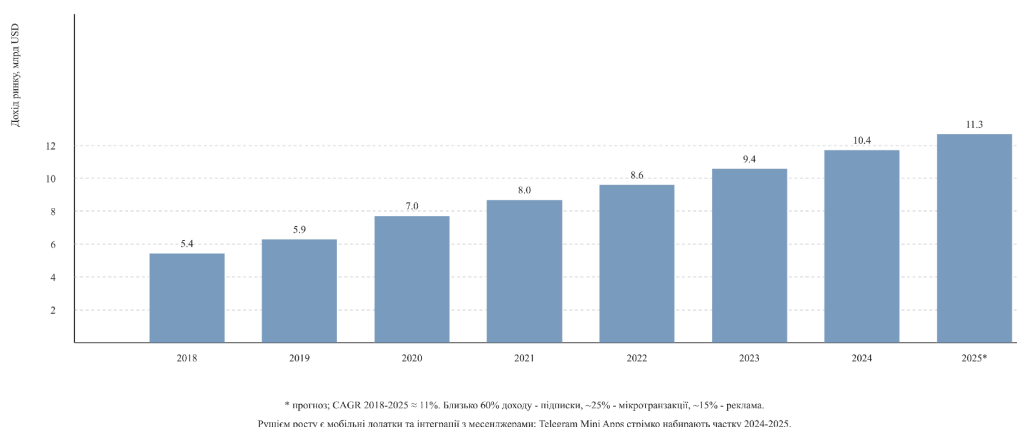


Рис. 1.1 Динаміка глобального ринку онлайн-знайомств (2018-2025, млрд доларів США)

Активна аудиторія складає близько 300 мільйонів унікальних користувачів на місяць по всьому світу; лівова частка з них – особи у віці 16-35 років. У 2024-2025 роках спостерігається стрімкий перерозподіл аудиторії на користь сервісів, інтегрованих у вже встановлені застосунки-месенджери: Telegram Mini Apps, WeChat Mini Programs, тощо. Цей зсув обумовлений тим, що користувач уникає необхідності встановлювати окремий застосунок та проходити процедуру реєстрації – авторизація відбувається через існуючий обліковий запис месенджера, а інтерфейс інтегрується у звичне середовище.

Цільову аудиторію розроблюваної системи становлять активні користувачі месенджера Telegram віком 16 років і старше. Telegram станом на 2025 рік налічує понад один мільярд щомісячних активних користувачів та активно інвестує у платформу Mini Apps. Демографічно цільова аудиторія розподіляється приблизно так: 16-24 роки – 41%, 25-34 роки – 28%, 35-44 роки – 19%, понад 45 років – 12%.

Аналіз поведінкових патернів цільової аудиторії дозволяє виокремити такі основні потреби.

Понад 60% потенційних користувачів залишають класичні dating-додатки на етапі реєстрації через надмірну кількість обов’язкових кроків. Telegram Mini App усуває цю проблему за рахунок автентифікації через існуючий обліковий запис месенджера.

Для більшості сценаріїв знайомств критично важлива географічна близькість потенційного партнера. Користувач очікує можливості точно налаштувати радіус пошуку і фільтрувати анкети як у межах міста, так і у радіусі заданої кількості кілометрів від поточної точки.

Поява нового повідомлення повинна доставлятись отримувачу впродовж менше секунди – затримка більше двох секунд призводить до виходу користувача з активного діалогу.

Користувачі більше довіряють моделі freemium, у якій базовий функціонал безкоштовний, а додаткові можливості (буст профілю, розблокування лайку) купуються разово або рекурентно за бажанням користувача.

Захист персональних даних – передусім захист від unauthorized scraping, а також гарантії, що дані користувачів не доступні стороннім.

Близько 30% користувачів цільової аудиторії регулярно беруть участь у тематичних спільних заходах (вечірки, спортивні зустрічі, кіноклуби); вбудована можливість створювати event'и та формувати групові чати навколо них суттєво розширює сферу застосування платформи.

Спираючись на аналіз потреб цільової аудиторії, було сформовано наступний перелік основних функціональних вимог. Повна діаграма варіантів використання наведена на рисунку 1.2.



Рис. 1.2 Діаграма варіантів використання системи (актори: Користувач, Адміністратор, Telegram)

Функціональні вимоги поділяються на кілька груп.

Авторизація та керування обліковим записом передбачає реєстрацію через підпис init-data Telegram Mini App або через email/пароль з бекап-каналу; автоматичне утворення профілів режиму dating та party при першому вході;

можливість перегляду балансу внутрішніх валют (hearts, boosts), історії транзакцій, реферальних посилань; самостійне видалення облікового запису з можливістю наступної реактивації; багатомовність (українська, англійська на момент здачі роботи).

Профіль користувача підтримує окремі профілі для режимів «dating» та «party»: 1-9 фото, аватар, опис до 1024 символів, інтереси (до 10), атрибути (зріст, вага, освіта, звички), особисті переваги (вікові межі, стать, орієнтація, радіус пошуку), часовий пояс, мова інтерфейсу. Передбачено автоматичне завантаження компресованих та заблюрованих прев'ю фото для відображення у профілі іншого користувача до моменту розблокування.

Геолокаційний пошук виконує просторовий запит за координатами користувача з підтримкою двох радіусів – у межах міста та поза ним; фільтрація за віком, статтю, підпискою; врахування boost'у – посилення, що тимчасово пересуває профіль на верх стрічки інших користувачів; пагінація через перелік уже переглянутих ID, щоб не показувати анкету повторно, поки результат взаємодії з нею пропагується кластером баз даних або взаємодія ще не була виконана.

Двосторонній лайк автоматично створює 1-to-1 чат; система валідаційних rolling-хешів унеможливорює лайк профілю, який не було зворотно «віддано» сервером – захист від enumeration та replay-атак (детальніше у п. 3.2); підрахунок «сили» лайку (суперлайк) – від 1 heart.

Чат у реальному часі – 1-to-1 діалог між користувачами з підтримкою тексту, фото, відео, голосових повідомлень; емодзі-реакції; відповіді на повідомлення; редагування та видалення; статуси «доставлено», «прочитано»; індикатор набору тексту; групові чати, що автоматично створюються для кожного party-event; системні повідомлення (UserJoined, UserLeft, UserKicked, ChatCreated, ChatDeleted тощо).

Party-events передбачають створення заходу з назвою, описом, часом проведення, локацією, максимальною кількістю учасників і обкладинкою; перелік медіа; пошук event'ів за географією та фільтрами; «лайк» на event автоматично

додає користувача до групового чату; кнопки «вийти» для учасника, «забанити учасника» для господаря event'у.

Внутрішні валюти та підписки реалізуються через hearts (придбання та витрати на лайки/розблокування), boosts (підвищення видимості профілю на N годин). Підписки трьох рівнів (Base, Lite, Elite) з пакетом переваг (необмежені лайки, заздалегідь видимі лайки від інших, зменшений ліміт чат-блокувань тощо). Передбачено інтеграцію з Telegram Stars / Telegram Invoice для безпечних платежів; webhook на стороні бота для оновлення балансу.

Завдання та реферальна система передбачає завдання з винагородою у hearts/днях підписки за виконання конкретних дій (підписатися на канал, запросити N друзів, заповнити профіль на 100%); унікальні start-параметри для реферальних посилань і докладний звіт про використання; нарахування реферру при першій реєстрації запрошеного.

Звіти та модерація: користувач може поскаржитися на профіль, повідомлення, event або групове повідомлення з вибором типу порушення (фейк, домагання, шахрайство, неприйнятна поведінка, неповнолітній, інше) і текстовим описом. Адміністратор переглядає чергу скарг, банить користувача, надсилає попередження або відхиляє скаргу.

Статистика та аналітика: усі системи генерують подій 4 типів – OneTime (перший вхід, джерело трафіку), PerEvent (клік, перегляд, відкриття модалки), PerSession (інформація про пристрій, мову, преміум-статус), MoneyOps (витрата та зарахування валюти). Події передаються разом із запитом через криптографічну ін'єкцію в запит на бекенд, звідки поточним gRPC до окремого сервісу аналітики, що пише їх у ClickHouse для подальшого OLAP-аналізу та побудови dashboard'ів.

Нефункціональні вимоги також визначають межі системи: продуктивність – API до 200 мс p95 для матчмейкінгу та до 100 мс p95 для інших HTTP-операцій, доставка повідомлення до 250 мс p95; масштабованість – горизонтальний scale-out сервісів і підтримка не менше 100 000 одночасних WebSocket-з'єднань; відмовостійкість – rolling-update без простою, liveness/readiness probes та запис у БД через MaxScale; безпека – 0 CVE в образах, non-root Pod'и, TLS 1.3, rolling

hash-валідація mutation-запитів, rate-limit, input validation і параметризовані SQL-запити; зручність розгортання – GitOps, env-injection та однаковий образ для dev/staging/prod; підтримуваність – 41 лінтер у CI, Swagger-документація, Clean Architecture (рисунк 2.2) та структуровані JSON-логи з request_id; локалізація – не менше чотирьох мов інтерфейсу; доступність – WCAG 2.1 AA для головних сценаріїв; регуляторні вимоги – GDPR-сумісне зберігання персональних даних і журналювання згод.

Система підтримує дві відносно незалежні бізнес-моделі. У режимі dating користувач шукає партнера у форматі один-на-один. Алгоритм пошуку базується на геолокаційній близькості, вікових і статевих преференціях. Кінцева мета – встановлення взаємного лайку та подальший приватний чат.

У режимі party користувач шукає одночасно (а) інших користувачів-учасників party-event'ів і (б) самі party-event'и для приєднання. Створювати event'и можуть як приватні особи, так і заклади-партнери та окремі організатори. Це дозволяє платформі обслуговувати ширший спектр сценаріїв соціальної взаємодії: від випадкової спільної прогулянки до тематичних вечірок, спортивних подій тощо.

Обидва режими використовують один обліковий запис, але мають окремі профілі – це дозволяє користувачу вибірково демонструвати ту чи іншу свою сторону відповідно до контексту. Внутрішня валюта (hearts, boosts) – спільна для обох режимів.

1.2 Аналіз існуючих програмних продуктів

З метою обґрунтування актуальності розробки було проведено детальний аналіз чотирьох найбільших гравців ринку онлайн-знайомств – Tinder, Bumble, Hinge, Badoo. Кожен із цих продуктів зайняв свою унікальну нішу, проте, як показав аналіз, жоден із них не покриває всього спектру потреб сучасного користувача, а особливо – українського користувача, що активно користується Telegram. Зведене порівняння наведено у п. 1.2.

Tinder – найбільший міжнародний dating-сервіс, що став прародичем «свайпової» парадигми. Архітектурно базується на хмарній інфраструктурі AWS з використанням Kubernetes (EKS) та власних мікросервісів. Має понад сто мільйонів користувачів, але орієнтований переважно на короткочасні знайомства, не підтримує party-events, не має нативної інтеграції з месенджерами і вимагає окремого застосунку.

Bumble позиціонує себе як «жіночий» dating-сервіс: першу ініціативу у чаті завжди має жінка. Архітектура поєднує великий РНР-моноліт з вибірково винесеними мікросервісами. Сильні сторони – продумана функція безпеки, режим BFF і Bizz, проте інтеграції з Telegram та самостійного створення party-event'ів користувачами немає.

Hinge робить ставку на докладні профілі та довші взаємодії: користувач відповідає на особисті prompts, що лягають в основу рекомендацій. Сильні сторони – продуманий онбординг і орієнтація на серйозні стосунки, але відсутні режим спільних заходів, Telegram-інтеграція та real-time чат на рівні WebSocket.

Badoo – один із найстарших dating-сервісів, що має значну користувацьку базу у країнах СНД та Латинській Америці. Він підтримує багато додаткових режимів і web-версію, проте інтерфейс наслідує дизайн середини 2010-х, Telegram у переліку способів реєстрації відсутній, а найближчий до party-events аналог – режим «Live» – фокусується на стрімах, а не на реальних спільних зустрічах.

Порівняння розроблюваної системи Match із найбільшими аналогами наведено у таблиці 1.1. Позначення: «+» – реалізовано, «-» – не реалізовано, «н/в» – інформація не оприлюднена.

Таблиця 1.1

Зведене порівняння Match із найбільшими аналогами

Критерій	Match	Tinder	Bumble	Hinge	Badoo
Інтеграція як Telegram Mini App	+	-	-	-	-
Окремий режим party-events	+	-	-	-	-
Нативний real-time chat (WebSocket)	+	+	+	-	+
gRPC у міжсервісній комунікації	+	частково	-	-	-
Геопошук на базі POINT/spatial index	+	+	+	+	+
Горизонтальне масштабування real-time-сервісу	+	+	+	+	+
0-CVE контейнерні образи	+	н/в	н/в	н/в	-
GitOps-розгортання FluxCD	+	н/в	н/в	н/в	-
Групові чати у форматі заходу	+	-	-	-	частково
Підтримка відеомедіа у чаті	+	+	+	+	+
Real-time аналітика (OLAP)	+	+	+	+	+
Веб-версія як рівноцінний канал	+	-	+	-	+
Підписки різних рівнів	+	+	+	+	+
Локалізація	+	+	+	+	+
Особистий бренд / репутація через рейтинг	+	-	-	-	+

Визначено, що Match покриває одночасно ті потреби користувачів, що інакше потребували б одночасного використання кількох різних сервісів: інтеграція з Telegram, повноцінний real-time-чат як у Tinder, party-events як ніша, не зайнята жодним великим гравцем, та сучасна DevOps-практика, що зустрічається у згаданих індустріальних рішеннях. Match закриває прогалину у поєднанні (1) інтеграції з Telegram у форматі Mini App, (2) спеціалізованої підтримки party-events, (3) сучасної інженерної практики GitOps та контейнерної безпеки нульового рівня – поєднанні, що відсутнє у будь-кого з аналогів. Це обґрунтовує актуальність розробки запропонованої системи: вона займає унікальну позицію в матриці «технологічна зрілість × нішевий функціонал», та з огляду на стрімке зростання сегмента Telegram Mini Apps має значний ринковий потенціал.

1.3 Огляд технологій, фреймворків та програмних засобів

Цей розділ систематизує технологічні рішення, обрані для побудови системи. Усі обрані компоненти об'єднано у п'ятишаровий стек: мови програмування, серверні фреймворки та бібліотеки, клієнтські технології, інфраструктура та засоби тестування і безпеки.

Go версії 1.26 обрано як основну мову серверних сервісів через комбінацію переваг, особливо цінних у мікросервісних системах: нативну конкурентність на основі goroutines та channels (M:N-планувальник дозволяє ефективно обслуговувати 100+ тисяч одночасних WebSocket-з'єднань на одному екземплярі); статичну строго типізовану семантику з простим синтаксисом; компіляцію в єдиний статичний бінарний файл, що уможливорює ущільнені контейнерні образи на базі scratch (детальніше у п. 3.5); вбудовану підтримку HTTP/2 та HTTP/3, gRPC та protobuf; швидкий час холодного старту (десятки мілісекунд), що принципово важливо для оркестраторів типу Kubernetes у сценаріях rolling-update. Зрілий стандартний pprof, go test, go vet дозволяють організувати

інструментування та профайлінг без сторонніх засобів. Поряд з цим, версія Go 1.26 принесла суттєві поліпшення в garbage collector – зменшення пауз STW до < 100 мкс, – що додатково підкреслює доцільність вибору.

TypeScript версії 5.x обрано як основну мову клієнтських додатків, оскільки строга типізація над JavaScript забезпечує суттєво кращу підтримуваність кодової бази та виявлення помилок на етапі компіляції. У режимі strict: true TypeScript примушує розробника явно обробляти null та undefined, що особливо актуально у real-time-системах, де частина даних може запізнюватись. TypeScript-компіляція добре інтегрується з системами білда Vite та Turbopack (Next.js 16), що зменшує час інкрементальної перебудови до часток секунди. Один із вирішальних факторів вибору – наявність повноцінних типобезпечних SDK для Telegram Mini Apps (@telegram-apps/sdk-react), Redux Toolkit, TanStack Query.

Як HTTP-фреймворк обрано Gin v1.11. Gin побудований на базі стандартного net/http, додаючи лише вискоєфективний роутер і ланцюжок middleware. Відсутність магії DI-контейнерів та рефлексії робить Gin передбачуваним і легким у налагодженні. Альтернативи – Echo, Fiber (на базі fasthttp). Echo має дещо менш зрілу екосистему; Fiber пропонує вищу пікову пропускну здатність, проте ціною несумісності з net/http стандартом, що ускладнює використання чисельних інструментів, middleware (rate-limiter, OpenTelemetry-інструментування тощо). Зрілість Gin та сумісність зі стандартом стали вирішальними чинниками. Перевага: простий, добре документований, має широку екосистему сторонніх middleware.

Для роботи з MariaDB обрано GORM v1.31 – найрозповсюдженішу ORM для Go. GORM підтримує автоматичну міграцію схеми, генерує параметризовані SQL-запити (захист від SQL-injection), має зручний механізм Preload для жадібного читання пов'язаних сутностей. Альтернативи – sqlx (ручний SQL з обгортанням у структури) та ent (code-generation ORM від Facebook). sqlx дає більший контроль над запитам, але змушує писати багато повторюваного коду; ent має дуже потужний механізм генерації, але вимагає постійного підтримування DSL-описів схеми. GORM займає золоту середину: продуктивний, простий,

підтримує необхідні нам розширення (включно з GIS-типом POINT та spatial index у MariaDB).

Для міжсервісної комунікації обрано gRPC v1.79 з кодуванням Protocol Buffers v3. gRPC надає чотири моделі взаємодії: unary RPC (REST-аналог), server-streaming, client-streaming, bidirectional streaming. Остання використовується для real-time-чату, де клієнт і сервер обмінюються повідомленнями асинхронно та за потребою. Кодування protobuf забезпечує:

1. Строгу типізацію контрактів між сервісами через .proto-файли, що автоматично генеруються у код мовами Go, TypeScript, Python.
2. Компактну бінарну репрезентацію з використанням varint для цілих чисел: число, що міститься у 7 бітах, займає 1 байт; до 14 бітів – 2 байти. Тоді як еквівалентна стрічка JSON {"id":1234567890} потребує 17 байт, protobuf-поле int64 у такому самому випадку – 6 байт.
3. HTTP/2-мультиплексування: один TCP-зв'язок переносить тисячі логічних gRPC-стрімів без додаткового handshake.

Альтернативами були REST/JSON (схема не строга, валідація на стороні розробника), GraphQL (гнучкіший, але складніший у моніторингу та гірший для streaming-сценаріїв), Cap'n Proto (вища швидкість серіалізації, проте бідніша екосистема). Зрілість gRPC та широка інструментальна підтримка стали ключовими чинниками вибору.

Socket.IO – широкоживаний фреймворк поверх WebSocket, що додає автоматичний reconnect, fallback на long-polling, namespace-rooms та структурований формат подій. Перевагою порівняно з «голим» WebSocket є автоматичне відновлення з'єднання після збою мережі та готова підтримка кімнат – що критично для адресного broadcast'у повідомлень у чаті. У ролі сервера Socket.IO виступає лише сервіс-проксі (MatchChatsProxyService); решта серверних компонентів спілкується через gRPC, що поєднує переваги обох технологій (масовість підключень + строга типізація).

В якості Redis-сумісного in-memory сховища обрано DragonflyDB. Він підтримує повний набір команд RESP3, тому клієнтський код Go

(github.com/redis/go-redis/v9) працює без жодних змін. Відмінність від класичного Redis – архітектура shared-nothing multi-threading: кожний потік обслуговує власне підмножину ключів, без глобального блокування, що в навантаженні SET/GET дає багатократно вищу пропускну здатність на одних і тих самих апаратних ресурсах. У системі Match Dragonfly використовується для кешу частих запитів до БД, sliding-window-лічильників rate-limit, presence-стану користувачів і короткоживучого стану підписок на чати. JWT не зберігається у DragonflyDB: токен підписується сервером і передається клієнту. Вибір саме Dragonfly, а не Redis, обґрунтований необхідністю обслуговувати десятки тисяч одночасних GET/SETEX-операцій без вертикального масштабування одного екземпляра.

В якості основної реляційної СУБД обрано MariaDB – повнофункціональний форк MySQL, що підтримує тип POINT із SRID 4326 та просторовий індекс на R-Tree, нативний JSON тип, INVISIBLE COLUMN, віртуальні колонки. Кластер розгорнуто у класичній конфігурації master + slave з асинхронною binlog-реплікацією: один primary-вузол приймає всі записи, репліки приймають read-only-навантаження. Маршрутизація запитів виконується через MariaDB MaxScale – проху, що автоматично розрізняє SELECT (балансує між репліками) і UPDATE/DELETE/INSERT (надсилає на primary), реалізуючи прозорий read/write splitting на рівні протоколу. Це забезпечує лінійне масштабування навантаження читання, мінімальну затримку запису (без розподіленого консенсусу) та автоматичне перемикання при відмові primary за рахунок MaxScale Monitor.

Для аналітичних запитів обрано ClickHouse – колонкову СУБД з агресивним стиском і вузьоспеціалізованою обробкою агрегацій. ClickHouse зберігає дані по колонках, що для запитів типу «SELECT page, count() FROM events WHERE event_name='profile_view' GROUP BY page» дозволяє читати лише задіяні колонки, прискорюючи запит у десятки разів. ClickHouse сприймає десятки мільйонів вставок на секунду на одному вузлі – що покриває потреби MatchStatisticsService. Альтернативи – PostgreSQL (рядкове сховище, не підходить для OLAP), Apache Druid (потужний, але складніший в експлуатації).

В якості об'єктного сховища обрано MinIO – повністю S3-сумісне self-hosted сховище. Усе медіа (фото, відео, поробки) зберігається у MinIO; клієнти отримують дозволені MatchBackend `file_id`, а backend визначає, чи віддати повний файл, чи blur-файл для заблокованого профілю. Перевага self-hosted-варіанту – відсутність vendor-lock-in та повний контроль над пропускнуою здатністю.

Усі контейнерні образи зберігаються у власному приватному OCI-реєстрі Harbor. Harbor підтримує політики доступу та сканування CVE через Trivy – все це інтегровано у конвеєр CI/CD.

Основний клієнтський додаток (Telegram Mini App) реалізовано на Next.js 16 – фреймворку поверх React, який забезпечує automatic code-splitting, hot module replacement у dev-режимі та експорт у статичний хостинг. React 19 принесла нативну підтримку Suspense, hooks `use()` для асинхронних даних, поліпшення транзитивних оновлень.

Глобальний стан керується Redux Toolkit (нащадок класичного Redux, що значно зменшує boilerplate). Серверний стан окремо керується TanStack Query – він автоматично кешує відповіді, виконує дедуплікацію паралельних запитів, фоново оновлює дані за politicians, мінімізує мережевий трафік. Поєднання Redux Toolkit для глобального UI-стану і TanStack Query для серверних даних дає чисту поділ відповідальностей.

Для real-time-чату на клієнті використовується socket.io-client v4. Він автоматично перезапускає з'єднання при втраті зв'язку, підтримує бінарний транспорт, працює як через WebSocket, так і через long-polling fallback (актуально, наприклад, для корпоративних мереж з блокуванням WS).

Як UI-набір обрано Chakra UI v3 – компонентну бібліотеку, що нативно підтримує тематизацію (зокрема – теми Telegram Mini App, що змінюються відповідно до налаштувань самого месенджера), доступність (a11y), адаптивну верстку.

Для офлайн-кешування використовується Dexie – обгортка над IndexedDB. Дозволяє зберігати локально чати, повідомлення, drafts, останній стан стрічки

матчмейкінгу. При відновленні з'єднання клієнт синхронізує тільки нові повідомлення (запит GET /messages?after_id=...), що економить трафік.

Бібліотека @telegram-apps/sdk-react v3.3 інтерпює Telegram Mini App API: ThemeParams, BackButton, MainButton, HapticFeedback, OpenLink. Дозволяє виконувати GPS-запит через Telegram, отримувати ім'я та аватар користувача без додаткового запиту.

Спеціалізовані бібліотеки Framer Motion, Embla Carousel, react-easy-crop, @dnd-kit відповідають за анімації (свайп карток у matchmaking, плавні переходи), карусель фото, кроп аватара з підтримкою зум, drag-and-drop переупорядкування медіа у профілі.

Адмін-панель збудована на стеку Vite + React 19 + shadcn/ui + TanStack Table. Адмін-панель як SPA не потребує SSR; Vite забезпечує блискавичний dev-server та продакшн-білд через Rollup. shadcn/ui надає набір не-залежних компонент, які копіюються до проекту і повністю керовані; TanStack Table – потужна таблиця з фільтрацією, сортуванням, пагінацією, групуванням; Recharts – для побудови дашбордів.

Усі сервіси розгортаються у Kubernetes-кластері з поділом за неймспейсами (match для застосунків, data для сховищ, monitoring для спостережності). Деплойменти описано декларативно у YAML-маніфестах, що зберігаються у Git-репозиторії. Архітектура розгортання детально подана у п. 2.1 та на рисунку 2.4.

Як контролер GitOps обрано FluxCD – він складається з кількох контролерів (source, kustomize, helm, image-reflector, image-automation), що забезпечують декларативне розгортання й автоматичне оновлення тегів образів. Альтернативою є ArgoCD, проте FluxCD має меншу ресурсомісткість.

CI/CD конвеєр базується на GitHub Actions; runners – самохост у тому ж Kubernetes-кластері (через Actions Runner Controller), що дає (а) високу потужність runners, (б) високу швидкість push'у образів та кешів (локальний мережний канал замість Internet), (в) відсутність обмежень безкоштовного тарифу GitHub.

Власна GitHub Action YaCodeDev/ActionMinIOCache@v0.1.11 зберігає BuildKit-кеш у self-hosted MinIO замість стандартного GitHub Cache, що було принципово важливим: GitHub Cache не має кешування між тегами версій, що виключає можливість його використання для пришвидшення збірки та обмежений 10 ГБ на репозиторій і має досить низьку швидкість на завантаження та вивантаження при роботі поза ранерами GitHub. Власна реалізація уможливила контроль над TTL та кількістю кешів, оптимізовану швидкість через локальну мережу. Технічна особливість BuildKit – кеш-точки `--mount=type=cache` не зберігаються між запусками runner'а; саме тому застосовується інструмент `buildkit-cache-dance`, який «танцює» з кешем: упаковує його у tar усередині image-стадії, дістає на хост, ActionMinIOCache завантажує/відновлює, а потім кеш «упаковується» назад у BuildKit-точки. Деталі – у п. 3.5 та на рисунку 3.10.

Усі образи перед публікацією проходять автоматичний скан Trivy у Harbor як приватному реєстрі. Для прийняття до production-теги вимагається результат 0 CVE.

Стандартний інструмент тестування Go – `go test`. Усі юніт- та integration-тести запускаються через `go test ./... -v -race`.

Бібліотека `testcontainers-go` дозволяє підіймати Docker-контейнери прямо з тестів – для інтеграційних тестів проти реальної MariaDB чи DragonflyDB замість моків. Гарантує, що SQL-запити поведуться однаково в production.

Метаінструмент `golangci-lint v2` об'єднує понад сто Go-ліній. У проекті активовано 41 ліній, включаючи: `errcheck` (всі помилки опрацьовано), `errorlint` (правильне використання `errors.Is/errors.As`), `exhaustive` (всі enum-кейси у switch), `nilnil` (відсутність двозначного (nil, nil) повернення), `bodyclose` (закриття `http.Response.Body`), `contextcheck` (передача контексту у горутини), `sqlclosecheck` (закриття `*sql.Rows`), `wrapcheck` (огортання помилок з контекстом), `protogetter` (правильний доступ до `protobuf`-полів), `govet` (38 вбудованих аналізаторів). Повний перелік обговорений у п. 3.6.

Vitest – швидкий тестовий фреймворк на базі Vite для frontend. Підтримує JSDOM-середовище, snapshot-тести, mocking ESM-модулів.

Авторизація – на базі JSON Web Token (HS256) через бібліотеку JWT v5.
Telegram-init-data – підпис, який Telegram передає у Mini App, перевіряється на стороні backend через бібліотеку telegram-mini-apps/init-data-golang.

Rolling hash для access-hash описано у п. 3.2.

Всгrypt використовується для зберігання паролів (для альтернативного шляху входу через email).

OWASP ZAP та Trivy використовуються у CI як автоматизовані сканери.

2 ПРОЕКТУВАННЯ СИСТЕМИ

2.1 Архітектура системи

Система Match реалізована як сукупність тринадцяти незалежних мікросервісів, що взаємодіють між собою через визначені програмні інтерфейси. Загальна схема архітектури наведена на рисунку 2.1.

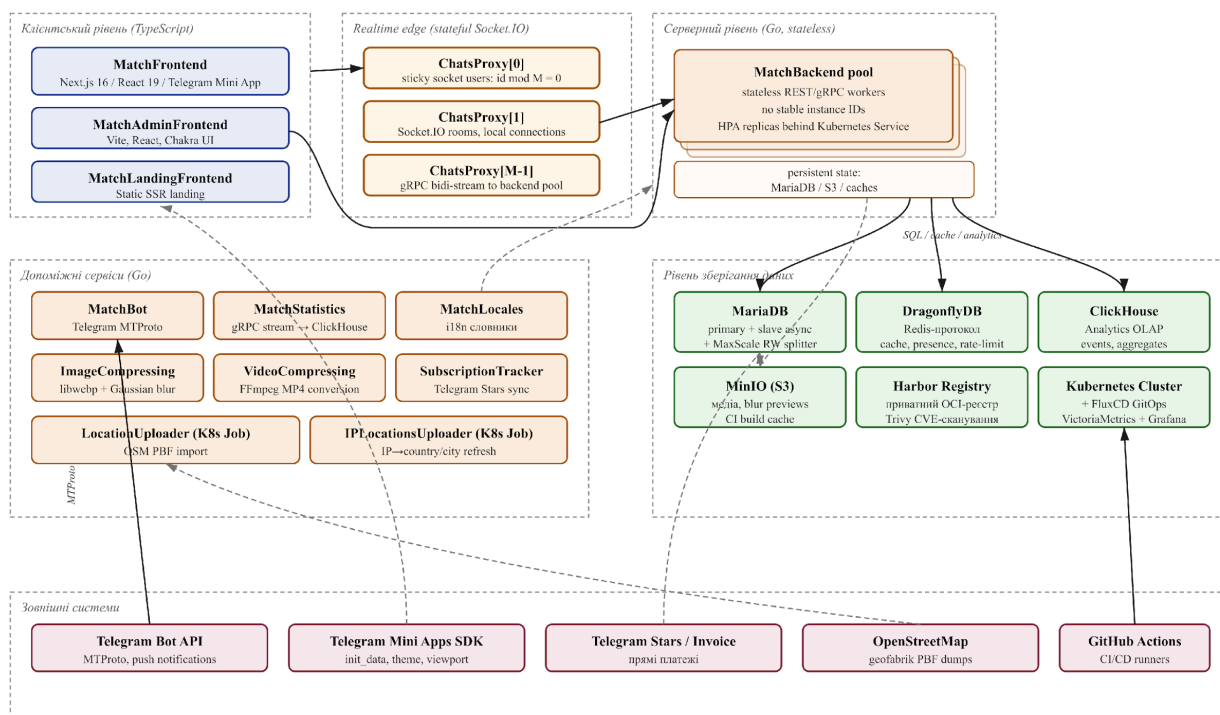


Рис. 2.1 Загальна мікросервісна архітектура системи Match

Архітектурно система ділиться на чотири зони відповідальності.

Клієнтський рівень утворюють три TypeScript-додатки, що обслуговують різні аудиторії: основний MatchFrontend як Telegram Mini App для звичайних

користувачів, MatchAdminFrontend як SPA-панель для адміністрування, MatchLandingFrontend як публічна сторінка для маркетингового онбордингу.

Серверний рівень доменних сервісів (Go 1.26) налічує шість основних серверних сервісів, що обслуговують різні домени системи. Центральний серед них – MatchBackend – реалізує REST-API та частину gRPC-API для всієї бізнес-логіки як stateless pool без стабільних номерів інстансів; репліки масштабує HPA за Kubernetes Service. Нумерованими є екземпляри MatchChatsProxyService: вони обслуговують sticky Socket.IO-з'єднання, локально тримають активні сокети користувачів і передають payload'и до backend pool через gRPC bidirectional streaming. MatchBot обслуговує інтеграцію з Telegram Bot API (платежі, нотифікації, deep-link). MatchStatisticsService обслуговує потокову аналітику і записує події у ClickHouse. MatchLocalesService повертає словники локалізації. MatchTelegramSubscriptionTrackerService синхронізує статуси підписок на Telegram канали для виконання партнерських завдань (мотитвований трафік).

Рівень допоміжних сервісів (Go 1.26) містить MatchImageCompressingService і MatchVideoCompressingService для стиснення та обробки медіа; MatchLocationUploader як Kubernetes Job вантажить дані OpenStreetMap у БД, відновлюючи пошкоджені фрагменти; MatchIPLocationsUploaderService оновлює БД відповідностей IP-діапазону до країни/міста.

Рівень даних утворюють MariaDB у конфігурації master + slave з MaxScale-проксі для транзакційних даних; DragonflyDB як кеш та Redis-сумісне сховище; ClickHouse для аналітики; MinIO як S3-сумісне сховище медіа; Harbor як OCI-реєстр приватних образів.

Система має п'ять зовнішніх інтеграцій: Telegram Bot API (для платежів та сповіщень) і Telegram Mini Apps SDK (на стороні клієнта) – обидва обслуговують комунікацію з месенджером; Telegram Stars / Telegram Invoice – для прямих платежів; OpenStreetMap (geofabrik) – для періодичного завантаження геоданих; GitHub Actions – для CI/CD-конвеєрів.

Усі Go-сервіси наслідують єдиний шаблон Clean Architecture (Robert C. Martin), адаптований до особливостей мови Go. На рисунку 2.2 наведено схему шарів та правило залежностей.

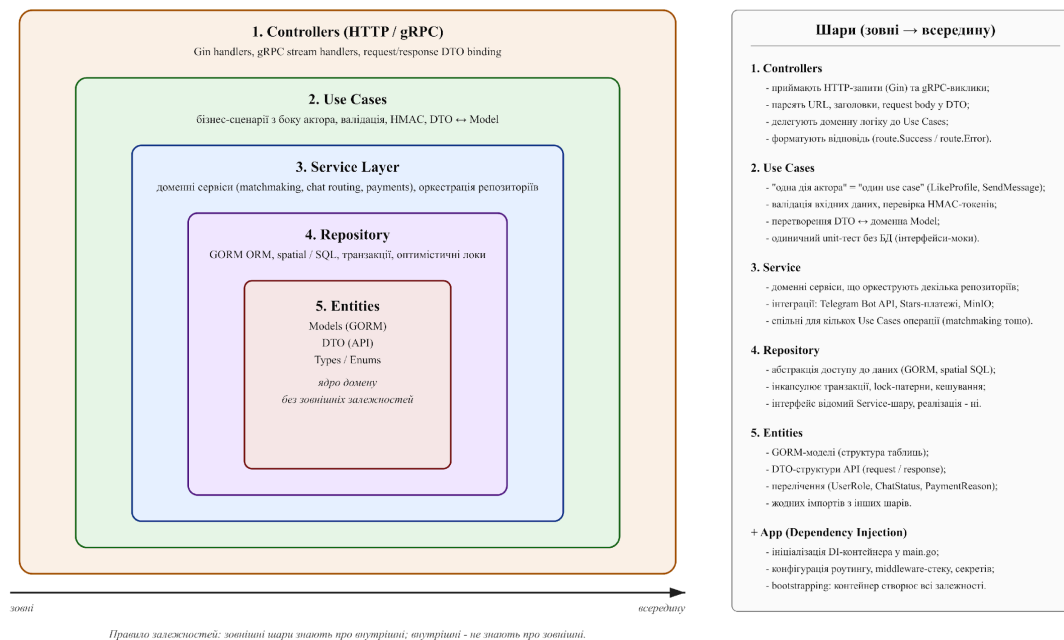


Рис. 2.2 Чиста архітектура (Clean Architecture) сервісу MatchBackend

Шари описано від зовнішнього до внутрішнього.

Шар Controllers (internal/controllers/) утримує обробники HTTP-запитів (Gin) та gRPC-запитів. Завдання – десеріалізувати вхідні дані, виконати валідацію формату, викликати відповідну операцію з шару Use Cases, серіалізувати результат у формат відповіді. Жодної бізнес-логіки тут немає.

Шар Use Cases (internal/usecases/) реалізує бізнес-операції з рівня одного актора (зазвичай – користувача). Шар викликає сервіси нижчого рівня, виконує перетворення між DTO та Models, реалізує генерацію та валідацію rolling-хешів, перевіряє передумови (наприклад, що користувачу вистачає hearts на лайк), komponує сценарій. Use Case не звертається безпосередньо до БД.

Шар Service (internal/service/) містить доменні сервіси, що оркеструють декілька репозиторіїв та інкапсулюють специфічну логіку домену (наприклад, matchmaking, like processing, chat routing, push-notification). Якщо Use Case описує

сценарій з точки зору актора, Service описує доменну операцію, яка може бути спільно використана кількома Use Cases.

Шар Repository (pkg/database/) є абстракцією над БД через GORM. Кожна доменна сутність має власний repository з визначеним інтерфейсом (UserRepo, DatingProfileRepo, ...). Це дозволяє замінити репозиторій моком у тестах.

Шар Entities (internal/entities/) включає незалежні структури даних – Models (зіставлення на таблиці БД), DTO (контракти API), Types (доменні типи), Constants (enum-переліки). Не залежать від інфраструктури.

Шар App / Dependency Injection (internal/app/) виконує композицію всього перерахованого. Тут створюються та зв'язуються конкретні реалізації репозиторіїв, сервісів, use cases, контролерів. У вхідній точці main.go достатньо викликати app.Run(), а вся ініціалізація лінійно виконується у пакеті app.

Правило залежностей (Dependency Rule): зовнішні шари можуть залежати від внутрішніх, але не навпаки. Завдяки цьому шар Use Cases чи Entities залишається повністю переносним і легко тестується.

Real-time-система чату – найвимогливіший до латентності компонент Match. Вона побудована на шаблоні Edge-proxy + worker-pool, описаному на рисунку 2.3.

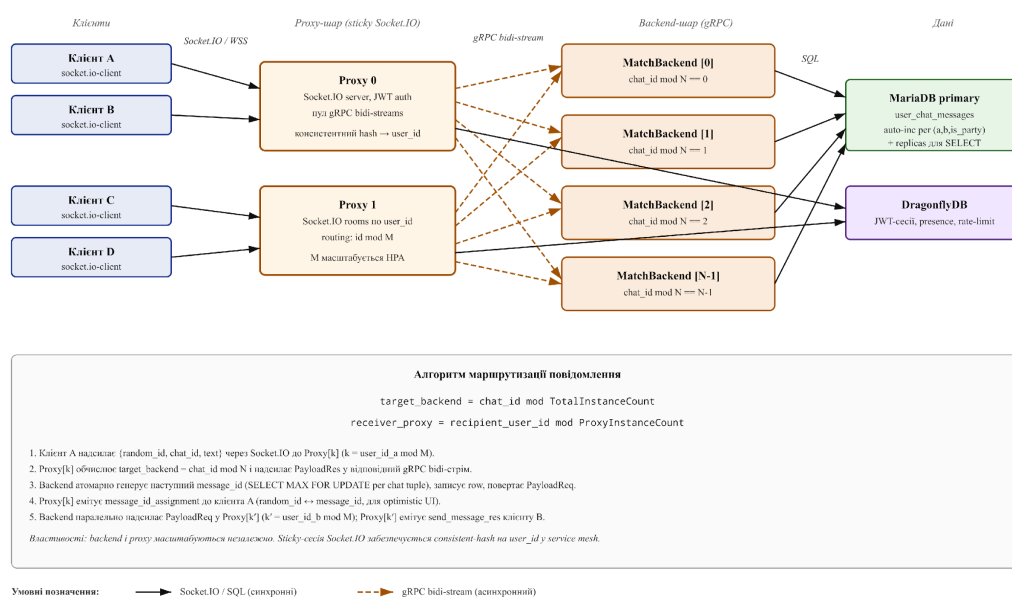


Рис. 2.3 Архітектура шардування чатів (Socket.IO та gRPC)

Алгоритм роботи такий. Клієнт підключається до сервісу MatchChatsProxyService через Socket.IO (WSS). NgINX ingress роутить запит до правильного інстансу проксі за формулою роутингу. При підключенні проксі автентифікує клієнта (JWT-токен Bearer), завантажує його user_id та зберігає його локально.

Формула роутингу обчислюється як $\text{target_proxy} = \text{chat_id} \bmod \text{TotalInstanceCount}$.

Маршрут відкривається через раніше створений gRPC bidirectional stream до екземпляра MatchChatsProxyService за DNS-адресою match-chats-proxy-service-{target_backend}.match-chats-proxy-service.match.svc.cluster.local:50051. Headless-сервіс Kubernetes видає окремий A-запис DNS для кожного екземпляра StatefulSet, що дозволяє точно адресувати конкретний match-chats-proxy-service-2.

Коли клієнт надсилає подію send_message з полями {random_id, chat_id, text, access_hash}, бекенд обирається по принципу round robin scheduling, а ціль маршрутизації до отримувача – за формулою роутингу.

MatchBackend отримує PayloadRes, валідує access_hash через rolling hash-перевірку (детально у п. 3.2), виконує INSERT повідомлення з блокуванням рядка для генерації послідовного id у межах (user_a_id, user_b_id, is_party)-кортежу, і повертає PayloadReq з присвоєним ідентифікатором.

Проксі отримує PayloadReq і виконує дві broadcast-операції. Відправнику А проксі емітує подію message_id_assignment з полями {random_id, message_id} для підміни оптимістичного повідомлення на остаточне. Отримувачу В проксі емітує подію send_message з повним payload'ом.

Така схема забезпечує дві основні властивості.

Горизонтальне масштабування дозволяє збільшувати кількість як backend-екземплярів (через шардування chat_id), так і проху-екземплярів (через шардування user_id) – незалежно.

Низька латентність досягається тим, що gRPC поверх HTTP/2 multiplexing'у не вимагає окремого TCP-handshake'у на кожне повідомлення; varint-кодування

протобуфу дає компактний формат payload'у; goroutine-моделі на Go дозволяють обслуговувати ~100 000 одночасних з'єднань на одному екземплярі проксі.

Кодування Protocol Buffers v3 додатково зменшує мережний трафік. Для прикладу: одне поле `chat_id` типу `uint64` зі значенням 1234567890 у `protobuf` займає 5 байт (`varint`-кодування), тоді як аналогічне поле у JSON "`chat_id`":1234567890 – 20 байт; виграш у 4 рази при тому ж семантичному навантаженні.

Розгортання усіх сервісів виконується декларативно у Kubernetes-кластері за GitOps-моделлю на базі FluxCD. Загальна схема – на рисунку 2.4.

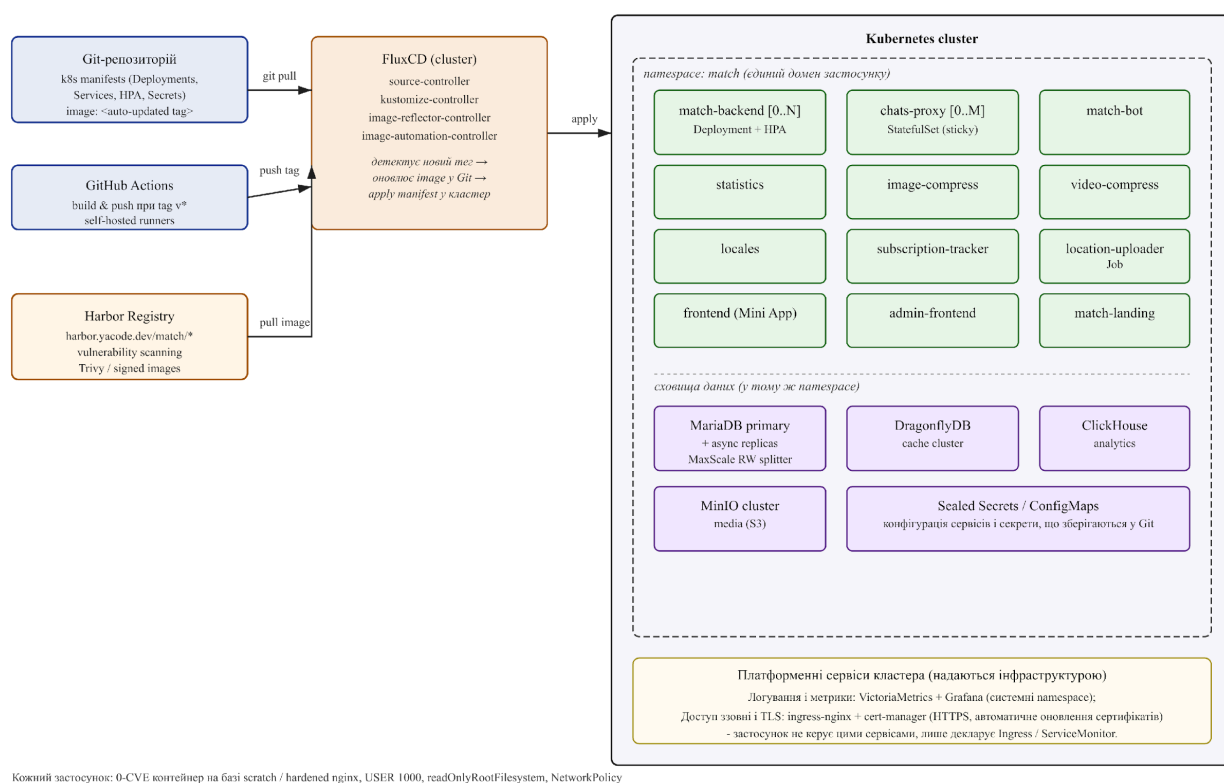


Рис. 2.4 Розгортання у Kubernetes-кластері з GitOps (FluxCD)

Принцип GitOps полягає в тому, що єдине джерело правди про стан кластера – це Git-репозиторій з Kubernetes-маніфестами. Будь-яка зміна виконується через PR і автоматично застосовується контролером, що періодично порівнює бажаний стан (Git) з фактичним (cluster) та виправляє розбіжності. Це дає прозорий `audit-trail` (хто, коли, що змінив), легкий відкат (`git revert`), повне відтворення кластера з нуля у разі катастрофи.

FluxCD складається з кількох контролерів.

Контролер `source-controller` слідкує за Git-репозиторієм і Helm-репозиторіями, кешує маніфести у внутрішньому storage.

Контролер `kustomize-controller` застосовує `kustomization.yaml`-файли до кластера, виконуючи `kustomize build`.

Контролер `image-reflector-controller` періодично сканує OCI-реєстр (Harbor) на предмет нових тегів образів, що відповідають заданим SemVer-політикам.

Контролер `image-automation-controller` при появі нового тегу автоматично оновлює відповідний image: ... у маніфесті у Git-репозиторії і робить commit від імені `flux-bot`.

Іншими словами, після того як CI-конвеєр публікує новий тег `v1.2.3` у Harbor, FluxCD автоматично детектує його, оновлює Deployment у Git, цей оновлений маніфест застосовується кустомізаційним контролером, і Kubernetes виконує rolling-update: створює новий Pod, перевіряє readiness, гасить старий. Час від push'у git tag'а до повного rolling-update'у в production – близько 6 хвилин при холодному кеші CI та 90 секунд при теплому (рисунок 2.11).

Особливості розгортання такі.

Єдиний неймспейс `match` містить усі застосунки, БД та сховища системи; кластер додатково надає платформенні сервіси спостереження (VictoriaMetrics + Grafana) та керування трафіком (ingress-nginx, cert-manager) у власних системних неймспейсах поза доменом застосунку.

MariaDB у конфігурації `master + slave` використовує асинхронну binlog-реплікацію за headless-сервісом; MaxScale у Deployment з кількома репліками виконує прозорий read/write splitting та автоматичний failover при відмові primary-вузла.

MatchBackend розгорнуто як Deployment з HPA: scale 2..10 за CPU/RAM-метриками; MatchChatsProxyService як StatefulSet (для sticky-sessions Socket.IO) з 2..6 репліками.

MatchLocationUploader виконується як Kubernetes Job: його запускають вручну або по cron, init-контейнер завантажує OSM-файл OpenStreetMap

(.osm.pbf), main-контейнер парсить і записує у MariaDB. Усе оброблення виконується в RAM через emptyDir: { medium: Memory } (tmpfs), щоб уникнути дискових I/O.

Секрети управляються через Sealed Secrets (Bitnami) – вони шифруються відкритим ключем кластера і безпечно зберігаються у Git.

NetworkPolicy за замовчуванням забороняє будь-який трафік; явно дозволяється лише потрібний (наприклад, MatchBackend до MariaDB через MaxScale).

PodSecurityPolicy та PSS «restricted» забезпечують виконання усіх Pod'ів як non-root, з readOnlyRootFilesystem: true, з drop усіх capabilities, з seccompProfile: RuntimeDefault.

Конвеєр CI/CD реалізовано трьома типами workflow-файлів GitHub Actions.

Перший тип – Build & Push для Go-сервісів (рисунок 2.5). Запускається на push тегу формату v*. Спочатку відновлюється BuildKit-кеш з MinIO через власну Action YaCodeDev/ActionMinIOCache, потім інструмент buildkit-cache-dance розпаковує кеш у потрібні точки --mount=type=cache. Виконується docker build за multi-stage Dockerfile (від golang:1.26-alpine до scratch) з CGO_ENABLED=0, -trimpath, -buildvcs=false. Образ сканується Trivy на CVE та публікується у Harbor.

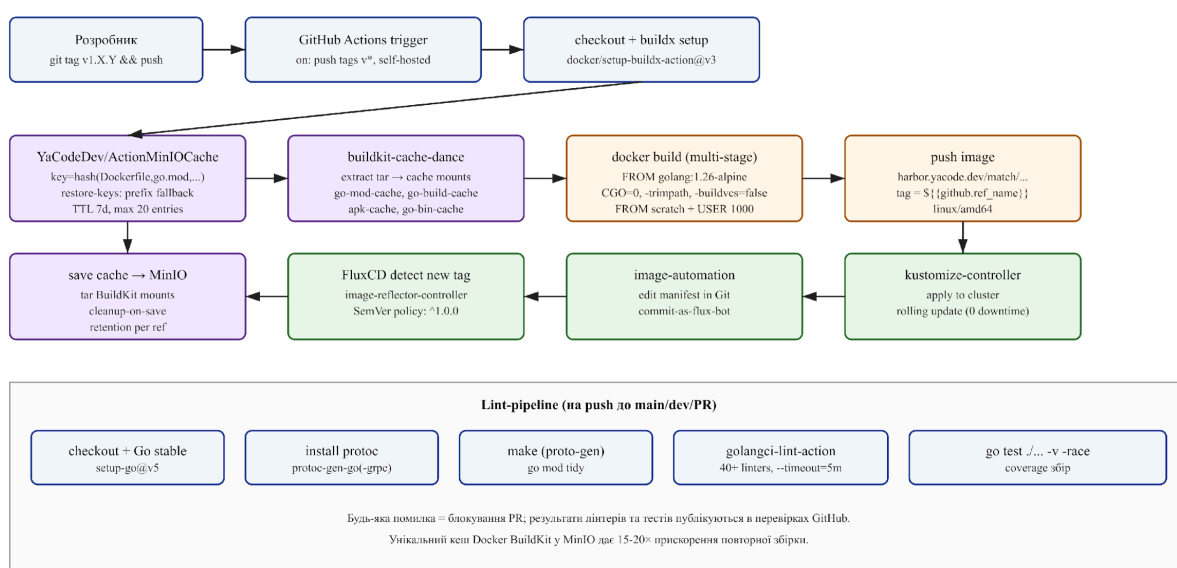


Рис. 2.5 Конвеєр CI/CD для Go-сервісів (з кешуванням у MinIO)

Другий тип – Build & Push для frontend-сервісів (рисунок 2.6). Аналогічна структура, проте multi-stage Dockerfile використовує oven/bun:alpine для збирання та власний захищений базовий образ harbor.yacode.dev/yacodedev-internal/frontend-base-slim:latest (~600 КБ) у полі runtime. Опис образу – у п. 3.5 та на рисунку 3.9. Кешування layers – через registry cache backend BuildKit. Секрети для приватного npm-реєстру передаються через `--mount=type=secret`.

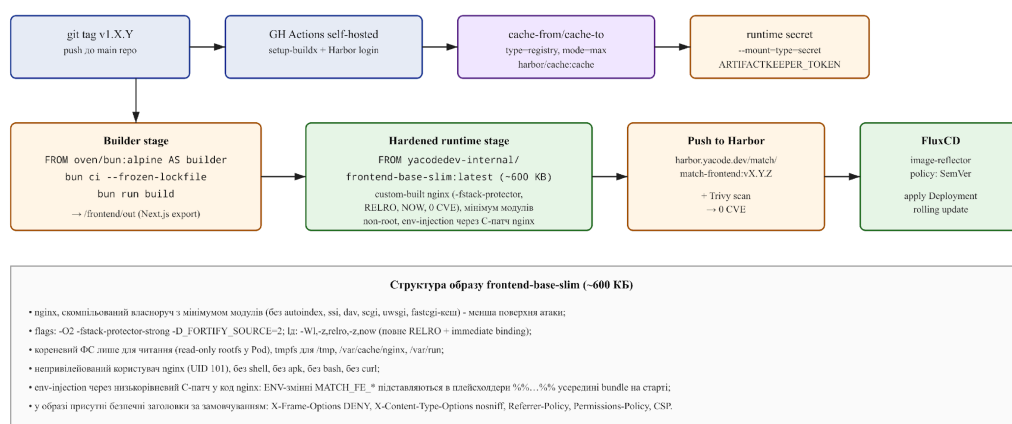


Рис. 2.6 Конвеєр CI/CD для frontend-сервісів

Третій тип – Lint & Test – запускається на push до main/master/dev/develop та на PR. Виконує make (генерація proto), go mod tidy, golangci-lint-action@v9 з 41 лінтером, go test ./... -v -race. Та відповідно bun run lint і bun run test для frontend. Усі результати публікуються як required checks GitHub. PR не може бути merged, доки всі checks не зелені.

Метрики ефективності кешування MinIO-cache:

- Холодна збірка (порожній кеш) – 4-6 хвилин.
- Тепла збірка (незмінні залежності) – 12-20 секунд.
- Середній cache hit rate – 92%.

Це означає, що на одному сервісі економиться ~85% хвилин CI порівняно з варіантом без кешування. Деталі – у п. 3.5.

2.2 Моделювання вимог

Повна діаграма варіантів використання (Use Case Diagram) системи побудована за нотацією UML 2.5 і наведена на рисунку 1.2 розділу 1. На ній показані три актори – «Користувач», «Адміністратор», «Telegram» – та 28 варіантів використання, що покривають весь функціонал системи.

«Користувач» взаємодіє з 21 варіантом використання, серед яких авторизація через Telegram, налаштування профілю, завантаження медіа, геолокаційний пошук, перегляд анкет, оцінка анкети, розблокування профілю, перегляд матчів, обмін повідомленнями, реакції на повідомлення, скарга на профіль, виконання завдань, реферальне запрошення, створення party-event, приєднання до event, груповий чат event, запрошення учасників, купівля підписки, поповнення hearts, буст профілю, налаштування фільтрів. «Адміністратор» взаємодіє з 5 варіантами: перегляд статистики, модерація скарг, управління реферралами, бан користувача, створення завдань. «Telegram» – з 3 варіантами: валідація `init_data`, обробка платежу через Stars/Invoice, push-сповіщення.

Виділено п'ять ключових сценаріїв, що ілюструють внутрішню роботу системи. Кожен сценарій описано діаграмою послідовностей UML.

Сценарій 1. Реєстрація користувача через Telegram Mini App наведена на рисунку 2.7. Mini App надсилає `POST /sign_up` з підписаним `init_data`. Backend через бібліотеку `telegram-mini-apps/init-data-golang` верифікує HMAC-підпис `init_data` за токеном бота, витягує `telegram_id`, `telegram_username`, `language_code`. Перевіряє наявність такого користувача у БД; якщо немає – створює запис у `users`, разом з пов'язаними `dating_user_profiles`, `party_user_profiles` (одна транзакція). Якщо у параметрі присутній реферальний код – підтверджується наявність такого реферера і у поле `invited_by_link` пишеться його `id`. Генерується JWT (HS256) з `claims {user_id, role, status, exp}`. Backend повертає `{token, user_id, role, status}`. Клієнт зберігає токен, ініціалізує Redux-стан і виконує `GET /me` за повним профілем.

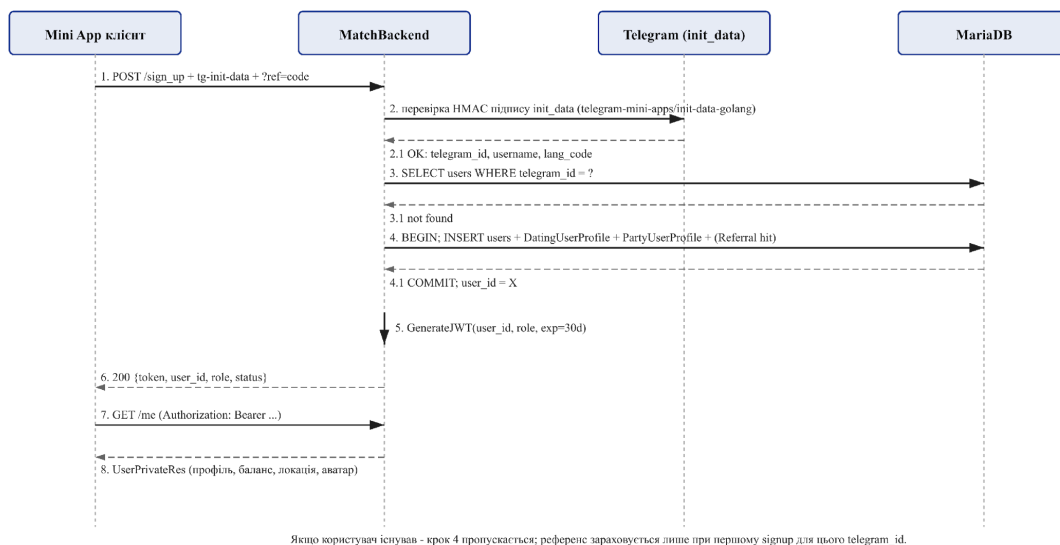


Рис. 2.7 Послідовність реєстрації користувача через Telegram Mini App

Сценарій 2. Геопошук та лайк наведено на рисунку 2.8. Клієнт надсилає GET /match_dating з параметрами min_age, max_age, searched_sex, desired_radius, location_id (або координатами). Backend обчислює просторовий запит з використанням spatial-index R-Tree (детальніше у п. 2.3). Для кожного знайденого профілю генерується validation_hash = HASH(time_bucket || user_id || profile_id || K_like) і повертається у DTO. Клієнт відображає стрічку. Коли користувач свайпає праворуч (лайк), надсилається POST /like_dating {profile_id, validation_hash, is_liked, power}. Backend перевіряє validation_hash, повторно обчислюючи його для поточного time bucket і попередніх bucket'ів у межах window_count. Списує power × hearts_per_like з балансу. Записує dating_user_profile_likes. Перевіряє наявність зворотного лайку від profile_id до поточного користувача – якщо так, створюється user_chats (з нормалізованим user_a_id < user_b_id). У разі mutual-match – push-сповіщення через MatchBot. Повертається {is_locked, hearts_left, match: {chat_id?}}.

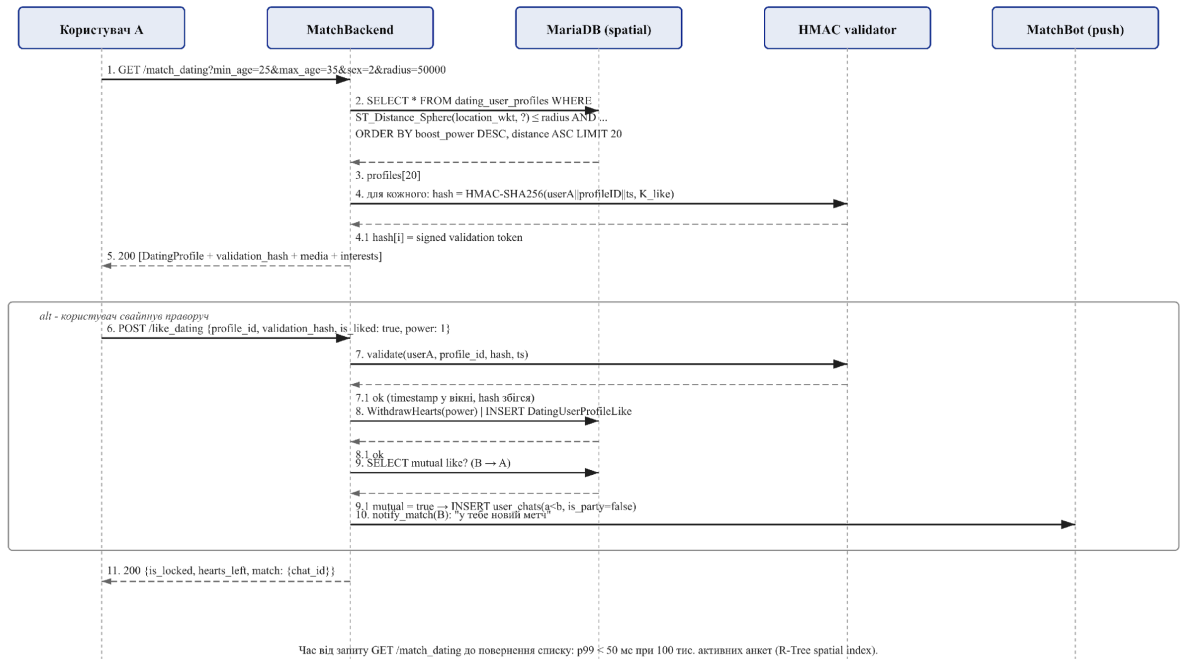


Рис. 2.8 Послідовність геопошуку та оцінки анкети

Сценарій 3. Надсилення повідомлення в реальному часі наведено на рисунку 2.9 та описано у п. 2.1, а також повторно деталізовано у sequence-діаграмі.

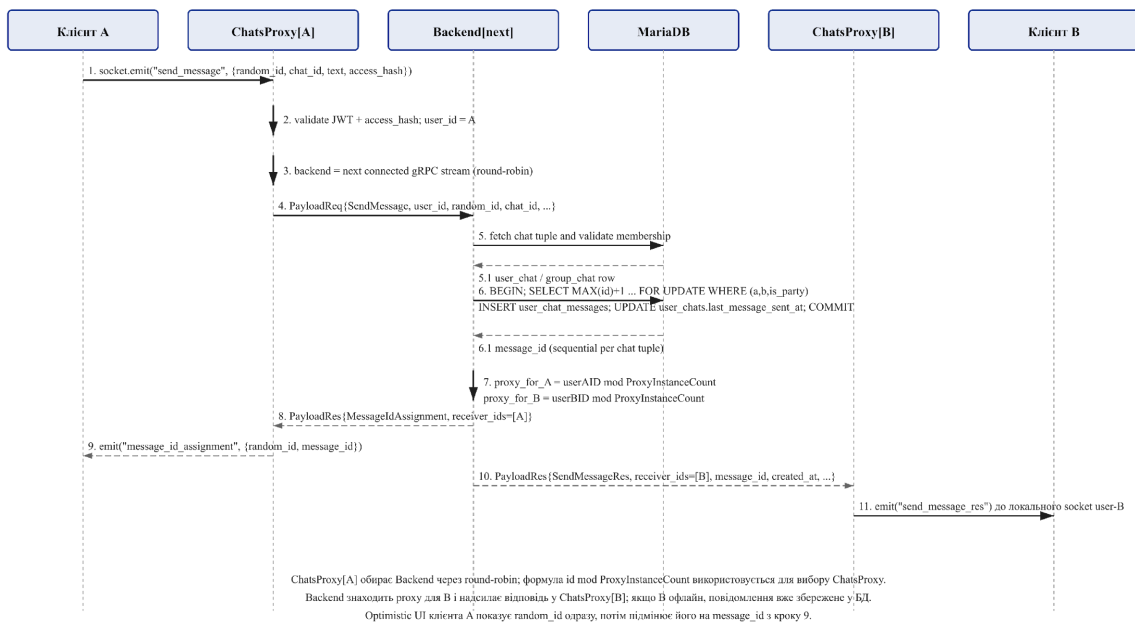


Рис. 2.9 Послідовність надсилення повідомлення (Socket.IO + gRPC)

Сценарій 4. Створення party-event та групового чату наведено на рисунку 2.10. Користувач А надсилає POST /profile_party_event {name, description, time, location_id, max_participants, avatar, media[]}. Backend у єдиній транзакції створює party_event_profiles та пов'язаний group_chats (поле event_id). Повертає {event_id, group_chat_id}. Інший користувач В виконує GET /match_party_event, отримує перелік event-карток; обирає event і виконує POST /like_party_entity {event_id, validation_hash, is_liked: true}. Backend записує party_event_profile_likes зі статусом is_accepted=false; кількість учасників і таблиця group_chat_members на цьому етапі не змінюються. Господар А окремо підтверджує заявку через POST /like_party_member {event_id, profile_id: B}. Після цього backend позначає заявку як прийняту, додає В до group_chat_members, збільшує members_count і надсилає системне повідомлення MessageEvent{event_type: UserJoined, user_id: B} через gRPC до MatchChatsProxyService, який broadcast'ить його активним членам групи.

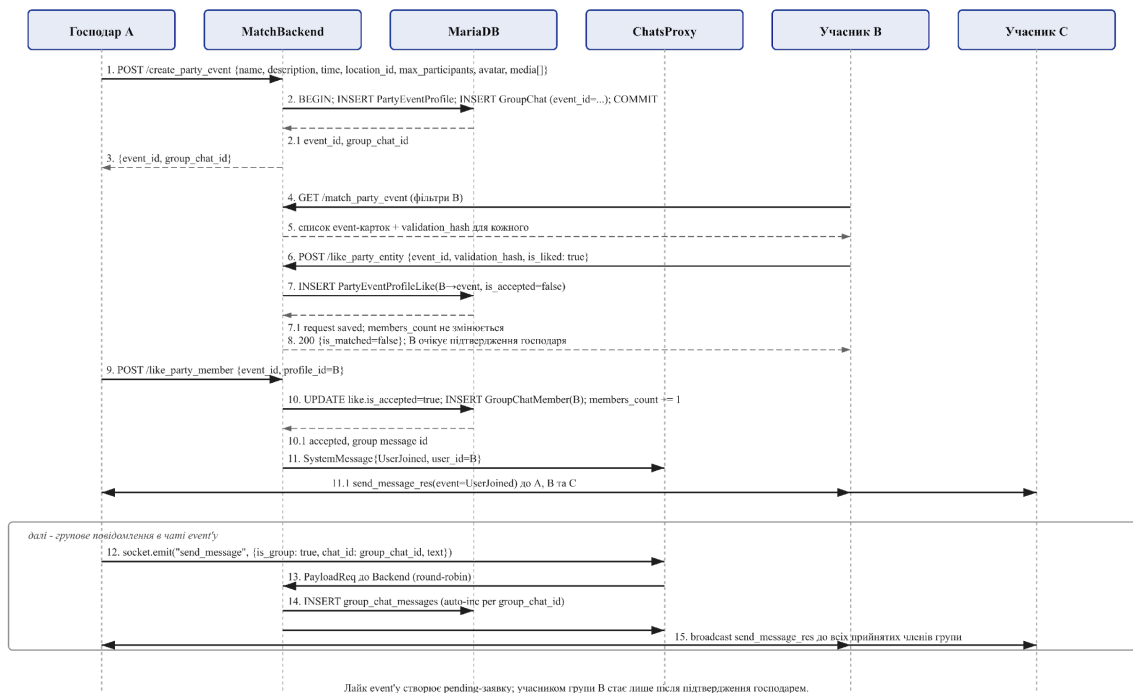


Рис. 2.10 Створення party-event та групового чату

Сценарій 5. CI/CD з GitOps-розгортанням наведено на рисунку 2.11. Розробник створює тег `git tag v1.2.3 && git push origin v1.2.3`. GitHub Actions, що триггерується `on push tags v*`, відновлює BuildKit-кеш з MinIO, виконує `buildkit-cache-dance`, збирає Docker-образ за multi-stage Dockerfile (scratch), перевіряє його Trivy-скануванням з очікуваним результатом 0 CVE та публікує у Harbor. FluxCD, що періодично перевіряє Harbor, детектує новий тег, що відповідає SemVer-політиці; `image-automation-controller` робить commit у Git-репозиторій з новим `image: ...`-полем у Deployment-маніфесті; `kustomize-controller` застосовує оновлений маніфест до кластера; Kubernetes виконує `rolling-update` без простою. Повний шлях від `push` тегу до завершеного `rolling-update` займає приблизно 6 хвилин при холодному кеші та 90 секунд при теплому.

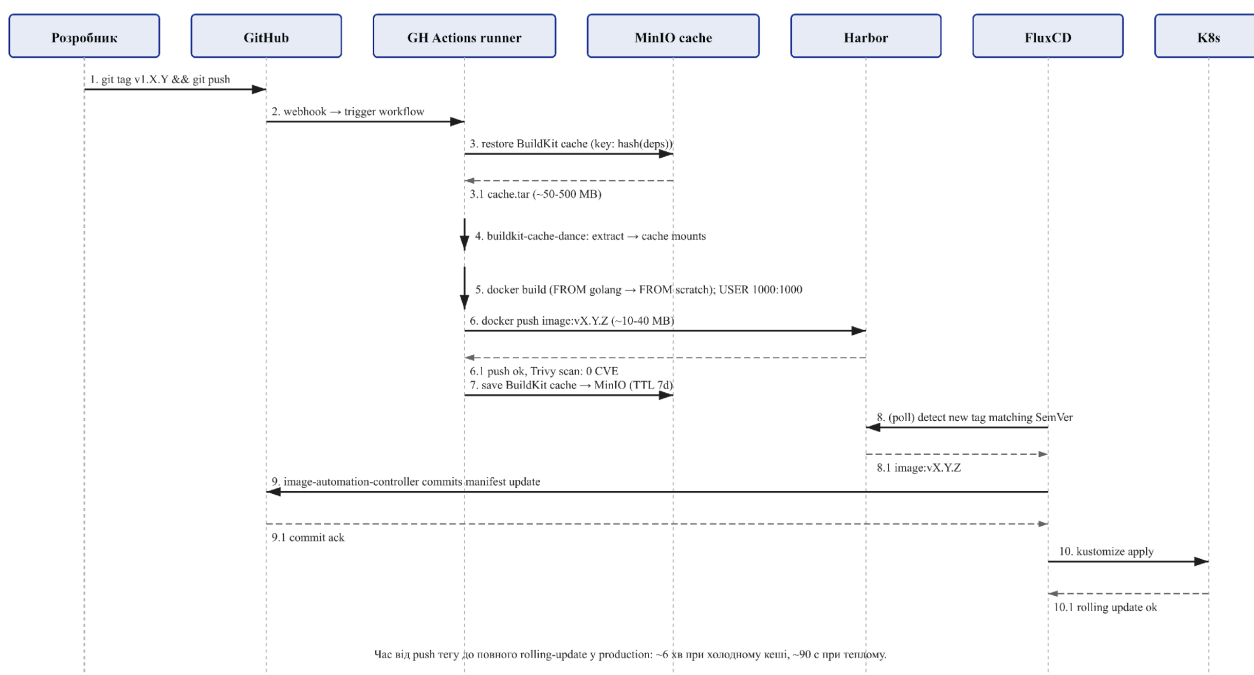


Рис. 2.11 Послідовність CI/CD з GitOps deployment

Нижче подано зведений перелік функціональних вимог системи. Для кожної вимоги вказано ідентифікатор, формулювання, пріоритет за схемою MSC (M – must, S – should, C – could), сервіс-носії та критерій прийняття.

FR-1. Реєстрація через Telegram `init_data`. Пріоритет: M; сервіс: MatchBackend; критерій: валідний `init_data` створює `user` і `JWT`; `/me` повертає профіль.

FR-2. Реєстрація / логін за `email+password`. Пріоритет: S; сервіс: MatchBackend; критерій: правильний пароль повертає `JWT`; неправильний – 401.

FR-3. Створення/редагування `dating`-профілю. Пріоритет: M; сервіс: MatchBackend; критерій: профіль зберігається; ліміти фото й опису валідуються.

FR-4. Створення/редагування `party`-профілю. Пріоритет: M; сервіс: MatchBackend; критерій: `party`-профіль змінюється незалежно від `dating`-профілю.

FR-5. Геолокаційний пошук `dating`. Пріоритет: M; сервіс: MatchBackend; критерій: відповідь містить тільки анкети в заданих фільтрах і радіусі.

FR-6. Геолокаційний пошук `party-events`. Пріоритет: M; сервіс: MatchBackend; критерій: відповідь містить тільки `event`'и в заданих фільтрах і радіусі.

FR-7. Лайк профілю з `rolling hash`-валідацією. Пріоритет: M; сервіс: MatchBackend; критерій: валідний `hash` записує лайк; невалідний/прострочений відхиляється.

FR-8. `Mutual match` – автоматичний чат. Пріоритет: M; сервіс: MatchBackend; критерій: взаємний лайк ідемпотентно створює один чат для пари.

FR-9. 1-to-1 чат у `real-time` (`WebSocket`). Пріоритет: M; сервіс: ChatsProxy + Backend; критерій: `send_message` повертає `message_id_assignment` і доставляє `peer`'у.

FR-10. Груповий чат `party-event`. Пріоритет: M; сервіс: ChatsProxy + Backend; критерій: лайк `event`'у створює заявку; `member` додається лише після асепт.

FR-11. Реакції на повідомлення. Пріоритет: S; сервіс: Backend; критерій: `set/unset` реакції зберігається і `broadcast`'иться учасникам.

FR-12. Завантаження медіа (фото, відео). Пріоритет: M; сервіс: Backend; критерій: валідний файл отримує `media_id`; занадто великий/невідомий тип відхиляється.

FR-13. Стиснення медіа. Пріоритет: M; сервіс: ImageCompressionService / VideoCompressionService; критерій: фото/відео конвертуються у цільовий формат.

FR-14. Blur-прев'ю заблокованих профілів. Пріоритет: M; сервіс: ImageCompressionService; критерій: locked-профіль повертає blur-preview замість оригіналу.

FR-15. Buy hearts/boosts через Telegram Stars. Пріоритет: M; сервіс: Backend + MatchBot; критерій: валідний webhook переводить транзакцію в done і нараховує баланс.

FR-16. Підписка Base/Lite/Elite. Пріоритет: M; сервіс: Backend; критерій: покупка оновлює subscription_type, level, valid_until.

FR-17. Реферальна система з кодами. Пріоритет: S; сервіс: Backend; критерій: код застосовується один раз під час signup і прив'язує реферера.

FR-18. Завдання з винагородами. Пріоритет: S; сервіс: Backend; критерій: виконане завдання можна claim'ити один раз; повторний claim відхиляється.

FR-19. Скарги на профілі/повідомлення. Пріоритет: M; сервіс: Backend; критерій: скарга з валідним типом/причиною створює запис для модерації.

FR-20. Адмін-панель: модерація скарг. Пріоритет: M; сервіс: AdminFrontend + Backend; критерій: admin може переглянути й закрити скаргу; user не має доступу.

FR-21. Адмін-панель: статистика по системі. Пріоритет: M; сервіс: Grafana + ClickHouse; критерій: dashboard показує події, реєстрації та фінансові дані.

FR-22. Локалізація інтерфейсу (4 мов). Пріоритет: M; сервіс: LocalesService + Frontend; критерій: для кожної мови повертається повний словник із fallback.

FR-23. Push-сповіщення через MatchBot. Пріоритет: S; сервіс: MatchBot; критерій: mutual match/message створює push для offline-користувача.

FR-24. Аналітика подій (4 типи). Пріоритет: M; сервіс: StatisticsService + ClickHouse; критерій: gRPC event записується у ClickHouse і доступний в агрегаціях.

FR-25. Завантаження геоданих з OSM. Пріоритет: C; сервіс: LocationUploader (Job); критерій: job ідемпотентно оновлює країни, області, міста та полігони.

FR-26. Зіставлення IP та локації. Пріоритет: C; сервіс: IPLocationsUploader; критерій: IP lookup повертає location_id або контрольований default.

FR-27. Trivy-сканування образів у CI. Пріоритет: M; сервіс: CI/CD; критерій: pipeline проходить тільки при Trivy result = 0 CVE.

FR-28. FluxCD-автоматичне розгортання. Пріоритет: M; сервіс: Cluster; критерій: новий SemVer-тег оновлює manifest і запускає rolling-update.

Нефункціональні вимоги детально розглянуто у п. 1.1 та підсумовано нижче (фрагмент). Для кожної вимоги вказано ідентифікатор, категорію, формулювання та цільову метрику.

NFR-1. Performance: час відгуку API; метрика: $p95 \leq 100$ мс (без матчмейкінгу).

NFR-2. Performance: матчмейкінг; метрика: $p95 \leq 200$ мс при 100 тис. анкет.

NFR-3. Performance: чат; метрика: end-to-end $p95 \leq 250$ мс.

NFR-4. Scalability: одночасні WebSocket; метрика: $\geq 100\,000$.

NFR-5. Scalability: backend-екземпляри; метрика: необмежено за HPA.

NFR-6. Reliability: rolling-update; метрика: 0 downtime.

NFR-7. Reliability: single-pod failure; метрика: без user-impact.

NFR-8. Security: CVE у образах; метрика: 0 CVE.

NFR-9. Security: non-root; метрика: усі Pod'и UID ≥ 1000 .

NFR-10. Security: TLS; метрика: 1.3 на всіх ingress.

NFR-11. Maintainability: лінери; метрика: 41 у golangci-lint.

NFR-12. Maintainability: coverage; метрика: $\geq 50\%$ statements.

NFR-13. Observability: structured logs; метрика: request_id в кожному записі.

2.3 Проектування бази даних

Модель даних розроблена в нотації «сутність-зв'язок» (Entity-Relationship). Вона містить понад 50 таблиць, що покривають усі домени системи. Фрагмент ER-діаграми, що містить найважливіші сутності та їхні зв'язки, показано на рисунку 2.12.

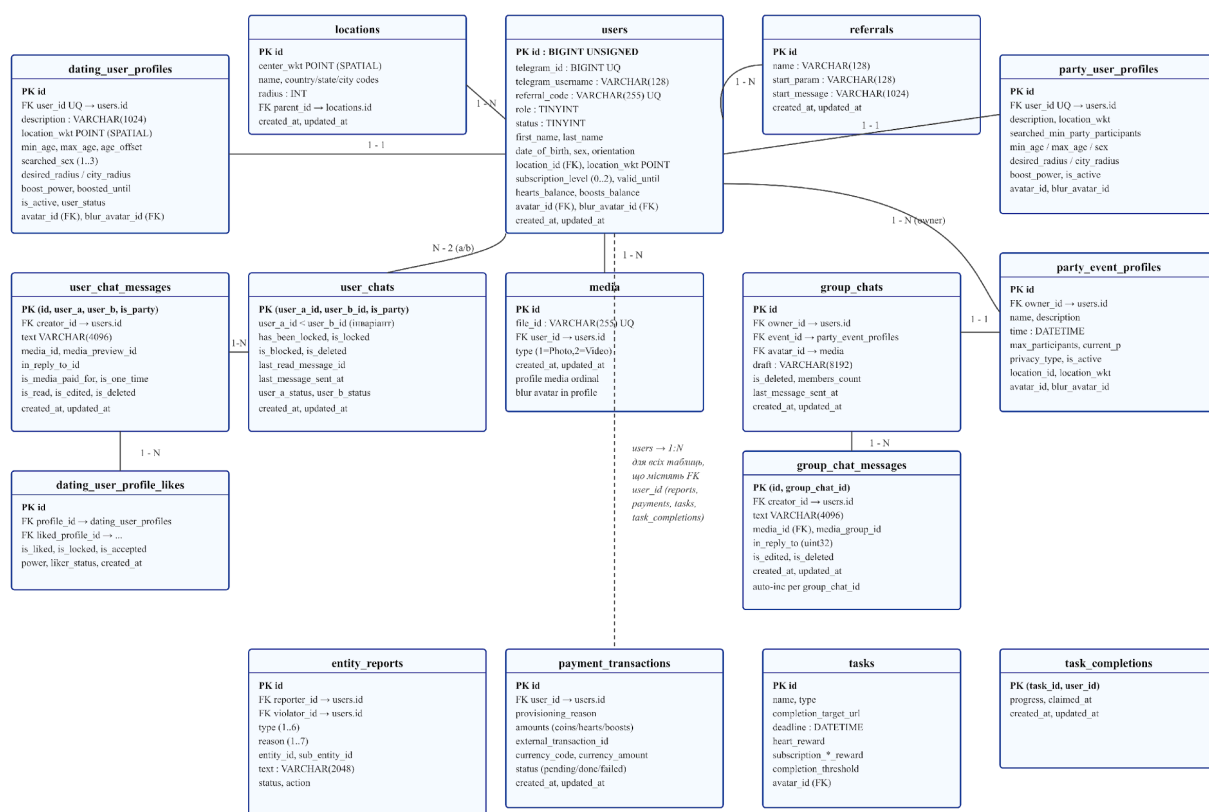


Рис. 2.12 ER-діаграма бази даних (фрагмент, основні сутності)

Центральною сутністю є *users* – обліковий запис. Кожен користувач має один *dating_user_profiles* (зв'язок 1:1) і один *party_user_profiles* (1:1) – окремі профілі для двох режимів. Користувач може бути господарем багатьох *party_event_profiles* (1:N через *owner_id*). Користувач може бути запрошувальником інших користувачів (1:N self-reference через *invited_by_user*) та використати реферальне посилання *referrals* (N:1 через *invited_by_link*).

Профілі (`dating_user_profiles`, `party_user_profiles`) мають окремі додаткові таблиці для багатозначних атрибутів: `dating_user_profile_media` (1:N – додаткові фото у профілі понад аватара), `dating_user_profile_interests` (M:N з довідником `interests`), `dating_user_profile_attribute_assignments` (M:N з довідниками `attributes` та значеннями).

`dating_user_profiles` лайкають один одного через таблицю `dating_user_profile_likes` (M:N з атрибутами `is_liked`, `is_locked`, `is_accepted`, `power`); аналогічно для `party_user_profile_likes` та `party_event_profile_likes`.

`party_event_profiles` мають один пов'язаний `group_chats` (1:1 через `event_id`); `group_chats` мають N учасників через `group_chat_members` (M:N з `users`); `group_chats` мають N повідомлень `group_chat_messages` (1:N).

Ключове рішення моделі – композитний первинний ключ для 1-to-1 чатів `user_chats(user_a_id, user_b_id, is_party)` з інваріантом `user_a_id < user_b_id`. Це дозволяє зберігати один запис на пару користувачів незалежно від того, хто кому написав першим, і одночасно мати окремий чат для `dating` та `party`-режимів.

`user_chat_messages` має композитний первинний ключ (`id`, `user_a_id`, `user_b_id`, `is_party`), де `id` – інкрементальне ціле, унікальне в межах пари. Це дає (а) просте сортування повідомлень за `id`, (б) безпечне посилання `in_reply_to_id` (інкремент в межах одного чату), (в) ізоляцію між чатами на рівні фізичних рядків.

Аналогічна структура у `group_chat_messages(id, group_chat_id)` – інкрементальний `id` у межах конкретної групи.

Інші важливі сутності: `media` (медіа-файли з `file_id` і типом), `locations` (ієрархія країн/областей/міст), `tasks` + `task_completions`, `entity_reports` (скарги), `payment_transactions`, `gifts`, `pop_ups`.

Нижче подано опис структури п'яти найважливіших таблиць – у вигляді переліку полів з типами даних, ключами та індексами.

Таблиця `users` є кореневою сутністю системи. Первинний ключ – синтетичний `id` типу `BIGINT UNSIGNED` з автоінкрементом. Серед бізнес-полів: унікальний `telegram_id` (для пошуку при логіні), `telegram_username`, унікальний `referral_code`, роль (`role` типу `TINYINT`: значення 1 – звичайний користувач, 200 –

адміністратор), статус облікового запису (1 – активний, 2 – видалений, 3 – заблокований), локаль, часовий пояс, опційні email та хеш password, ідентифікатор реферера й посилання-реферала, антропометричні поля (зріст, вага, стать, орієнтація), геолокація – location_id (зовнішній ключ до locations) та точка location_wkt типу POINT з SRID 4326, поля підписки (subscription_type, subscription_level, subscription_valid_until), баланси внутрішніх валют (hearts_balance, boosts_balance), посилання на медіа аватара та його заблюровану версію. Передбачені часові мітки created_at та updated_at. Індексація: SPATIAL-індекс по location_wkt, унікальні індекси по telegram_id і referral_code, композитний індекс (status, created_at) – для аналітики реєстрацій.

Таблиця dating_user_profiles містить параметри dating-профілю. Первинний ключ – id; зв'язок 1:1 з users через зовнішній ключ user_id з обмеженням UNIQUE. Поля: текстовий опис до 1024 символів, точка location_wkt, пререференції пошуку (min_age, max_age, age_offset, searched_sex, три радіуси – desired_radius, city_radius, out_city_radius, прапорець only_city), статус активності (is_active, user_status), параметри boost'у (boost_power, boosted_until), посилання на аватар та його заблюровану версію. Індексація: SPATIAL-індекс по location_wkt, композитний індекс (user_status, is_active).

Таблиця user_chats реалізує зв'язок «користувач-користувач-режим» з композитним первинним ключем (user_a_id, user_b_id, is_party). Інваріант user_a_id < user_b_id забезпечується на рівні бізнес-логіки і додатково перевіркою CHECK; завдяки цьому пара (A, B) і пара (B, A) представлена тим самим рядком. Прапорець is_party розрізняє dating- та party-чати між тими ж учасниками. Серед інших полів – статуси блокування й видалення, ідентифікатор останнього прочитаного повідомлення, час останнього повідомлення, статуси обох учасників. Індексація: окремі композитні індекси для пошуку чатів конкретного користувача А або В із сортуванням за часом останнього повідомлення.

Таблиця user_chat_messages має композитний первинний ключ (id, user_a_id, user_b_id, is_party). Поле id – інкрементальне ціле, унікальне у межах однієї пари; такий ключ дозволяє безконфліктно генерувати послідовні номери повідомлень

одночасно у багатьох чатах без глобального лічильника, водночас зберігаючи природне сортування. Зовнішній ключ посилається на user_chats за тим самим композитним ключем. Серед полів – посилання на автора повідомлення, текст до 4096 символів, опційний медіа-вміст, ідентифікатор повідомлення-відповіді, прапорці платного, одноразового та переглянутого медіа, видалення. Інкремент виконується транзакцією з конструкцією `SELECT COALESCE(MAX(id),0)+1 ... FOR UPDATE` (детально – у п. 3.2).

Таблиця group_chat_messages має аналогічну структуру для групових чатів party-event'ів: композитний первинний ключ (id, group_chat_id), поля автора, тексту, медіа, повідомлення-відповіді, прапорці редагованості й видалення. Інкремент id виконується аналогічним `FOR UPDATE`-блокуванням рядка group_chats для атомарного отримання нового номера повідомлення.

Інші таблиці моделі (media, locations, tasks, task_completions, entity_reports, payment_transactions, referrals, gifts, pop_ups, проміжні таблиці інтересів і атрибутів) побудовані за подібними принципами і також забезпечені композитними індексами для типових запитів.

Геопошук – найвідповідальніша операція з боку продуктивності. Реалізовано через MariaDB-тип POINT зі SRID 4326 (WGS84 – система координат GPS) та SPATIAL INDEX (R-Tree). На рисунку 2.13 проілюстровано принцип роботи R-Tree.

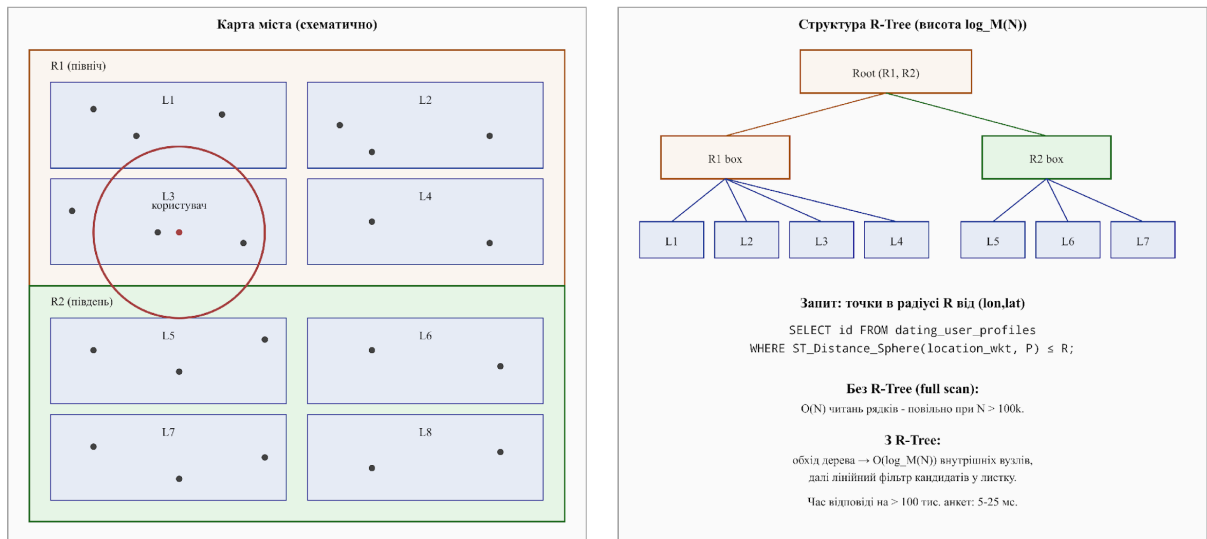


Рис. 2.13 Просторовий індекс R-Tree для геопошуку (POINT, SRID 4326)

R-Tree організовує точки у вкладену ієрархію прямокутників (Minimum Bounding Rectangles, MBR). Запит «знайти всі точки в радіусі R від (lon, lat) » виконується так:

- Від кореня дерева перевіряється кожен MBR на перетин з колом радіусу R .
- Рекурсивно входимо лише в ті піддерева, чий MBR перетинає коло.
- У листках виконується точний фільтр через `ST_Distance_Sphere`.

Складність – $O(\log_M N)$ внутрішніх вузлів проти $O(N)$ для повного скану. На 100 тисяч анкет час відповіді вдається утримувати у межах 5-25 мс. Приклад – фрагмент продуктового запиту:

```
SELECT dp.*, ST_Distance_Sphere(dp.location_wkt, P) AS distance
FROM dating_user_profiles dp
WHERE dp.is_active = 1
AND dp.user_status = 1
AND ST_Distance_Sphere(dp.location_wkt, P) <= 50000 -- 50 км
AND dp.user_id != :requester_id
AND dp.id NOT IN (:ignore_ids)
AND dp.sex = :sex
AND TIMESTAMPDIFF(YEAR, dp.date_of_birth, CURDATE()) BETWEEN :min_age AND
:max_age
```



```
ORDER BY dp.boost_power DESC, distance ASC
LIMIT 20;
```

де P = ST_GeomFromText('POINT(lon lat)', 4326).

Окремий випадок – пошук «лише в межах міста»: він виконується через порівняння ID локації користувачів.

Окрім spatial-індексу, побудовано:

- users(telegram_id) – UNIQUE для пошуку при логіні.
- users(referral_code) – UNIQUE для розрахунку реферера.
- users(status, created_at) – для адмін-статистики.
- dating_user_profile_likes(profile_id, liked_profile_id) – для перевірки mutual-match.
- user_chat_messages(user_a_id, user_b_id, is_party, created_at) – для пагінації повідомлень.
- payment_transactions(user_id, status, created_at) – для звітності.
- Композитні індекси на інші поля, що часто фільтруються разом, у порядку від селективнішого до менш селективного.

Жадібне читання (Preload) виконується GORM-методом

Preload("Avatar").Preload("Media"), що виконує окремий SELECT для пов'язаних сутностей з масовим WHERE id IN (...) – це уникнення проблеми N+1.

Денормалізація лічильників: last_message_sent_at, last_read_message_id, members_count зберігаються прямо у user_chats / group_chats для уникнення COUNT(*) при кожному запиті списку чатів.

2.4 Проектування програмного інтерфейсу

REST API системи реалізовано на базі URI-версіонування з префіксом /api/v1. Авторизація – через заголовок Authorization: Bearer <jwt>. Content-Type для всіх mutation-запитів – application/json; для POST /media_upload – multipart/form-data. Усі endpoints описано у форматі OpenAPI 3 через Swagger та доступні за адресою /swagger/index.html.

Загальна карта endpoints наведена на рисунку 2.14, що групує їх за доменами. За RegisterRoutes у коді MatchBackend система експонує 78 HTTP routes: 77 під /api/v1 та root route /swagger/*any.

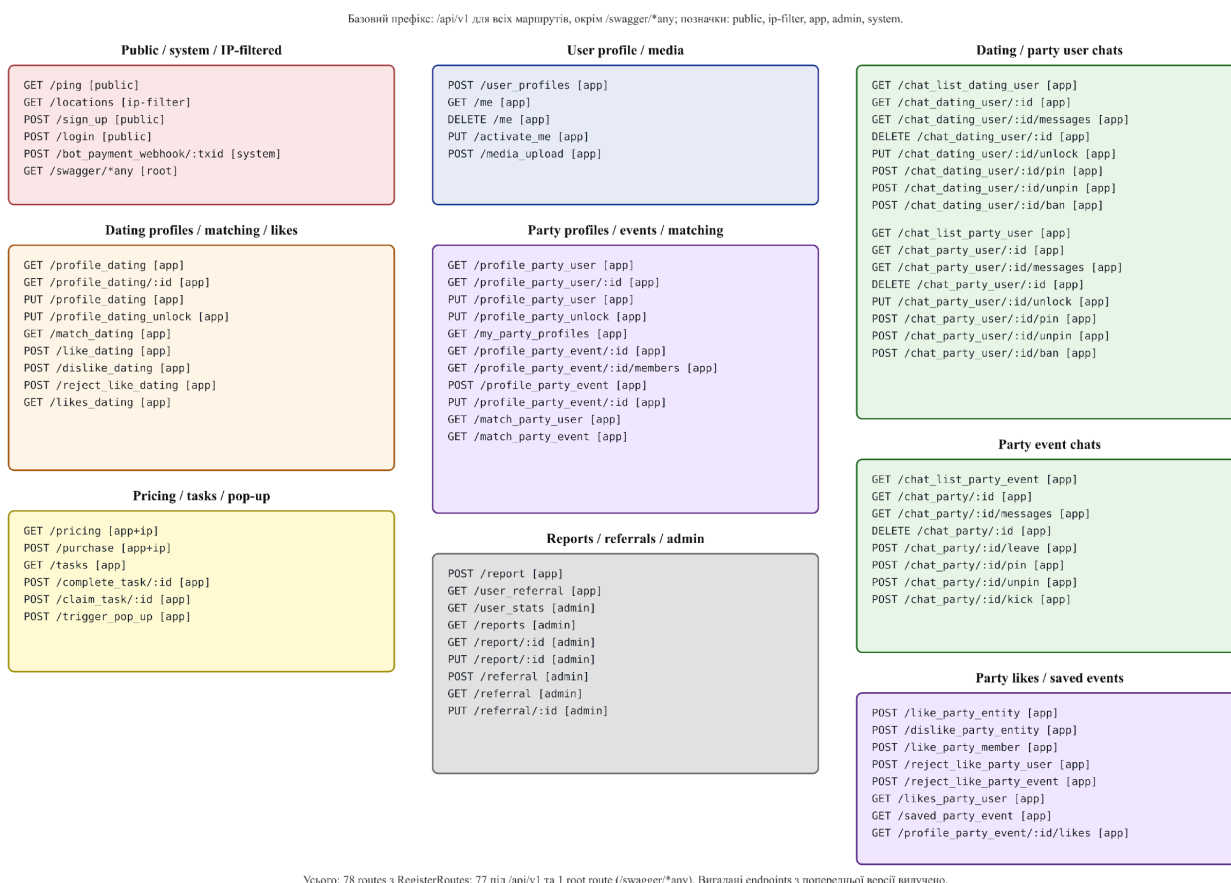


Рис. 2.14 Карта REST API (категоризація endpoints)

Найважливіші endpoint'и наведено нижче (фрагмент). Для кожного зазначено HTTP-метод і шлях, призначення, потрібний рівень авторизації, формат запиту та відповіді.

- POST /sign_up – реєстрація через Telegram. Auth: -. Запит: tg-init-data. Відповідь: {token, user_id}.
- POST /login – email-логін. Auth: -. Запит: {email, password}. Відповідь: {token}.
- GET /me – профіль поточного користувача. Auth: AppUser. Запит: -. Відповідь: UserPrivateRes.

- POST /user_profiles – оновлення профілю. Auth: AppUser. Запит: UserPrivateProfilesReq. Відповідь: 200.
- GET /match_dating – геопошук у dating-режимі. Auth: AppUser. Запит: query. Відповідь: DatingProfile[].
- POST /like_dating – лайк dating-профілю. Auth: AppUser. Запит: {profile_id, hash, is_liked}. Відповідь: LikeRes.
- POST /media_upload – завантаження медіа. Auth: AppUser. Запит: multipart. Відповідь: MediaRes.
- POST /profile_party_event – створення event'у. Auth: AppUser. Запит: CreatePartyEvent. Відповідь: {event_id}.
- POST /like_party_entity – лайк event'у (заявка на приєднання). Auth: AppUser. Запит: {event_id, hash, is_liked}. Відповідь: 200.
- GET /chat_list_dating_user – список dating-чатів. Auth: AppUser. Запит: limit, offset. Відповідь: ChatListRes.
- GET /chat_dating_user/:id/messages – повідомлення чату. Auth: AppUser. Запит: limit, offset. Відповідь: Message[].
- POST /report – скарга. Auth: AppUser. Запит: CreateReport. Відповідь: 201.
- GET /reports – усі скарги. Auth: Admin. Запит: -. Відповідь: Report[].
- PUT /report/:id – прийняти/відхилити скаргу. Auth: Admin. Запит: {action, status}. Відповідь: 200.

Стандартизована форма відповіді розрізняється для успішних і помилкових результатів. Успішна відповідь має формат { data: ..., meta?: { total, limit, offset } }, статус 200/201. Відповідь з помилкою – { error: { code, message, details } }, статус 400/401/403/404/409/422/500.

Контракти gRPC описані у proto3-схемі, що генерує клієнтський та серверний код як для Go, так і для TypeScript. Основний сервіс – GRPCService, що оголошує два методи:

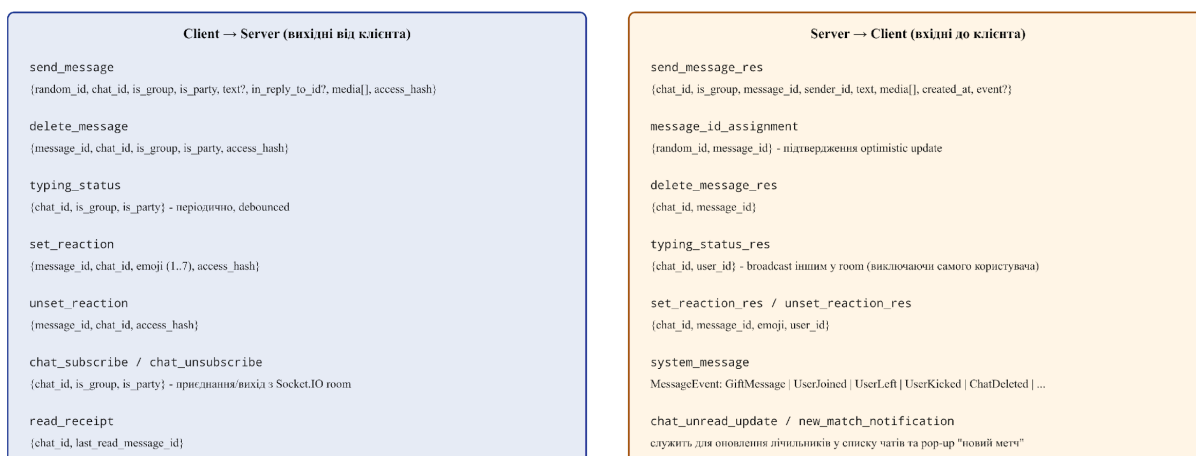
```
service GRPCService {
  rpc Stream(stream PayloadRes) returns (stream PayloadReq);
  rpc SendSystemMessage(SystemMessage) returns (Ack);
}
```

Stream – bidirectional streaming, по якому проксі та backend обмінюються повідомленнями PayloadRes (від клієнта/проксі до backend) та PayloadReq (від backend до проксі/клієнта). Усередині – oneof з типами:

- SendMessageReq/Res – надсилання повідомлення.
- DeleteMessageReq/Res – видалення.
- TypingStatusReq/Res – індикатор набору.
- SetMessageReactionReq/Res, UnsetMessageReactionReq/Res – реакції.
- ChatSubscribeReq – підписка на оновлення чату.
- MessageIdAssignment – відповідь на оптимістичне повідомлення (відповідність random_id до message_id).

Перевага protobuf-кодування – компактність (varint, fixed64) та строга типізація. Поле message_id типу uint32 при значенні 100 займає 1 байт, при значенні 1 000 000 – 3 байти. У JSON еквівалентне поле "message_id":1000000 – 18 байт. На високонавантажених маршрутах це дає чотирикратне зменшення трафіку.

Подієва модель Socket.IO для real-time-чату наведена на рисунку 2.15. Розрізняються вихідні події від Mini App до сервера та вхідні події від сервера до Mini App.



Усі payloads проходять normalizeSocketPayload() (camelCase ↔ snake_case, BigInt-safe parsing) перед потраплянням у Redux.

Рис. 2.15 Класифікація подій Socket.IO у системі чатів

Усі payload'и проходять через нормалізатор `normalizeSocketPayload()` (на стороні клієнта) і `BindPayloadJSON()` (на стороні проксі), що виконує перетворення між `camelCase` і `snake_case`, `BigInt-safe`-парсинг ID, валідацію `media`-масивів. `random_id` – клієнтський ідентифікатор, який сервер повертає у `message_id_assignment` для оптимістичного оновлення UI.

2.5 Проектування інтерфейсу користувача

Mobile Mini App – основна форма доставки інтерфейсу. Тому при проектуванні дотримано таких принципів.

Telegram Mini App UI Guidelines передбачають використання нативних компонент Telegram (`BackButton`, `MainButton`, `ThemeParams`), узгодження кольорів з активною темою месенджера (світла/темна), мінімізацію необхідності повторно приходити до Telegram-чату.

Mobile-first підхід задає мінімальний розмір контролю – 44×44 pt, безпечні відступи від нотчів.

Progressive disclosure: складні фільтри сховані за «Advanced»-розділом; основні дії дають максимальну простоту.

Optimistic UI: повідомлення у чаті, лайки, реакції – відображаються в інтерфейсі до підтвердження сервером.

Lazy loading: фото підвантажуються на льоту за отриманим `file_id`; перші 200 повідомлень чату з кешу IndexedDB.

Time-to-interactive менше 2 секунд на середньому смартфоні Android (4 ГБ RAM, 2024 рік).

A11y – відповідність WCAG 2.1 AA для головних сценаріїв; ARIA-labels на кнопках; підтримка темної теми Telegram.

Виокремлено п'ять головних сценаріїв (user flows).

Flow 1. Новий користувач відкриває Mini App у Telegram, проходить автоматичну авторизацію, виконує онбординг (3 кроки: ім'я, дата народження,

локація), завантажує аватар і потрапляє у стрічку матчмейкінгу. Загальна тривалість – близько 90 секунд.

Flow 2. Постійний користувач відкриває Mini App, одразу потрапляє у стрічку, виконує свайп та лайк, отримує mutual-match toast, переходить у чат і відсилає перше повідомлення.

Flow 3. Користувач, що хоче провести подію, відкриває вкладку «Party», натискає «Створити event», вводить назву, час, локацію, фото, натискає «Опублікувати» і отримує посилання, яким може поділитися у Telegram.

Flow 4. Користувач, що хоче придбати підписку Elite, переходить у Profile, далі Subscription, обирає Elite на місяць, отримує інвойс через Telegram Stars, виконує оплату у самому Telegram і дочекується автоматичного оновлення статусу через webhook.

Flow 5. Користувач, що отримав неприйнятне повідомлення, довго натискає на нього, обирає «Скарга», обирає причину «Домагання», описує проблему та submit'ить; адмін бачить скаргу у черзі.

Реальні мобільні екрани трьох ключових сценаріїв – стрічка матчмейкінгу, екран чату, профіль – наведено на рисунках 2.16, 2.17, 2.18 відповідно.

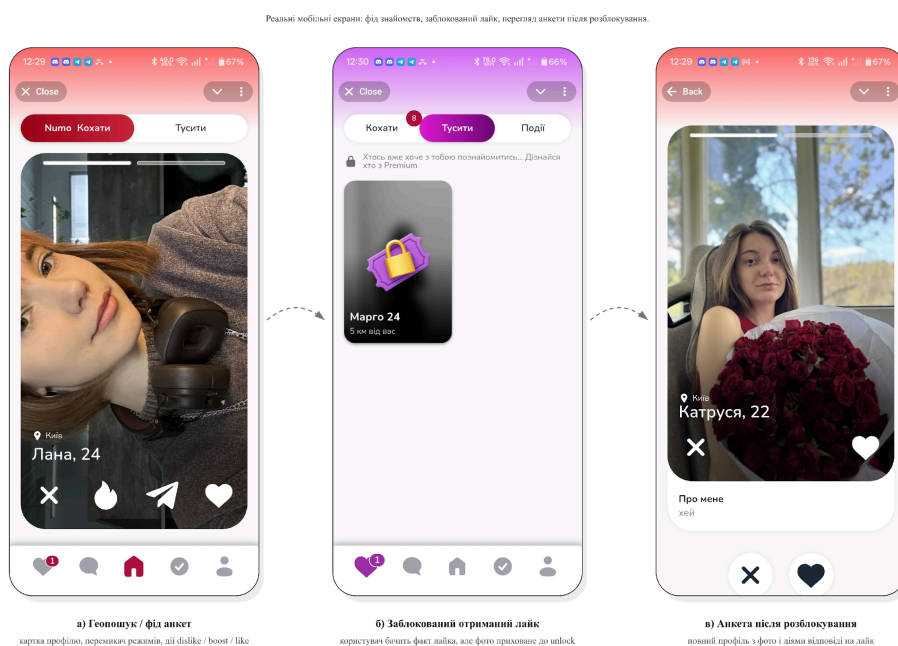


Рис. 2.16 Екрани матчмейкінгу та отриманого лайка

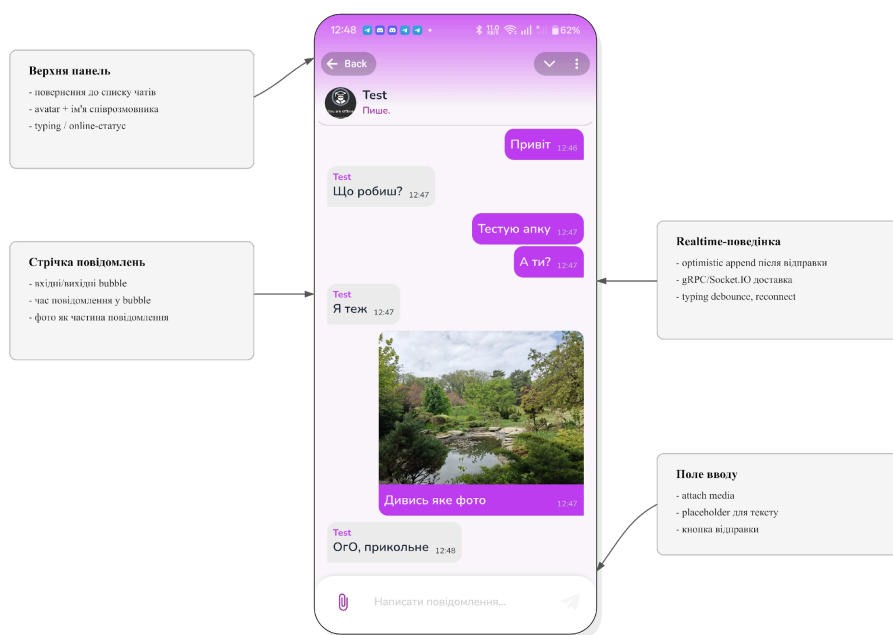


Рис. 2.17 Екран чату (1-to-1)

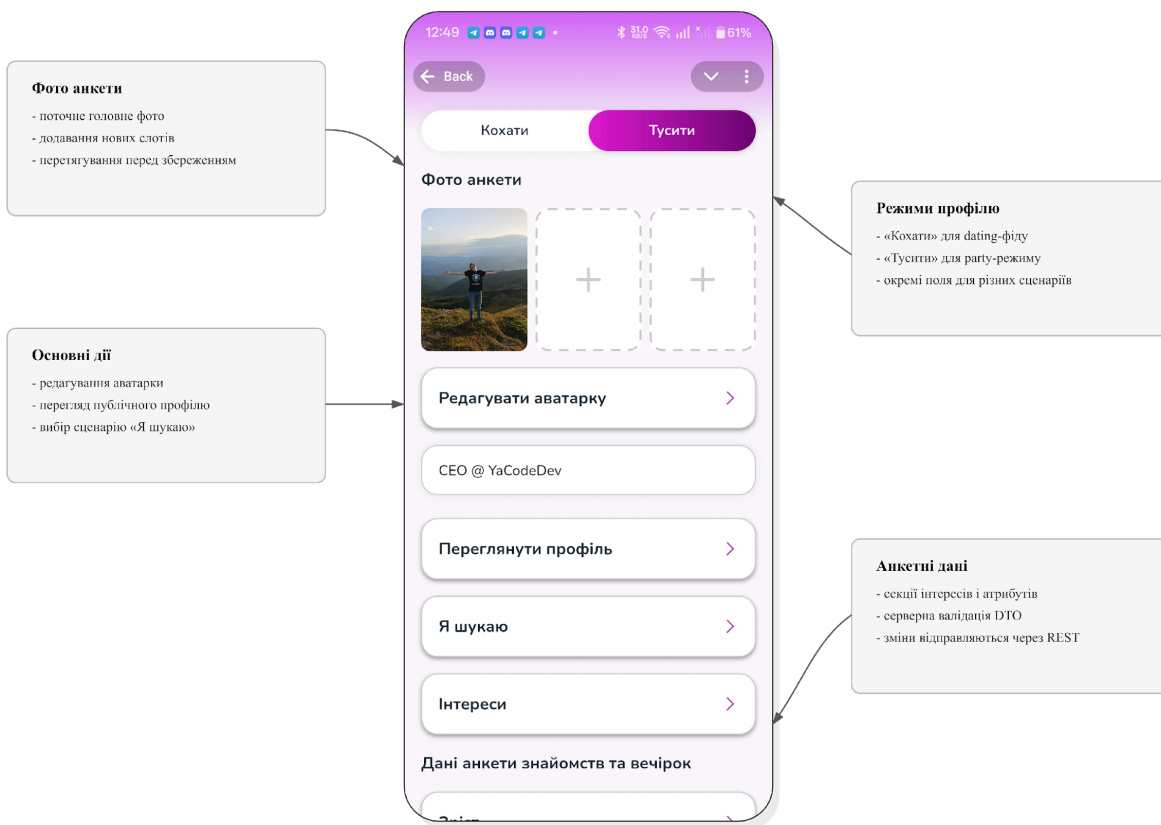


Рис. 2.18 Екран редагування профілю

Для адмін-панелі (десктопна SPA) використано таблично-орієнтований дизайн з шаблонами TanStack Table: широкий sidebar з навігацією (Користувачі, Скарги, Реферрери, Завдання, Статистика, Платежі), центральна таблиця з фільтрами і пагінацією, drawer з деталями. Графіки реалізовано на Recharts, фільтри - Zod-валідовані форми.

3 РЕАЛІЗАЦІЯ ТА ТЕСТУВАННЯ СИСТЕМИ

3.1 Структура проекту й організація коду

Кодова база системи Match складається з тринадцяти незалежних сервісів – кожний сервіс розміщено в окремому Git-репозиторії з власною документацією, конфігурацією CI/CD, Dockerfile та переліком залежностей. Спільні політики (бренч-захист, конвенції) задаються через GitHub Organization-рівні правила. Загальна структура наведена на рисунку 3.1.

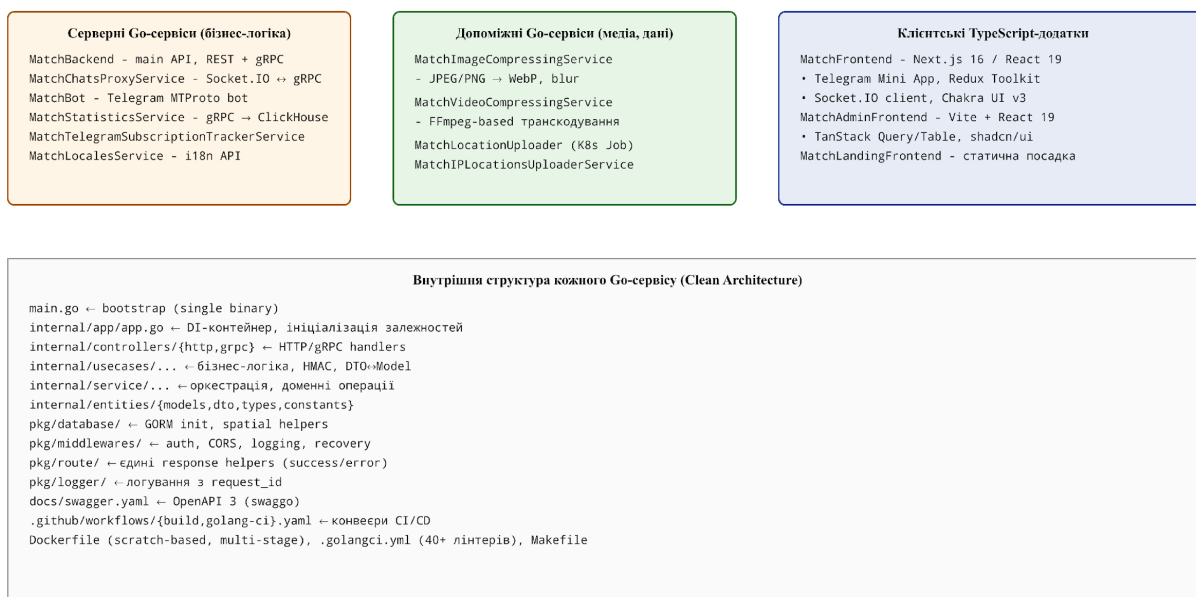


Рис. 3.1 Структура проекту Match (13 сервісів)

На верхньому рівні розташовуються:

1. Серверні Go-сервіси бізнес-логіки (MatchBackend, MatchChatsProxyService, MatchBot, MatchStatisticsService, MatchTelegramSubscriptionTrackerService, MatchLocalesService).

2. Допоміжні Go-сервіси для медіа та геоданих (MatchImageCompressingService, MatchVideoCompressingService, MatchLocationUploader, MatchIPLocationsUploaderService).
3. Клієнтські TypeScript-додатки (MatchFrontend, MatchAdminFrontend, MatchLandingFrontend).

Усі Go-сервіси наслідують єдину внутрішню структуру шарів Clean Architecture, описану у п. 2.1. Окрім вихідного коду, кожний сервіс містить:

9. Dockerfile – multi-stage build на базі golang:1.26-alpine із фінальною стадією FROM scratch.
10. .golangci.yml – конфігурація 41 лінера (детально у п. 3.6).
11. Makefile – цілі для генерації protobuf, документації Swagger, форматування коду, локального запуску.
12. .github/workflows/ – два YAML-файли: golang-ci.yml (для lint/test на push/PR) та build.yaml (build & push на тег v*).

Для frontend-додатків використовується додаткова структура: package.json, bun.lock, tsconfig.json, vite.config.ts (для Admin / Landing) або next.config.ts (для Mini App), eslint.config.js, prettier.config.js.

Конвенції коду такі.

- Go: golangci-lint v2 із 41 лінером, формат через послідовність gofmt, далі gofumpt, далі goimports, далі golines з максимальною довжиною рядка 120 символів.
- TypeScript: ESLint + Prettier; усе під strict: true, заборонений any.
- Protobuf: protoc-gen-go + protoc-gen-go-grpc; згенерований код у proto/-піддиректорії, не редагується вручну.
- Документація API: Swagger 2.0 та OpenAPI 3.1 через swaggo (make docs).
- Конвенції коммітів: Conventional Commits (feat/fix/chore/refactor/...).
- Пре-комміт: lefthook для локальних попередніх перевірок (lint).

Усі сервіси версіонуються незалежно за SemVer. Тег у Git формату v1.4.2 запускає build конкретного сервісу (через path-filter у workflow).

3.2 Реалізація серверної частини

MatchBackend – найбільший серверний сервіс, що інкапсулює основну бізнес-логіку. Composition root оформлено у пакеті internal/app. Вхідна точка main.go мінімалістична: створює об'єкт додатку та викликає метод Run(). Усі залежності вибудовуються у явній послідовності:

- Завантаження конфігурації з ENV-змінних та валідація.
- Ініціалізація structured-логера (yalogger з KV-форматом, рівнем з ENV).
- Підключення до MariaDB через MaxScale (read-write splitter), параметризація пулу (MaxOpenConns=128, MaxIdleConns=16, ConnMaxLifetime=30m).
- Автоматична міграція схеми через GORM.
- Підключення до DragonflyDB (go-redis/v9 з пулом 64 з'єднань).
- Підключення до MinIO через minio-go/v7.
- Створення rolling hash-валідаторів (по одному для media/chat/like/price).
- Створення репозиторіїв, передача їм *gorm.DB.
- Створення сервісів, що оркеструють репозиторії.
- Створення use cases, що оркеструють сервіси.
- Створення HTTP-сервера Gin із ланцюжком middleware.
- Створення gRPC-сервера на порту 50051 з реєстрацією всіх gRPC-handler'ів.
- Реєстрація HTTP-routes.
- Запуск HTTP та gRPC серверів у власних goroutine з graceful shutdown через signal.NotifyContext(context.Background(), SIGTERM, SIGINT).

Така експліцитна послідовність спрощує налагодження та зберігає чистоту імпортів. Структура App зберігає посилання на ключові залежності для зовнішнього доступу у тестах.

Усі HTTP-запити проходять через однаковий ланцюжок middleware (рисунок 3.2). Послідовність важлива: Recovery загортає увесь ланцюжок (у разі panic – повертається 500 без падіння процесу), Logger створює yalogger з request_id і кладе його в context, Interceptor у debug-режимі друкує параметри запиту, CORS обмежує allowed-origins white-list'ом у production-режимі.

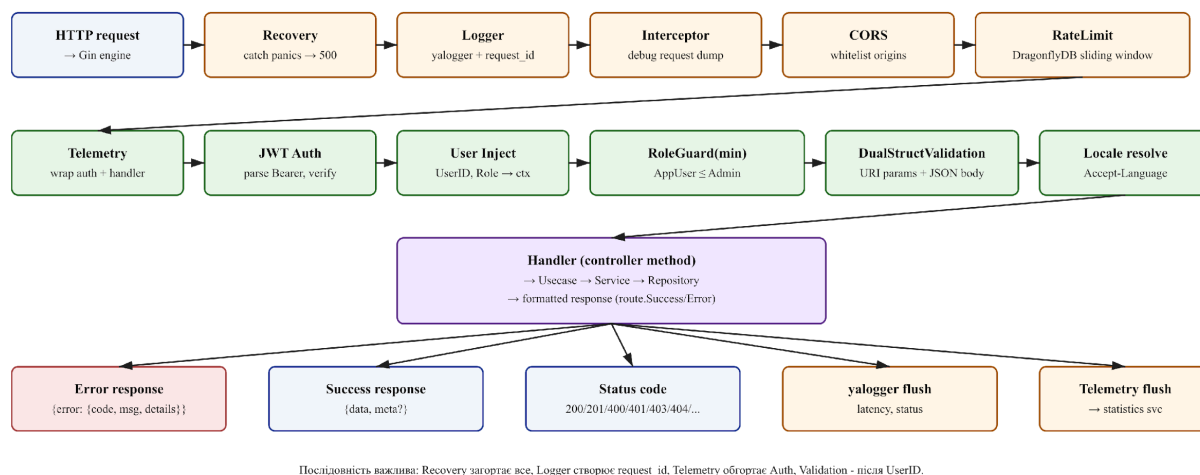


Рис. 3.2 Ланцюжок middleware HTTP-запиту

Безпекова частина: RateLimit використовує DragonflyDB sliding-window-лічильник з ключем `rl:<ip>:<path>` для неавторизованих маршрутів та `rl:<user_id>:<path>` для авторизованих; Telemetry стоїть перед JWT Auth і обгортає авторизацію та handler; JWT Auth парсить Bearer-токен, перевіряє підпис HS256 ключем JWT_SECRET, валідує exp, читає user_id/role/status і додає їх до gin.Context; RoleGuard(min) перевіряє роль (наприклад, RoleGuard(RoleAdmin) для адмін-маршрутів); DualStructValidation валідує одночасно URI-параметри та JSON-body через теги `validate:"required,min=1,max=50"`; Locale визначає локаль за заголовком Accept-Language або полем користувача; Telemetry надсилає мета-дані запиту у MatchStatisticsService через async-канал.

Один з ключових механізмів безпеки системи – система валідаційних rolling-хешів, що супроводжують усі mutation-запити, які працюють з конкретною сутністю (профіль, повідомлення, подія). Архітектуру системи показано на рисунку 3.3.

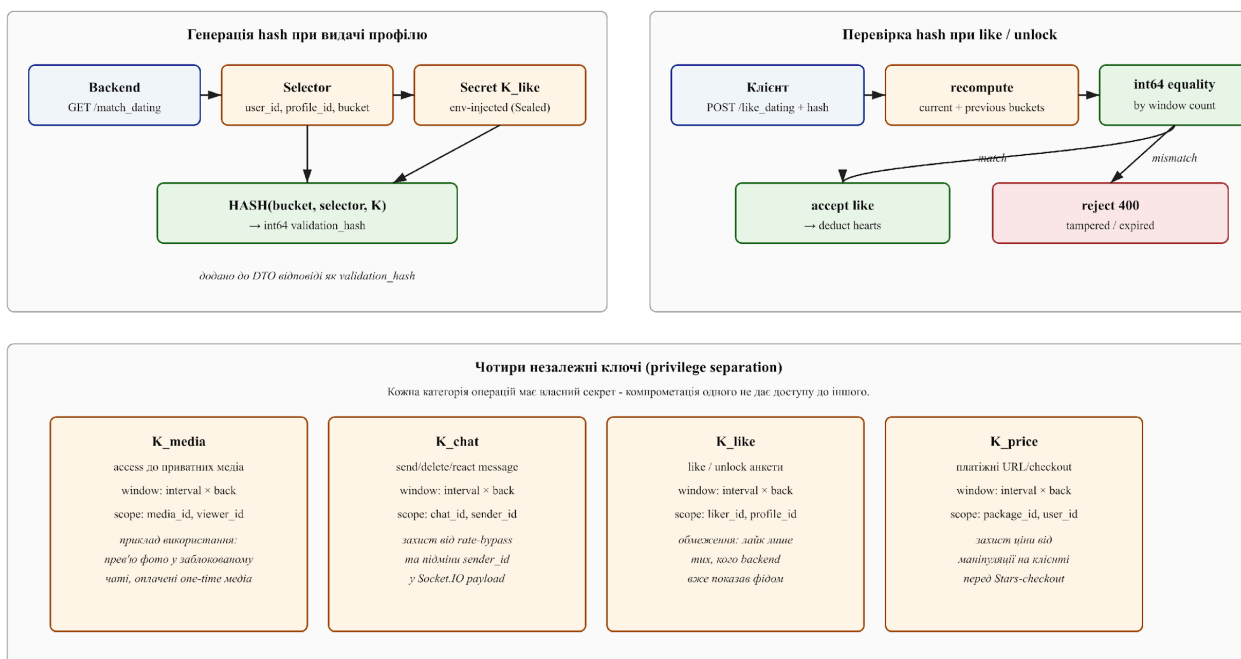


Рис. 3.3 Система валідаційних rolling-хешів

Технічна реалізація. Створено чотири незалежних валідатори, кожен зі своїм секретом і таргет-доменом:

1. K_media – для access до приватних медіа (наприклад, фото у заблокованому профілі).
2. K_chat – для send/delete/react на повідомлення.
3. K_like – для лайку та розблокування профілю.
4. K_price – для платіжних URL та checkout'у.

Розділення ключів реалізує принцип privilege separation: компрометація однієї категорії не дає доступу до іншої. Кожен ключ – 32-байтова випадкова послідовність, завантажена з Kubernetes Secret.

При формуванні відповіді сервера, що містить посилання на чутливу сутність, генерується одноразовий токен у формі:

```

token = HASH(
    time_bucket || user_id || entity_id
    || ...metadata that affects validity
    || K
) → int64

```

де `time_bucket = floor(now_unix / window_size)`. Результат `HASH(...)` інтерпретується як `int64` – це робить `payload` компактним. Окремий `ts` не передається: під час перевірки `Validate` повторно обчислює хеш для поточного `time_bucket` і `window_count` попередніх `bucket`’ів, визначених через `window_size`. Це захищає від трьох класів атак:

1. Enumeration: атакуюча сторона не може перебирати ID профілів, підставляючи свої – хеш буде відрізнятися.
2. Replay: хеш приймається лише в межах налаштованих `rolling`-вікон.
3. CSRF: токен пов’язаний з конкретним `user_id`, не може бути використаний іншою стороною.

Перевірка виконується через повторне обчислення `int64` для допустимих `bucket`’ів і звичайне порівняння з отриманим `validation_hash`. У разі провалу перевірки повертається 400 з нейтральним повідомленням про невалідний `hash`.

Сервіс `MatchmakingService` реалізовано як композицію кількох `use cases`:

- `FetchDatingUserProfiles(ctx, requesterID, filters)` – головний метод.
- `FetchPartyUserProfiles(ctx, requesterID, filters)` – для `party`-режиму.
- `FetchPartyEventProfiles(ctx, requesterID, filters)` – пошук подій.

Кожен з них формує SQL-запит з обов’язковими фільтрами `is_active=1`, `user_status=1`, `dp.user_id != requesterID` та опціональними – за віком, статтю, радіусом. Boost-механіка: профілі з `boost_power > 0` та активним `boosted_until` отримують додаткове сортування `ORDER BY boost_power DESC` перед `distance ASC`. Пагінація – через клієнтський `ignore_ids[]`, що передається у `WHERE dp.id NOT IN (?)`.

Для уникнення `monopolization profile` у стрічці одного користувача застосовується `RAND()-secondary-sorting` у межах одного рівня пріоритету.

Реалізація бекенд-частини чату міститься у `internal/usecases/chat.go` та `internal/service/chat.go`. Найскладніше технічне рішення – атомарна генерація послідовного `id` повідомлення у межах однієї пари (`user_a_id`, `user_b_id`, `is_party`).

Класичне рішення – `AUTO_INCREMENT` на таблиці – не підходить, бо дало б глобально-послідовний `id` через всі чати, що ускладнює пагінацію та надуманий `in_reply_to_id`. Натомість застосовано:

```
BEGIN;
SELECT COALESCE(MAX(id),0)+1 FROM user_chat_messages
WHERE user_a_id=? AND user_b_id=? AND is_party=?
FOR UPDATE;
INSERT INTO user_chat_messages (id, user_a_id, ...) VALUES (?, ?, ...);
UPDATE user_chats SET last_message_sent_at=NOW()
WHERE user_a_id=? AND user_b_id=? AND is_party=?;
COMMIT;
```

`FOR UPDATE` накладає `row-level X-lock` на рядок `user_chats`, що серіалізує паралельні відправлення в межах однієї пари, але не блокує паралельні відправлення в інших парах. Завдяки цьому продуктивність не страждає.

Інваріант `user_a_id < user_b_id` нормалізує пару незалежно від того, хто кому пише: `OrderUserPair(senderID, receiverID)` повертає коректний пару (`a`, `b`), у пакеті `internal/usecases/user_chat.go`.

Розподіл `chat_id` між `backend`-екземплярами здійснюється функцією `UintIDsIntoInstanceMap`, що приймає список `ID` і поточну `TotalInstanceCount`, повертає `map[instanceID][]chatID`.

Завантаження медіа починається з `REST-endpoint`'у `POST /media_upload`. `Multipart-обробник MatchBackend` читає файл у пам'ять, визначає тип за сигнатурою файлу, перевіряє допустимий `MIME-type`, ліміти розміру і тривалості відео. Після збереження `raw-файлу` до `MinIO` ставиться задача у внутрішню чергу обробки.

Подальший конвеєр – асинхронний (рисунок 3.4).

`ImageCompressingService` отримує ключ об'єкта у `MinIO`, читає `raw-зображення`, застосовує `EXIF-orientation`, конвертує його в один `WebP-файл` і

перезаписом формату прибирає початкові EXIF-метадані. Blur-файл створюється тільки за запитом MatchBackend для безпечного показу заблокованих профілів.

VideoCompressingService запускає FFmpeg-pipeline і записує один MP4-файл у MinIO.

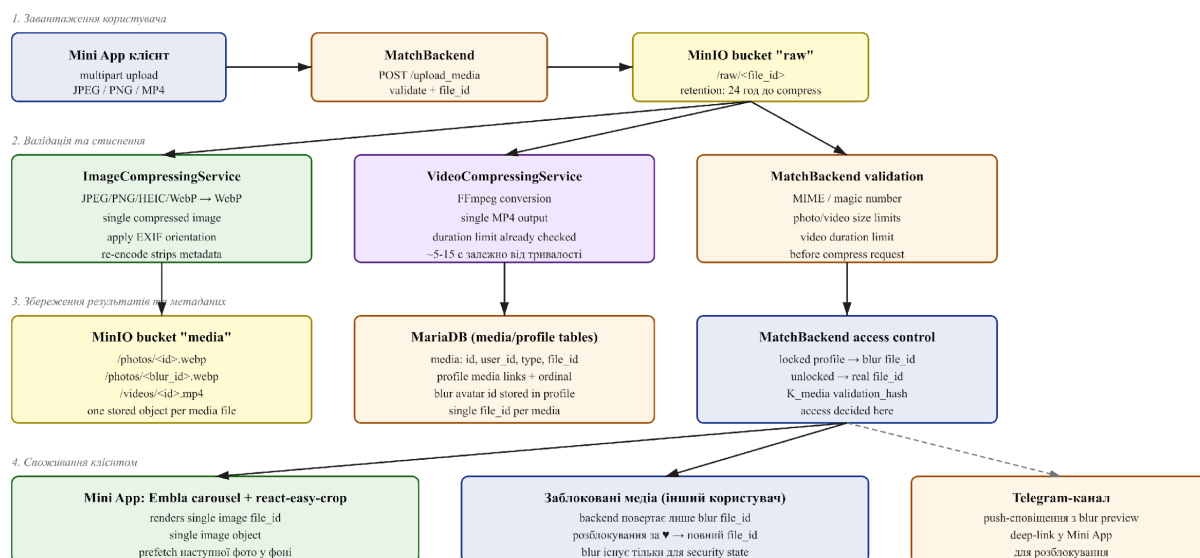


Рис. 3.4 Конвеєр обробки медіа (фото / відео)

Перед стисненням MatchBackend виконує лише серверну валідацію файлу: тип, розмір і тривалість відео. EXIF-метадані не обробляються окремим сервісом: ImageCompressingService читає orientation для правильного повороту і видаляє початкові метадані під час re-encode у WebP.

MatchBackend контролює, який file_id потрапляє у DTO: для заблокованого профілю повертається тільки blur-версія, після розблокування - повний file_id медіа.

Платіжна логіка реалізована у internal/usecases/pricing.go та internal/service/money_ops.go. Прайс-конфігурація живе у JSON-файлі (pricing.json), завантаженому на старті, і містить три рівні підписки (Base, Lite, Elite) × періоди (7/30/180/365 днів) × валюти (Stars, USD, EUR, UAH). Окремо - пакети мікротранзакцій: small_hearts, medium_hearts, big_hearts, single_boost, multi_boost. Можливі знижкові коди (модель discounts).

При запиті POST /purchase Backend генерує запис payment_transactions зі статусом pending, формує URL інвойсу через Telegram Bot API (createInvoiceLink), повертає клієнту. Користувач сплачує у Telegram (Stars або картою). Telegram надсилає подію successful_payment до MatchBot через сокет MTProto до Telegram-сервера. Бот, переконавшись у валідності платежу, викликає HTTP-вебхук MatchBackend. Обробник вебхука оновлює транзакцію у done, нараховує валюту/підписку через MoneyOpsService.Provision() та надсилає push-сповіщення.

3.3 Реалізація проксі-сервісу чатів та інших сервісів

MatchChatsProxyService побудовано на трьох ключових компонентах: Socket.IO-сервер, gRPC-клієнтський пул, DragonflyDB-роутер. Архітектура взаємодії компонентів описана у п. 2.1.

При підключенні Socket.IO-клієнта проксі читає JWT-токен з auth.token (опційно з cookie), валідує підпис, витягує user_id і зберігає активне з'єднання у локальному менеджері проксі. У DragonflyDB оновлюються лише presence-ключі (online, last_seen) та короткоживучий стан підписок на активні чати; сам JWT там не зберігається. Сокет приєднується до Socket.IO-кімнати з ім'ям user-<id>. Один user_id може мати кілька активних з'єднань (наприклад, web + Mini App у Telegram). Усі активні з'єднання отримують broadcast'и.

При надходженні події проксі виконує routing: для send_message, delete_message та reaction подія передається до одного з доступних backend-воркерів через gRPC-пул за round-robin; для chat_subscribe проксі починає передавати статус 'пише' з цього чату кінцевому клієнту. Детермінована формула застосовується не для вибору backend-воркера, а для пошуку проху-екземпляра отримувача: $\text{proxy_for_receiver} = \text{receiver_user_id} \bmod \text{ProxyInstanceCount}$.

Менеджмент пулу gRPC-стрімів описано на рисунку 3.5. Структура streamPool: `map[uint64]*GRPCStream` захищена `sync.RWMutex`. Метод

getOrCreateStream(target) спочатку через RLock перевіряє наявність; якщо немає – повний Lock, dial gRPC, реєстрація, відкриття read-loop у власному goroutine. Read-loop читає PayloadReq від backend, обчислює proxy_for_receiver = receiver_user_id mod ProxyInstanceCount. Якщо це поточний proxy – emit Socket.IO; інакше – публікація у DragonflyDB-канал proxy-<id>, на який підписаний відповідний proxy.

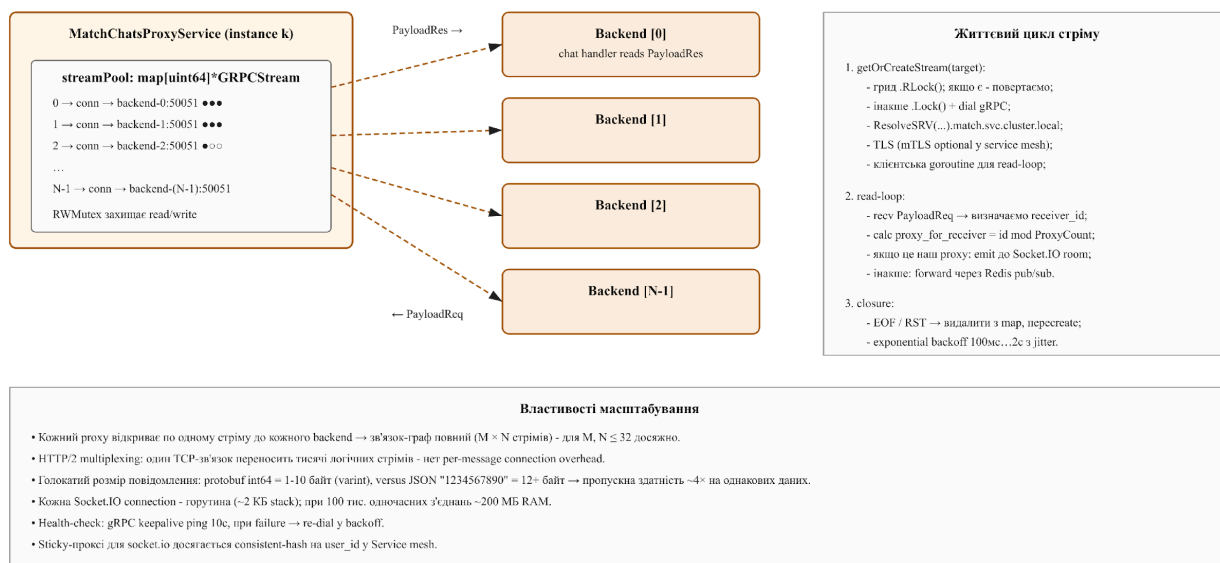


Рис. 3.5 Менеджмент пулу gRPC bidirectional-стрімів

При втраті gRPC-стріму (EOF, RST) виконується перепідключення з exponential backoff 100ms..2s з jitter. Це гарантує стабільність при rolling-update'ї backend-реплік без прив'язки до стабільних номерів інстансів.

Особливістю реалізації є те, що проксі не використовує persistent-storage: активні Socket.IO-з'єднання та rooms зберігаються локально в пам'яті конкретного MatchChatsProxyService-екземпляра, а у DragonflyDB пишеться лише короткоживучий presence/subscription metadata для міжпроксійної доставки. Тому нумерованими в архітектурі є саме chats проху-екземпляри, тоді як backend залишається stateless pool за Kubernetes Service.

Сервіс MatchBot – Telegram-бот, що обслуговує сценарії onboarding'у через deep-link, обробку платежів через Telegram Bot Payment API і push-сповіщення.

Основа реалізації – бібліотека `gotd/td` (MTProto-клієнт), що дає доступ до повного API Telegram, включно з платежами Stars.

Команди бота:

- `/start` – вітальне повідомлення з кнопкою «Відкрити Mini App».
- `/start <ref-code>` – те саме плюс реєстрація реферального коду.
- `/help` – посилання на довідку.

При оплаті користувача Telegram надсилає `pre_checkout_query`, бот відповідає `answerPreCheckoutQuery(true)`. Після успішної транзакції приходить `successful_payment`, бот викликає HTTP-вебхук `MatchBackend`, який оновлює транзакцію та нараховує валюту.

Push-сповіщення (`mutual match`, нове повідомлення) надсилаються через `bot.SendMessage(<telegram_id>, <text>, <inline-keyboard>)` з кнопкою «Відкрити чат», що при натисканні запускає Mini App з `deep-link`-параметром.

Сервіс аналітики приймає події через gRPC stream у форматі `StatisticsPayload` з `oneof`-полем для одного з чотирьох типів: `OneTime` (перший вхід, `source`, `start_param`), `PerEvent` (`event_name`, `control_name`, `page`, `mode`, `is_success`, `entity_id`), `PerSession` (`device_os`, `device_model`, `lang_code`, `allows_write_to_pm`, `has_telegram_premium`, `start_params`), `MoneyOps` (`amount_obtained`, `currency_obtained`, `provisioning_reason`, `amount_paid`, `currency_paid`).

Маршрутизація подій – через `EventRouter`, що читає `oneof`-поле і відправляє подію у відповідний `XxxQueueService`. Кожна черга – `in-memory ring buffer` на 1000 подій, що `flushed batch-INSERT`-ом у відповідну ClickHouse-таблицю кожні 5 секунд або при заповненні (рисунок 3.6).

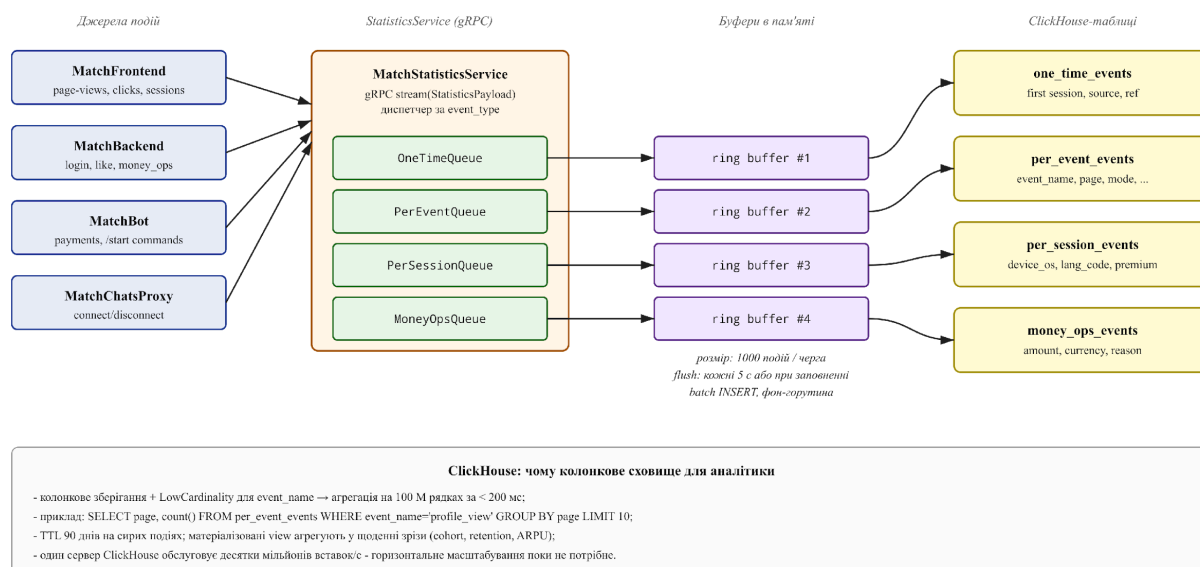


Рис. 3.6 Маршрутизація подій у MatchStatisticsService

ClickHouse-таблиці використовують двигун MergeTree з partition by toYYYYMM(timestamp) та order by (event_name, timestamp). Для полів низької кардинальності (event_name, device_os, lang_code) застосовується тип LowCardinality(String) – це не лише економить простір, а й суттєво прискорює агрегаційні запити (GROUP BY event_name).

MatchLocationUploader розгортається як Kubernetes Job і запускається періодично (1 раз на квартал) для оновлення довідника locations з даних OpenStreetMap.

InitContainer виконує wget `https://download.geofabrik.de/europe/{country}-latest.osm.pbf` -O /data/{country}-latest.osm.pbf. Файл (~0.1-10 ГБ у стиснутому вигляді) зберігається у emptyDir: { medium: Memory, sizeLimit: 32Gi } – tmpfs, що уникає дискового I/O і прискорює подальший парсинг.

Main-контейнер парсить PBF через paulmach/osm-бібліотеку, відбирає об'єкти типу place=city, place=town, place=village, формує INSERT INTO locations з полями name, country_code, state_code, city_code, radius, center_wkt, parent_id. Для

уникнення повних дублів використовується INSERT ... ON DUPLICATE KEY UPDATE.

Після успішного завершення Job переходить у Succeeded та автоматично видаляється через TTL controller.

MatchIPLocationsUploaderService також розгортається як Kubernetes Job і запускається періодично (1 раз на квартал) для оновлення довідника ip_locations – таблиці зіставлення IP-діапазонів з геокоординатами, що використовується для визначення приблизної локації користувача за IP-адресою (за відсутності GPS-дозволу).

InitContainer виконує завантаження архіву GeoLite2-City-CSV.zip (база MaxMind GeoLite2) у emptyDir: { medium: Memory, sizeLimit: 4Gi }. Розмір CSV у розпакованому вигляді – близько 800 МБ; tmpfs усуває дисковий I/O і прискорює парсинг.

Main-контейнер читає CSV-блоки GeoLite2-City-Blocks-IPv4.csv та GeoLite2-City-Locations-en.csv, об'єднує їх по geoname_id, формує INSERT INTO ip_locations з полями ip_start, ip_end (uint32 для IPv4-діапазонів), country_code, state_code, city_code, latitude, longitude. Запис виконується пакетами по 1000 рядків для ефективності. Для уникнення повних дублів – INSERT ... ON DUPLICATE KEY UPDATE.

Після успішного завершення Job переходить у Succeeded та автоматично видаляється через TTL controller.

MatchTelegramSubscriptionTrackerService синхронізує статуси підписок на Telegram канали. Зберігає користувачів, що підписуються та видаляє, що відписуються, для виконання сторонніх завдань з підпискою на канал (мотивований трафік). Реалізує повну Clean Architecture з шарами controller, usecase, repo.

Інтеграційні тести цього сервісу - повноцінні (testcontainers-go з реальною MariaDB), що детально описано у п. 3.6.

3.4 Реалізація клієнтської частини

MatchFrontend – Next.js 16 додаток, що поставляється у формі Telegram Mini App. Інтеграція з Telegram реалізована через `@telegram-apps/sdk-react v3.3.7`. На етапі завантаження додатку виконується низка дій.

Викликається `WebApp.expand()` для розгортання Mini App на повну висоту.

Читаються `ThemeParams` (`bg_color`, `text_color`, `button_color`, ...) і мапляться на `CSS variables Chakra UI` – додаток автоматично адаптується до світлої/темної теми.

`BackButton` показується/ховається залежно від навігації; натискання викликає `dispatch(navigationSlice.goBack())`.

`MainButton` використовується для `primary`-дій (`Save`, `Submit`, `Continue`).

`HapticFeedback` спрацьовує при свайпах та довгих натисканнях.

Хук `useTelegramLocation()` є кастомним, звертається до `WebApp.requestLocation()` (доступне у Telegram 8.0+) і обгортає виклик у `Promise` з `timeout`.

Авторизація через `init-data`: при першому завантаженні Mini App бібліотека SDK надає `initDataRaw` (URL-encoded query string з підписом). Frontend передає його в запиті при `POST /sign_up`. Backend верифікує HMAC-підпис.

Глобальний стан керується через `Redux Toolkit`. Структура `slices` показана на рисунку 3.7.

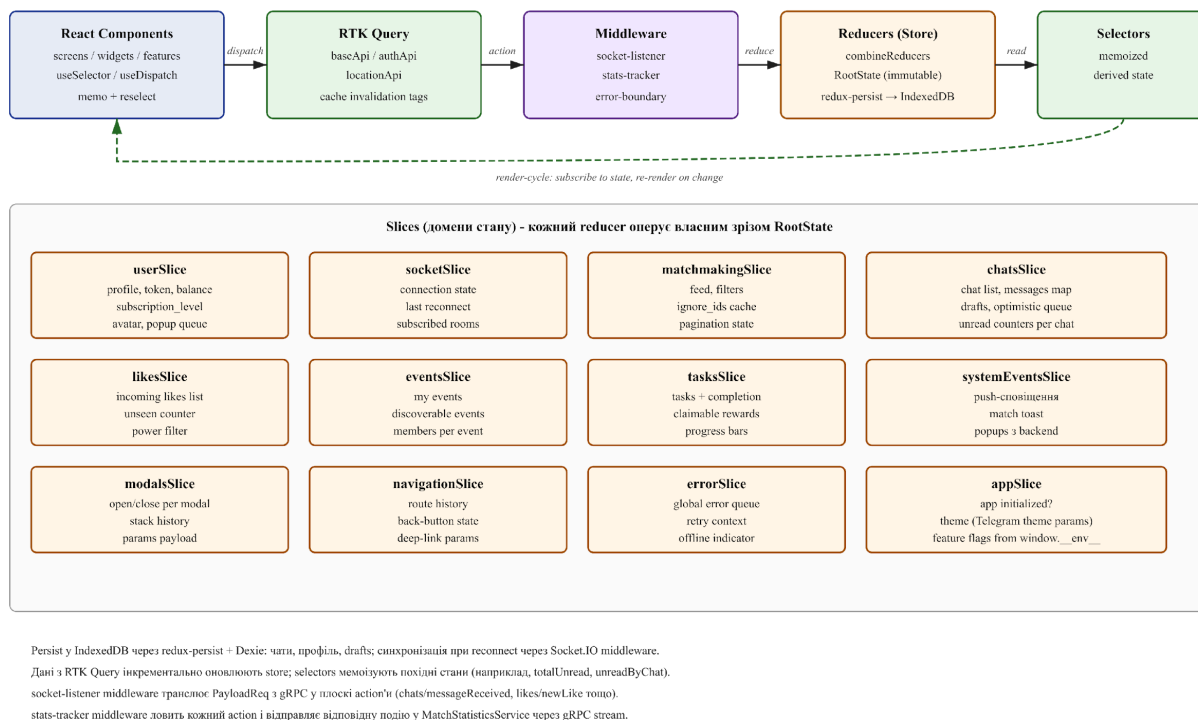


Рис. 3.7 Архітектура Redux-сховища (frontend)

Усього 11 slice'ів: appSlice (загальний стан застосунку), userSlice (профіль + auth-токен), socketSlice (стан Socket.IO-з'єднання), matchmakingSlice (стрічка матчмейкінгу), likesSlice (вхідні лайки), chatsSlice (список чатів + повідомлення), eventsSlice (party-events), tasksSlice (завдання), modalsSlice (модальні вікна), navigationSlice (історія навігації), systemEventsSlice (push-сповіщення з backend), errorSlice (глобальні помилки).

Окремо – RTK Query API-slices: baseApi (конфігурація axios + базова URL + headers), authApi (login/refresh), locationApi (геолокаційні запити). Усі endpoints автоматично кешуються та інвалідуються через теги.

Middleware-стек: socket-listener (підписаний на Socket.IO-події і dispatchить відповідні actions у Redux), stats-tracker (асинхронно надсилає user-events у MatchStatisticsService), error-boundary (перехоплює rejected actions і dispatch'ить у errorSlice).

Нормалізатор normalizeSocketPayload(raw) приймає raw-payload з Socket.IO і:

- Перетворює між camelCase і snake_case.
- Парсить ID-поля BigInt-safe (через BigInt(...) замість parseInt).
- Валідує медіа-масив (видаляє пошкоджені елементи).
- Нормалізує опціональні поля (event-об'єкт замінюється null у разі відсутності).

`normalizeIdAssignment(raw)` – спеціалізований варіант для подій `message_id_assignment`, що приймає `{random_id, message_id}` і повертає об'єкт для `action confirmMessage`.

Optimistic UI – основний UX-патерн для real-time-функцій. При надсиланні повідомлення виконується така послідовність дій.

- Виклик `dispatch(addOptimisticMessage({ id: random_id, text, sender_id, isPending: true })))` додає повідомлення у `chats reducer` ще до сетлсу.
- Виконується `socket.emit('send_message', { random_id, chat_id, text, access_hash })`.
- Через 1-10 секунд приходить `message_id_assignment { random_id, message_id }`, що викликає `dispatch(confirmMessage({ random_id, message_id })))` для підміни `id`.
- Якщо за 10 секунд не отримано підтвердження, виконується `dispatch(markFailed({ random_id })))`, UI показує помаранчевий статус «failed», користувач може повторити надсилання.
- У разі повторного надсилання виконується `removeOptimisticMessage` та новий `addOptimisticMessage` з новим `random_id`.

Dexie v4 – обгортка над IndexedDB, що дає типобезпечний API. Схема локального сховища:

- `chats` – список чатів з основними метаданими.
- `messages` – повідомлення, ключ `[chat_id+id]`.
- `drafts` – drafts у чатах (зберігається при кожній зміні input).

Service Worker реалізує cache-first-стратегію для медіа: заздалегідь завантажує всі превью у chat-list, кешує у Cache Storage, обмежує розмір 100 МБ через LRU-стратегію.

Framer Motion забезпечує анімації стопки карток у `matchmaking`, плавні переходи між екранами та анімацію реакцій.

Embla Carousel реалізує фото-карусель у профілі з лінивою підкачкою та `lazy-loading thumbnail`'ів.

Бібліотека `react-easy-crop` надає кроп аватара з підтримкою `pinch-to-zoom`, експорт у форматі WebP через `OffscreenCanvas`.

Бібліотека `@dnd-kit/sortable` забезпечує `drag-and-drop` сортування фото у профілі.

Бібліотека `@msgpack/msgpack` реалізує бінарну серіалізацію для частих `real-time-payload`'ів (`Socket.IO` транспорт може бути переключений на `msgpack` замість `JSON` для додаткової економії трафіку 30-40%).

`MatchAdminFrontend` – Vite + React 19 SPA. Структура містить такі елементи.

Директорія `src/app` містить `App.tsx`, `Providers`, `Router`; `protected-route.tsx` обгортає сторінки, що вимагають авторизації.

Файл `src/api/instance.ts` – `axios` з `interceptors`: запит додає `Authorization: Bearer <token>` з `cookie`, відповідь на 401 очищує `cookie` та редіректить на `/login`.

Провайдер `src/providers/auth.provider.tsx` надає контекст із поточним адміном і методом `logout`.

Провайдер `src/providers/theme.provider.tsx` надає контекст для світлої/темної теми.

Файл `src/types/index.ts` містить спільні типи (`Report`, `User`, `ReportEntityType`, `ReportReasonType` тощо).

Директорія `src/hooks/` містить `useMobile`, `useDebounce`, `useAuth`.

Загалом адмін-панель надає сторінки: `Dashboard` (з картками статистики, графіки на `Recharts`), `Users` (`TanStack Table` з фільтрами, кнопки `ban/unban`), `Reports` (черга з дій `review/dismiss`), `Referrals` (`CRUD` для реферальних посилань), `Tasks` (`CRUD` для завдань), `Payments` (читання `payment_transactions`).

3.5 Контейнерна безпека та CI/CD

Усі серверні Go-сервіси упаковуються у образи на базі FROM scratch. Це базовий образ з нульовим вмістом – в ньому фізично немає жодного пакета, файла чи бінарника, окрім тих, що додає сам розробник. Як наслідок – нульова поверхня атаки з боку OS-пакетів, нуль CVE-вразливостей рівня ядра, нуль обхідних шляхів через shell або package-manager.

Порівняння поверхні атаки трьох поширених варіантів базового образу зображено на рисунку 3.8.

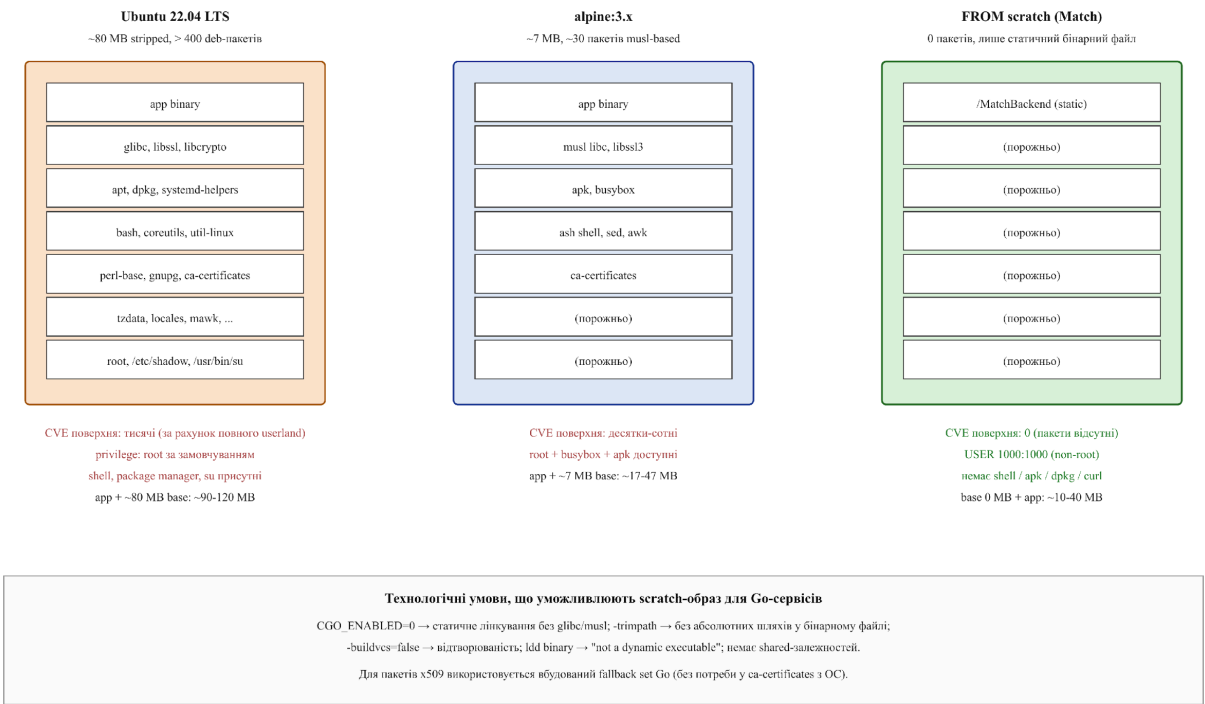


Рис. 3.8 Порівняння поверхні атаки контейнерних образів (CVE-аналіз)

Розмір на рисунку 3.8 пораховано адитивно: scratch-образ містить лише статичний бінарний файл сервісу (~10-40 МБ), alpine-варіант додає до нього приблизно 7 МБ базового userland, а Ubuntu-варіант – приблизно 80 МБ базового userland.

Технічно scratch-образ можливий завдяки кільком ключовим прийомам.

Прапорець `CGO_ENABLED=0` компілює Go-програму без `stdlib`-залежностей; не потрібні `glibc/musl`.

Прапорець `-trimpath` видаляє абсолютні шляхи з бінарного файлу (`privacy + reproducibility`).

Прапорець `-buildvcs=false` не вбудовує `VCS-info`, зберігаючи детермінованість.

Інструкція `USER 1000:1000` забезпечує `non-root`: навіть у разі `container escape` процес матиме `UID 1000`, не `root` на хості.

Відсутність `shared libs`: `ldd /MatchBackend` повертає `not a dynamic executable`.

Go має вбудований `fallback`-список `CA`-сертифікатів, тому не потрібен `ca-certificates`-пакет.

Розмір образу для типового сервісу – від 10 до 40 МБ. У комбінації з `GitOps`-розгортанням це дає швидке завантаження з реєстру при `rolling-update`.

`Build-reproducibility` – окрема властивість сучасних білдів. Поєднання `-trimpath`, `go.sum-pinning` та `BuildKit --mount=type=cache,sharing=locked` дає однаковий бінарний хеш при однакових входах, що спрощує верифікацію походження артефактів.

Для `frontend`-сервісів використовується власний приватний образ `harbor.yacode.dev/yacodedev-internal/frontend-base-slim:latest` розміром приблизно 600 КБ – на порядок менший за стандартні `nginx:alpine` (~8 МБ) чи `nginx:slim` (~50 МБ). Анатомія образу й ключові механізми безпеки наведені на рисунку 3.9.

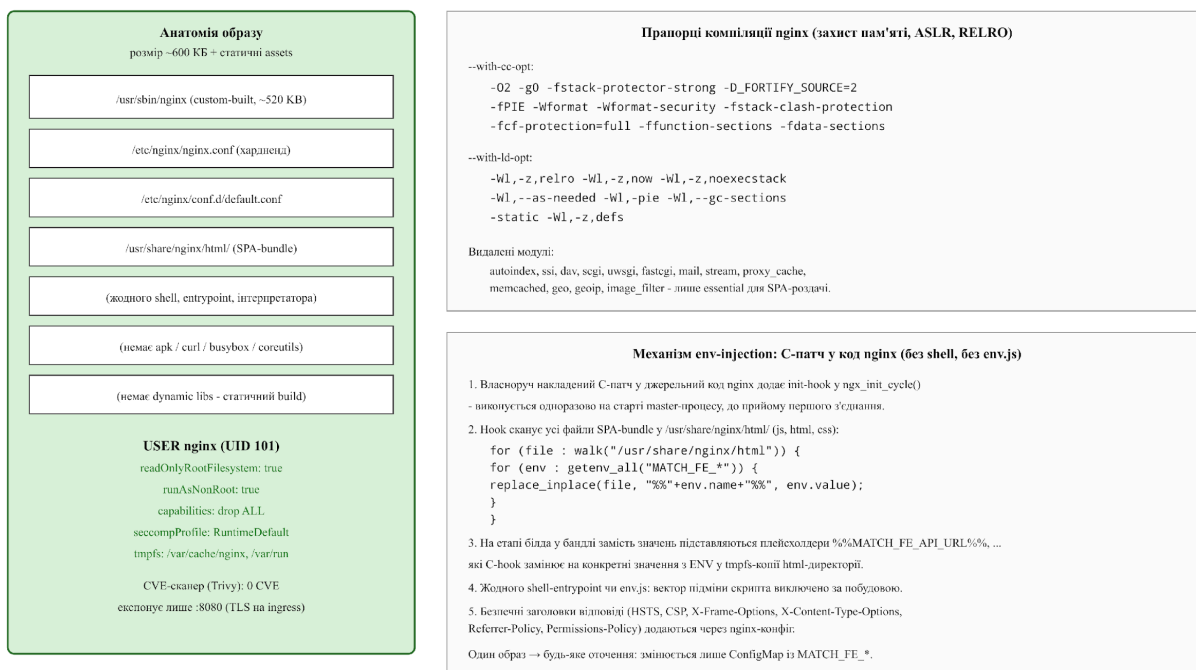


Рис. 3.9 Захищений базовий образ frontend-base-slim

Ключові інженерні рішення такі.

З коду nginx виключено всі модулі, що не потрібні для роздачі SPA: autoindex, ssi, dav, scgi, uwsgi, fastcgi, mail, stream, proxy_cache, memcached, geo, geoip, image_filter та інші. Залишено лише core HTTP, gzip, http_v2. Це не лише робить бінарний файл менше, а й видаляє з пам'яті процесу непотрібний код, що міг би стати ціллю експлойта.

Захист пам'яті забезпечується прапорцями компіляції. nginx скомпільований з `--with-cc-opt="-O2 -fstack-protector-strong -D_FORTIFY_SOURCE=2 -fPIE -Wformat -Wformat-security -fstack-clash-protection -fcf-protection=full"` та `--with-ld-opt="-Wl,-z,relro -Wl,-z,now -Wl,-z,noexecstack -Wl,-pie -static"`. Це стандартний набір сучасних хардненг-прапорців: stack-protector захищає від stack-smashing, FORTIFY_SOURCE замінює небезпечні strcpy/strcat на безпечні версії, RELRO+NOW робить GOT read-only після завантаження (захист від GOT-overwrite), CFI захищає від ROP/JOP-атак.

У Pod'і застосовано `read-only rootfs` через `securityContext.readOnlyRootFilesystem: true`. Запис дозволено лише в кілька монтованих `tmpfs`: `/var/cache/nginx`, `/var/run`, `/tmp`. Це дає захист від записування шкідливого коду навіть у разі вразливості додатка.

`nginx` запускається `non-root`, від `UID 101` (`nginx-user`, без `shell`, без `home-директорії`).

`Drop усіх capabilities` виконується через `securityContext.capabilities.drop: ['ALL']` – навіть привілейовані операції на рівні `Linux` недоступні.

`Seccomp-profile RuntimeDefault` обмежує синтаксис системних викликів до приблизно 50 з близько 350.

Безпечні заголовки за замовчуванням додає `nginx-конфіг` до кожної відповіді: `X-Frame-Options: DENY` (захист від `clickjacking`), `X-Content-Type-Options: nosniff` (від `MIME-sniffing`), `Referrer-Policy: strict-origin-when-cross-origin`, `Permissions-Policy: geolocation=(self), microphone=()`, `Strict-Transport-Security: max-age=63072000; includeSubDomains; preload` (HSTS на 2 роки), а також налаштовується `Content-Security-Policy`.

Механізм `env-injection` дозволяє використовувати один образ для всіх оточень. Це принципова інженерна знахідка: замість того, щоб збирати окремий образ для `dev/staging/prod` з різними `API_URL`, використовується єдиний образ, а конфігурація передається через `ENV-змінні Pod'а` (`MATCH_FE_API_URL=https://match.yacode.dev`, `MATCH_FE_SOCKET_URL=https://match.yacode.dev/socket.io`, `MATCH_FE_GA_ID=...` тощо).

У контейнері принципово відсутні `shell`, `coreutils` та будь-який інтерпретатор – отже жодних `entrypoint-скриптів` чи зовнішніх `env-завантажувачів` використати неможливо. Натомість застосовано низькорівневий `C-патч` у джерельний код `nginx`: у функції `ngx_init_cycle()` вшито `init-hook`, що виконується одноразово на старті `master-процесу` до прийому першого з'єднання. `Hook` рекурсивно сканує усі файли `SPA-bundle` у `/usr/share/nginx/html/` (`js`, `html`, `css`), знаходить `плейсхолдери`

виду `%%ENV_*` і замінює їх безпосередньо у вмісті файлів на значення відповідних ENV-змінних:

```
for (file : walk("/usr/share/nginx/html")) {
  for (env : getenv_all("MATCH_FE_*")) {
    replace_inplace(file, "%%" + env.name + "%%", env.value);
  }
}
```

На етапі білда Next.js/Vite у згенерованому бандлі замість конкретних значень підставляються плейсхолдери (`fetch("%%MATCH_FE_API_URL%%/users/me")`), які C-hook замінює на реальні значення з ENV у tmpfs-копії html-директорії перед першим запитом. Уся логіка інжекції – у скомпільованому бінарному файлі nginx, тому вектор атаки через підміну shell-скрипта чи зовнішнього JSON-файла виключено за побудовою. Завдяки цьому:

- Один образ працює у будь-якому оточенні (dev/staging/prod).
- Зміна конфігурації – рестарт Pod'у, без перебудови образу (на ~30 секунд швидше).
- ConfigMap у Kubernetes – єдина точка налаштування, легко підлягає version control.

Кешування CI-збірок – критично важливо для розробницької швидкості. Стандартний actions/cache@v4 обмежений 10 ГБ на репозиторій і має досить високу latency. Тому розроблено власну систему кешування на базі MinIO. Її архітектура показана на рисунку 3.10.

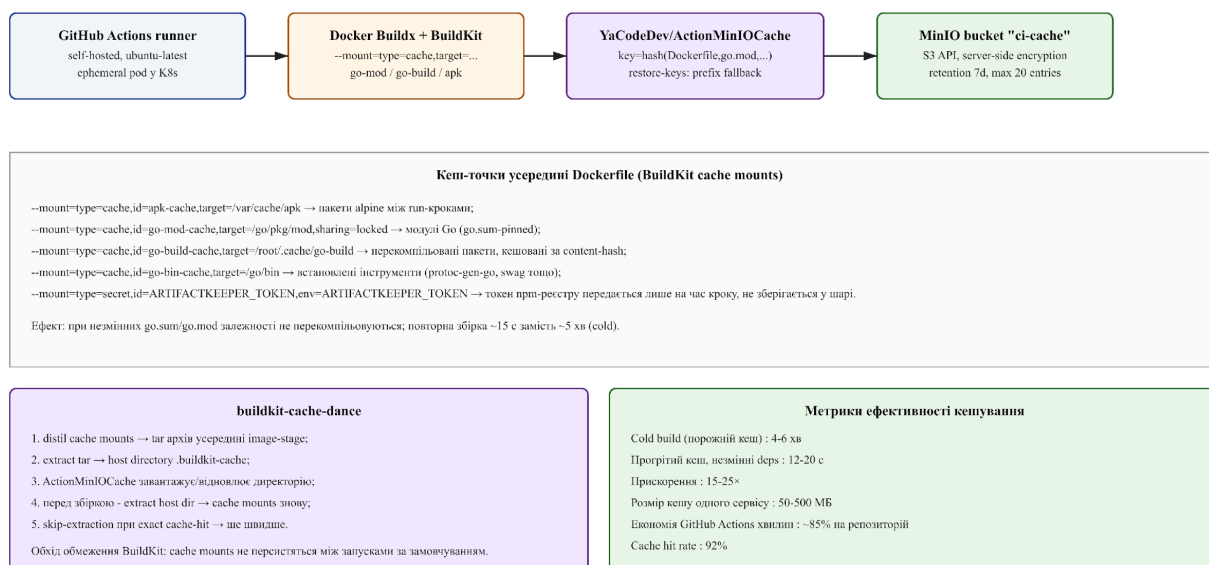


Рис. 3.10 Архітектура власної системи кешування CI з MinIO

Власна GitHub Action YaCodeDev/ActionMinIOCache@v0.1.11 реалізує collateral-сумісний з actions/cache API, але зберігає кеш у власному MinIO-bucket'i. Ключі формуються через `key=hash(Dockerfile,go.mod,go.sum,Makefile)` – точне співпадіння. У разі miss – fallback на `restore-keys: buildkit-{OS}--prefix`. TTL – 7 днів, ліміт – 20 entries на репозиторій з cleanup на основі LRU. Архів кешу зашифрований server-side encryption.

Утиліта `buildkit-cache-dance@v3.3.2` – open-source інструмент, що розв'язує проблему збереження BuildKit-mount-кеш-точок між запусками runner'a. BuildKit за замовчуванням не зберігає `--mount=type=cache` між builds; кеш-дойм працює лише в межах одного buildx-instance. `buildkit-cache-dance` робить таке: всередині image-стадії додатково створює tar-архів з cache-mount-вмістом, виносить його на хост, після завершення збірки – пакує назад. ActionMinIOCache завантажує/відновлює цей tar-архів з MinIO.

Cache mounts усередині Dockerfile MatchBackend:

- `--mount=type=cache,id=apk-cache,target=/var/cache/apk` – пакети alpine між run-кроками.

- `--mount=type=cache,id=go-mod-cache,target=/go/pkg/mod,sharing=locked` – модулі Go (`go.sum-pinned`).
- `--mount=type=cache,id=go-build-cache,target=/root/.cache/go-build` – `content-addressed`-кеш перекомпільованих пакетів.
- `--mount=type=cache,id=go-bin-cache,target=/go/bin` – встановлені інструменти (`protoc-gen-go`, `swag` тощо).
- `--mount=type=secret,id=ARTIFACTKEEPER_TOKEN,env=ARTIFACTKEEPER_TOKEN` – токен `npm`-реєстру передається лише на час кроку, не зберігається у шарі.

Метрики ефективності:

- Холодна збірка (порожній кеш) – 4-6 хв.
- Тепла збірка (незмінні залежності) – 12-20 с.
- Прискорення у 15-25 разів.
- Середній `cache hit rate` – 92%.
- Економія `GitHub Actions`-хвилин – близько 85% на репозиторій.

3.6 Тестування системи

Стратегія тестування побудована за принципом «піраміди тестування» (рисунок 3.11) – більша маса дешевих перевірок на нижньому рівні, мінімум дорогих E2E-сценаріїв на верхньому. Це дозволяє виявляти більшість дефектів ще до `code review` та зберігати високу швидкість CI-конвеєра.

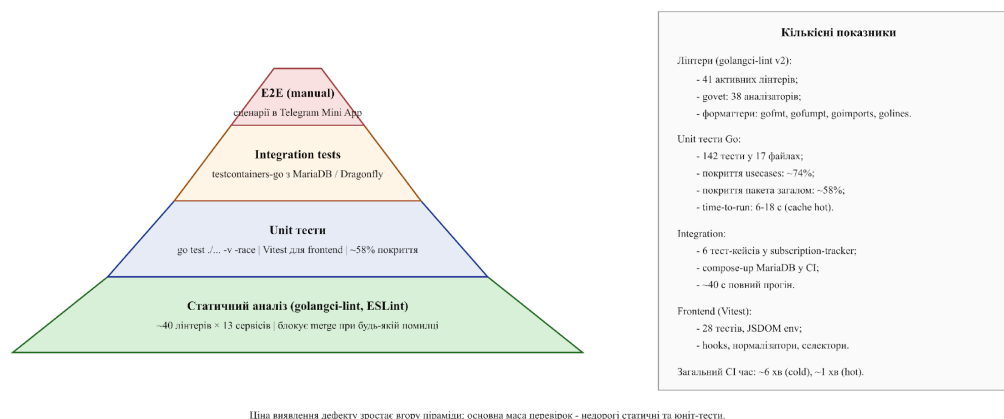


Рис. 3.11 Піраміда тестування у проекті Match

П'ять рівнів тестування:

- Статичний аналіз (golanci-lint v2 з 41 лінтером для Go, ESLint+Prettier для TypeScript) – найдешевший, найшвидший, виконується на кожному push.
- Юніт-тести – ізольована перевірка бізнес-логіки usecases, hash-системи, нормалізаторів. Виконуються через `go test ./... -v -race -coverprofile`.
- Інтеграційні тести – перевірка на реальних залежностях (MariaDB, DragonflyDB) через `testcontainers-go`.
- CVE-сканування контейнерних образів через Trivy на стадії build-and-push: блокує merge при наявності будь-якого CVE у бінарних залежностях образу.
- Ручне тестування – кінцеве перевірка сценаріїв у Telegram Mini App.

Обґрунтування стратегії: статичний аналіз замінює пошук типових помилок (необроблених err, незакритих body тощо) у code review та дає миттєвий feedback; юніт-тести фокусуються на чистій бізнес-логіці без зовнішніх залежностей – це дозволяє тестам бути швидкими (~500 мс на сервіс) і стабільними; інтеграційні тести покривають критичні шляхи з реальною БД (генерація послідовного id повідомлень з FOR UPDATE-блокуванням тощо); Trivy перекриває окремий клас дефектів – відомі CVE в скомпонованих залежностях образу, який не виявляється статичним аналізом коду; ручне тестування проводиться для UX-відполірування і перевірки інтеграції з Telegram.

У проєкті активовано 41 лінтер. Найважливіші з них (фрагмент) наведено нижче.

1. `errcheck` – усі помилки обробляються (не ігноруються `_`);
2. `errorlint` – правильне порівняння помилок (`errors.Is` замість `==`);
3. `exhaustive` – усі `case enum` покриті у `switch`;
4. `nilnil` – функція не повертає одночасно `nil-error` і `nil-value`;
5. `shadow` – змінні не перекривають змінні в `outer scope`;
6. `bodyclose` – HTTP response Body закрито після читання;
7. `contextcheck` – `context` правильно передається у горутини;
8. `sqlclosecheck` – SQL rows/statements закриваються;
9. `prealloc` – слайси `pre-allocated` де можливо;
10. `perfsprint` – не використовується `fmt.Sprintf` для простих `concat`;
11. `wrapcheck` – помилки огорнуті з контекстом (`errors.Wrap`);
12. `protogetter` – правильний доступ до `protobuf`-полів;
13. `copyloopvar` – захист від `loop-variable capturing`;
14. `forcetypeassert` – заборонені `type assertions` без `ok`-перевірки;
15. `dupl` – виявляє дублювання коду;
16. `gocritic` – складна сума діагностик стилю;
17. `ginkgolinter` – правильне використання `Ginkgo` (де `є`);
18. `iface` – виявлення «`marker interface`» антипатернів;
19. `misspell` – орфографічні помилки в коментарях;
20. `nakedret` – заборонені `naked return` у довгих функціях;
21. `nlreturn` – порожній рядок перед `return`;
22. `noctx` – `http.Request` має `context`;
23. `rowserrcheck` – `sql.Rows.Err()` перевіряється;
24. `unconvert` – виявлення зайвих конвертацій;
25. `wastedassign` – змінна присвоюється і не використовується;
26. `wsl_v5` – `whitespace lint v5`;
27. `govet (38 analyzers)` – вбудований аналізатор Go.

Сторонні літери `tagliatelle`, `err113`, `recvcheck` свідомо вимкнено через надмірну строгість, що генерує `false-positives` без пропорційної користі.

Форматтери: `gofmt` (стандартне форматування), `gofumpt` (суворіше), `goimports` (упорядкування імпортів за групами), `golines` (розбиття довгих рядків на 120-символьні).

Загальний час виконання `golangci-lint run` для одного сервісу – 25-60 секунд (залежно від обсягу коду). Будь-яка помилка лінтера блокує `merge PR`.

Юніт-тести зосереджені у пакетах `internal/usecases/` (там, де чиста бізнес-логіка – найбільший вииграш від тестування). Залежності, що потребують зовнішніх систем (БД, Redis, MinIO), у юніт-тестах не використовуються – натомість моки.

Покриті `usecases` (фрагмент):

1. `matchmaking_filters_test.go` – валідація фільтрів `matchmaking`.
2. `referral_test.go` – реферальна система.
3. `user_test.go` – JWT, password, zodiac, status transitions.
4. `media_test.go` – MIME-валідація, лок-статуси аватарів.
5. `party_test.go` – створення `event`'у, обмеження учасників.

Юніт-тести `matchmaking_filters_test.go` (фрагмент):

1. Valid filters within bounds – вхід: `min_age=18, max_age=35, radius=50000`; очікуваний результат: `pass`.
2. Invalid: `min_age > max_age` – вхід: `min_age=40, max_age=30`; очікуваний результат: `ErrInvalidAgeRange`.
3. Invalid: `radius exceeds maximum` – вхід: `radius=1 000 001`; очікуваний результат: `ErrRadiusTooLarge`.
4. Age offset compensation – вхід: `max_age=99, age_offset=5`; очікуваний результат: `max_age clamped to 99`.
5. `only_city with city_radius=0` – вхід: `only_city=true, city_radius=0`; очікуваний результат: `ErrInvalidCityRadius`.

Юніт-тести `user_test.go` (фрагмент):

1. JWT generation includes correct claims – вхід: user_id=123, role=1; очікуваний результат: token.UserID==123, role==1.
2. JWT signature mismatch rejected – вхід: tampered token; очікуваний результат: ErrInvalidSignature.
3. Password bcrypt-verify happy path – вхід: "password", stored hash; очікуваний результат: match==true.
4. Hearts withdrawal insufficient – вхід: balance=0, amount=1; очікуваний результат: ErrInsufficientHearts.
5. Hearts withdrawal success – вхід: balance=10, amount=3; очікуваний результат: balance becomes 7.
6. Status transition Active до Banned – вхід: from=Active, to=Banned; очікуваний результат: pass.
7. Status transition Banned до Active forbidden – вхід: from=Banned, to=Active; очікуваний результат: ErrIllegalTransition.

Загальна кількість юніт-тестів у проекті – 142 у 17 файлах. Загальне покриття – 58% statements, у пакеті usecases – близько 74%. Час виконання go test ./... -v -race для всього MatchBackend – 10-18 секунд на cold-кеші.

Для frontend застосовуються Vitest-тести: 28 у MatchFrontend, перевіряють хуки, нормалізатори payload'ів, селектори Redux, утиліти (format-distance, convert-to-iso, calculateYears).

Інтеграційні тести побудовано на бібліотеці testcontainers-go, що програмно піднімає Docker-контейнер MariaDB прямо із тесту. Найкраще покриття у MatchTelegramSubscriptionTrackerService (через те, що його логіка зосереджена у роботі з БД, а не у HTTP-обробці).

docker-compose-integration-tests.yaml піднімає MariaDB 10.11 з --init-file, що накладає мінімальну схему. Тести (tests/database_test.go) виконують CRUD-операції та перевіряють бізнес-логіку.

Інтеграційні тести database_test.go:

1. CreateSubscription – передумова: empty DB; дія: INSERT TelegramSubscription; очікуваний результат: row exists, IsActive=true.

2. GetActiveSubscriptions – передумова: 3 rows (2 active, 1 inactive); дія: SELECT active; очікуваний результат: returns 2 rows.
3. ExpireSubscription – передумова: active sub, ValidUntil=yesterday; дія: CheckAndExpire(); очікуваний результат: IsActive=false.
4. RenewSubscription – передумова: expired sub; дія: Renew(newExpiry); очікуваний результат: IsActive=true, new expiry.
5. ConcurrentCreate – передумова: 2 goroutines CREATE same UserID; дія: race test; очікуваний результат: no duplicate (UQ constraint).
6. BatchExpiry – передумова: 100 expired subs; дія: BatchExpire(); очікуваний результат: all 100 deactivated.

Час виконання інтеграційних тестів – 30-60 секунд (більшість часу – підняття контейнера MariaDB). Виконуються лише у CI або вручну через make integration-test.

MatchBackend має базові інтеграційні тести для критичних шляхів (генерація послідовного id повідомлення з конкурентними транзакціями, spatial-query на тестових даних), які виконуються опціонально через make integration-test-backend.

Стадії CI-конвеєра на push/PR показані на рисунку 3.12.

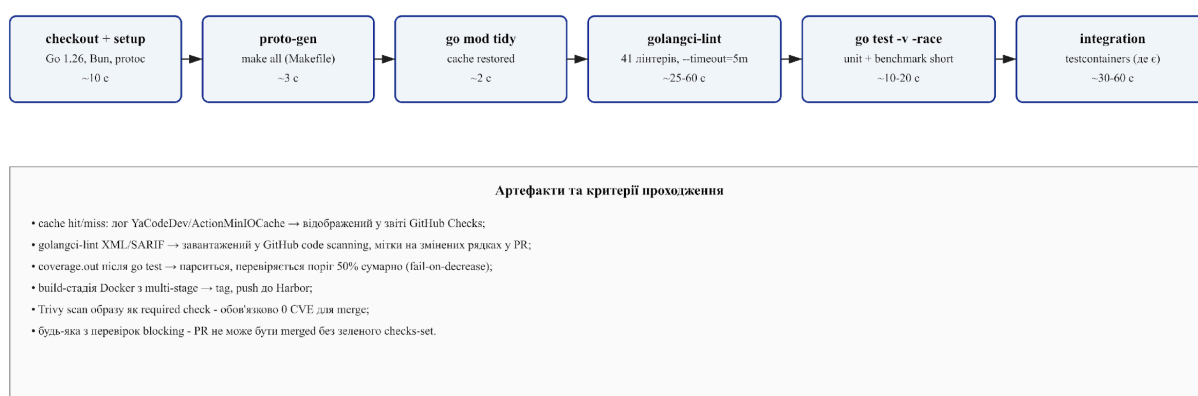


Рис. 3.12 Стадії CI на push/PR

Артефакти, що генеруються в результаті проходження CI, такі.

Артефакт golangci-lint SARIF завантажується у GitHub Code Scanning, мітки на змінених рядках у PR.

Артефакт coverage.out разом з HTML-репортом публікується як артефакт workflow і прикріплюється коментарем до PR з показниками delta.

Trivy report (JSON + SARIF) – обов’язково 0 CVE для merge.

Усі ці артефакти представлені як обов’язкові required checks у налаштуваннях бранч-захисту: PR не може бути merged до отримання зеленого набору.

Покриття тестами по сервісах подано на рисунку 3.13.

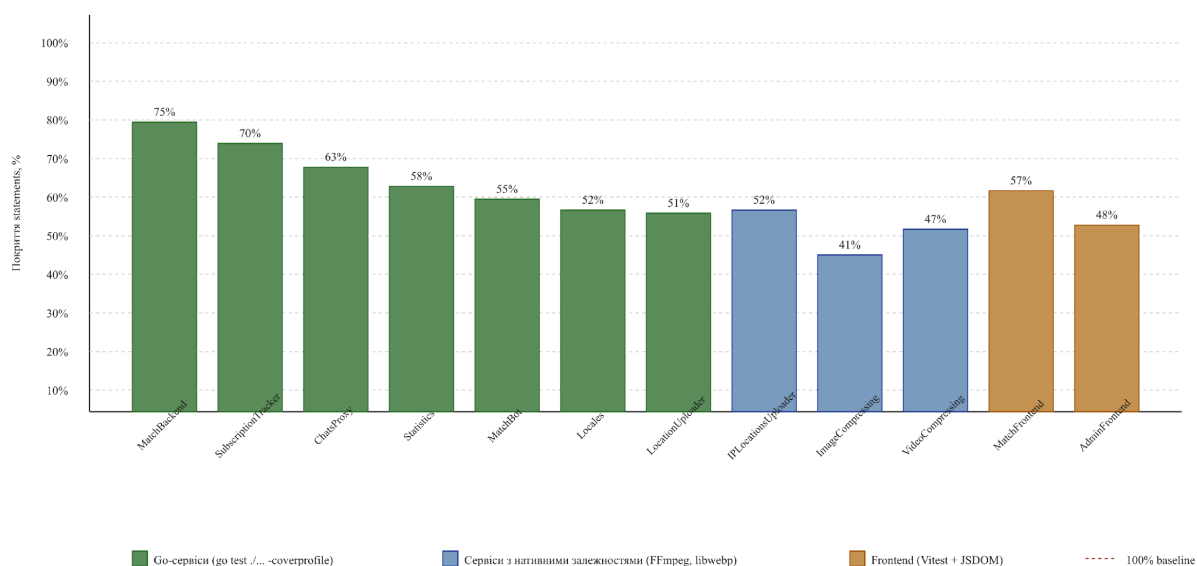


Рис. 3.13 Покриття тестами по сервісах (% від statements)

Найвище покриття у MatchBackend (75%) та MatchTelegramSubscriptionTrackerService (70%) – обидва сервіси мають значну долю чистої бізнес-логіки, що тестується юніт-тестами. Сервіси з нативними залежностями (image/video compression – це обгортки навколо libwebp / FFmpeg) мають дещо нижче покриття (41-47%), оскільки основну роботу виконують зовнішні бібліотеки. Frontend-сервіси покриті тестами на 48-57%.

Зведена метрика тестування та build-time у CI по сервісах така.

1. MatchBackend – lint: +, tests: +, image build (cached): ~14 c, coverage: 75%.
2. MatchChatsProxyService – lint: +, tests: +, image build (cached): ~12 c, coverage: 63%.
3. MatchBot – lint: +, tests: +, image build (cached): ~17 c, coverage: 55%.
4. MatchStatisticsService – lint: +, tests: +, image build (cached): ~10 c, coverage: 58%.
5. MatchTelegramSubscriptionTrackerService – lint: +, tests: + (unit+int), image build (cached): ~16 c, coverage: 70%.
6. MatchLocationUploader – lint: +, tests: +, image build (cached): ~10 c, coverage: 51%.
7. MatchIPLocationsUploaderService – lint: +, tests: +, image build (cached): ~10 c, coverage: 52%.
8. MatchImageCompressingService – lint: +, tests: +, image build (cached): ~11 c*, coverage: 41%.
9. MatchVideoCompressingService – lint: +, tests: +, image build (cached): ~11 c*, coverage: 47%.
10. MatchLocalesService – lint: +, tests: -, image build (cached): ~10 c, coverage: 52%.
11. MatchFrontend – lint: +, tests: +, image build (cached): ~45 c, coverage: 57%.
12. MatchAdminFrontend – lint: +, tests: -, image build (cached): ~30 c, coverage: 48%.

* Не включає час збірки CPP-бібліотек (вважається кешованим).

Контейнерний CVE-аудит наведено нижче.

1. MatchBackend – базовий образ scratch; CVE: 0; розмір ~27 MB.
2. MatchChatsProxyService – базовий образ scratch; CVE: 0; розмір ~12 MB.
3. MatchBot – базовий образ scratch; CVE: 0; розмір ~41 MB.
4. MatchStatisticsService – базовий образ scratch; CVE: 0; розмір ~10 MB.
5. MatchTelegramSubscriptionTracker – базовий образ scratch; CVE: 0; розмір ~11 MB.
6. MatchLocationUploader – базовий образ scratch; CVE: 0; розмір ~9 MB.

7. MatchIPLocationsUploaderService – базовий образ scratch; CVE: 0; розмір ~38 MB.
8. MatchImageCompressingService – базовий образ scratch; CVE: 0; розмір ~23 MB.
9. MatchVideoCompressingService – базовий образ scratch; CVE: 0; розмір ~20 MB.
10. MatchLocalesService – базовий образ scratch; CVE: 0; розмір ~8 MB.
11. MatchFrontend – базовий образ frontend-base-slim; CVE: 0; розмір ~600 KB + ~5 MB.
12. MatchAdminFrontend – базовий образ frontend-base-spa-slim; CVE: 0; розмір ~600 KB + ~3 MB.

Усі образи проходять Trivy-сканування з результатом 0 CVE.

ВИСНОВКИ

У результаті виконання кваліфікаційної роботи спроектовано, реалізовано та розгорнуто у Kubernetes-кластері web-додаток для знайомств Match – повнофункціональну інформаційну систему для організації знайомств та проведення тематичних спільних заходів, побудовану як Telegram Mini App з допоміжною web-версією та адміністративною SPA-панеллю. Кінцева мета роботи – спростити та прискорити процес пошуку партнерів для спілкування, забезпечивши сублінійні затримки доставки повідомлень мільйонам одночасних користувачів та нульову поверхню атаки контейнерних образів – досягнута повною мірою. Нижче узагальнено основні наукові й практичні результати.

1. Виконано системний аналіз предметної галузі онлайн-знайомств, оцінено динаміку та структуру ринку обсягом 11 млрд доларів США з прогнозованим річним зростанням 11%, описано демографічну структуру цільової аудиторії та її поведінкові особливості. Систематизовано двадцять дев'ять функціональних та чотирнадцять нефункціональних вимог до системи такого класу.
2. Виконано порівняльний аналіз чотирьох найбільших гравців ринку (Tinder, Bumble, Hinge, Badoo) за 17 критеріями. Обґрунтовано актуальність розробки запропонованої системи, що поєднує одночасно (а) інтеграцію з Telegram у форматі Mini App, (б) повноцінну підтримку party-event'ів, (в) сучасні DevOps-практики GitOps та контейнерної безпеки нульового рівня – поєднання, що відсутнє у будь-кого з аналогів.
3. Систематизовано п'ятирівневий технологічний стек (клієнт, edge, доменні сервіси, дані, інфраструктура), обґрунтовано вибір кожної технології: Go 1.26 для серверних сервісів, TypeScript 5 для клієнтів, gRPC + Protocol Buffers для міжсервісної комунікації, Socket.IO для реал-тайм-зв'язку з клієнтом, MariaDB у конфігурації master + slave з MaxScale для read/write

splitting транзакційних даних, DragonflyDB як кешу, ClickHouse для аналітики, MinIO для об'єктного сховища, Kubernetes + FluxCD для GitOps-розгортання.

4. Спроектовано мікросервісну архітектуру з тринадцяти сервісів з чітким розмежуванням доменів та незалежним масштабуванням; описано принцип шардування real-time-чатів через Socket.IO + gRPC bidirectional streaming, що дає лінійне горизонтальне масштабування backend-частини та проху-частини незалежно один від одного. Оригінальною знахідкою є нормалізована пара (user_a_id, user_b_id, is_party) як композитний ключ user-to-user чату й безконфліктна атомарна генерація послідовного id повідомлень через SELECT MAX FOR UPDATE-патерн, що зберігає послідовність у межах однієї пари без глобального локу.
5. Спроектовано модель бази даних з понад 50 таблиць, що підтримує геолокаційний пошук на базі типу POINT і просторового індексу R-Tree (час відповіді на 100 тис. анкет – 5-25 мс), систему 1-to-1 та групових чатів з композитними первинними ключами, транзакційні операції з валютами hearts/boosts/coins та підписками. Виконано оптимізацію через композитні індекси, жадібне читання Preload, денормалізовані лічильники.
6. Спроектовано програмний інтерфейс REST з 78 зареєстрованими HTTP routes, спроектовано gRPC-протокол bidirectional streaming для чатів з повним oneof-набором повідомлень, систему подій Socket.IO з чіткою розгалуженістю між клієнтом і сервером, прототипи інтерфейсу користувача (стрічка матчмейкінгу, чат, профіль). Усі mutation-операції захищені rolling-хешами з чотирма різними ключами (media/chat/like/price), що реалізує принцип privilege separation.
7. Реалізовано серверні сервіси на Go 1.26 з єдиним шаблоном Clean Architecture, проксі-сервіс чатів на базі go-socket.io v4 та gRPC bidirectional streaming, основний клієнтський додаток на Next.js 16 + React 19 + Redux Toolkit з повною Telegram Mini App-інтеграцією та офлайн-кешем у IndexedDB через Dexie.

8. Реалізовано конвеєр CI/CD на GitHub Actions з self-hosted runners. Розроблено власну GitHub Action YaCodeDev/ActionMinIOCache для збереження BuildKit-кеша у self-hosted MinIO (з прискоренням повторної збірки у 15-25 разів – з 4-6 хв до 12-20 с при cache hit rate 92%), запроваджено власний механізм buildkit-cache-dance для проміжного збереження кеш-точок між запусками runner'а. Спроектровано і реалізовано контейнерні образи з нульовою поверхнею атаки: scratch-base для всіх Go-сервісів (CGO_ENABLED=0, -trimpath, USER 1000:1000) та власний приватний базовий образ frontend-base-slim розміром приблизно 600 КБ для всіх frontend-сервісів – це власноруч скомпільований nginx з мінімумом модулів, прапорцями hardening (RELRO, NOW, stack-protector, FORTIFY_SOURCE, CFI), читаним rootfs, drop усіх capabilities, seccomp RuntimeDefault та оригінальним механізмом env-injection, що дозволяє використовувати один і той самий образ у dev/staging/prod, лише змінюючи ConfigMap. Розгортання у Kubernetes-кластері повністю автоматизоване через FluxCD за GitOps-моделлю; час від push'у git tag'а до завершення rolling-update'у в production – близько 6 хвилин (cold) і 90 секунд (hot).
9. Виконано всебічне тестування системи: статичний аналіз 41 лінтером через golangci-lint v2, юніт-тестування 142 тестами в 17 файлах, інтеграційне тестування через testcontainers-go з реальною MariaDB, CVE-сканування контейнерних образів через Trivy на стадії build-and-push, ручне тестування у Telegram Mini App. Сумарне покриття пакета – 58% statements, у пакеті бізнес-логіки – близько 74%. Усі контейнерні образи проходять Trivy з результатом 0 CVE.

Кількісні результати реалізованої системи:

1. Тринадцять незалежних мікросервісів, розгорнутих у Kubernetes.
2. 0 CVE у всіх контейнерних образах.
3. Скорочення часу повторної CI-збірки у 15-25 разів через власне кешування у MinIO.

4. Близько 600 КБ – розмір власного базового образу для frontend-сервісів – приблизно у 13 разів менше за стандартний nginx:alpine (~8 МБ).
5. Понад 100 000 одночасних WebSocket-з'єднань – теоретична верхня межа на одному екземплярі проксі-сервісу.
6. Від 5 до 25 мс – час відповіді геопошуку при 100 тис. активних анкет завдяки spatial-індексу R-Tree.
7. 0 downtime при rolling-update – стандартна поведінка GitOps.
8. 78 REST HTTP routes, bidirectional gRPC streaming та 9 типів подій Socket.IO як єдиний контракт системи.

Наукова новизна результатів полягає у поєднанні архітектурних рішень, що до цього часу не зустрічалися сукупно у вітчизняних роботах з програмної інженерії: оригінальна схема шардування real-time-чатів через комбінацію Socket.IO + gRPC bidirectional streaming з безконфліктною атомарною генерацією послідовного id повідомлень; система валідаційних rolling-хешів з чотирма виокремленими ключами; власна система кешування BuildKit-стану у MinIO з утилітою buildkit-cache-dance; власний хардненг базовий образ nginx з механізмом env-injection.

Практичне значення. Запропоновані архітектурні рішення безпосередньо застосовні у будь-яких соціальних та комунікаційних web-системах. Власний образ frontend-base-slim придатний для розгортання SPA-фронтендів у будь-яких бізнес-доменах. Власна Action ActionMinIOCache безпосередньо рекомендована для команд, що мають self-hosted MinIO та стикаються з обмеженнями GitHub Actions cache. Система валідаційних rolling-хешів – універсальне рішення для будь-яких систем, що видають клієнтам посилання на чутливі сутності.

Перспективи розвитку. Подальший розвиток системи передбачає (а) інтеграцію WebRTC для відеодзвінків між користувачами; (б) ML-засновану схему матчмейкінгу через embedding'и профілів у векторну БД (pgvector / qdrant); (в) розширення на інші месенджери (WhatsApp Business, iMessage Apps); (г) запровадження federation-моделі з поділом даних за регіонами для відповідності локальним вимогам конфіденційності.

Усі поставлені у вступі задачі виконано в повному обсязі. Мета роботи – спрощення та прискорення процесу пошуку партнерів для спілкування і знайомств шляхом розробки масштабованого, відмовостійкого web-додатку з геолокаційним пошуком, обміном повідомленнями у реальному часі та системою заходів, реалізованого засобами TypeScript і Go, – досягнута. Результати роботи апробовано у двох наукових публікаціях.

ПЕРЕЛІК ПОСИЛАНЬ

1. Newman S. Building Microservices: Designing Fine-Grained Systems. 2nd ed. Sebastopol : O'Reilly Media, 2021. 612 p.
2. Richardson C. Microservices Patterns: With Examples in Java. Shelter Island : Manning Publications, 2018. 520 p.
3. Donovan A. A. A., Kernighan B. W. The Go Programming Language. Boston : Addison-Wesley Professional, 2015. 380 p.
4. Cox-Buday K. Concurrency in Go: Tools and Techniques for Developers. Sebastopol : O'Reilly Media, 2017. 238 p.
5. Indrasiri K., Kuruppu D. gRPC: Up and Running: Building Cloud Native Applications with Go and Java for Docker and Kubernetes. Sebastopol : O'Reilly Media, 2020. 202 p.
6. Kubernetes: Up and Running: Dive into the Future of Infrastructure / B. Burns, J. Beda, K. Hightower, L. Evenson. 3rd ed. Sebastopol : O'Reilly Media, 2022. 328 p.
7. Domingus J., Arundel J. Cloud Native DevOps with Kubernetes. 2nd ed. Sebastopol : O'Reilly Media, 2022. 356 p.
8. The Site Reliability Workbook: Practical Ways to Implement SRE / edited by B. Beyer, N. R. Murphy, D. K. Rensin, K. Kawahara, S. Thorne. Sebastopol : O'Reilly Media, 2018. 512 p.
9. Evans E. Domain-Driven Design: Tackling Complexity in the Heart of Software. Boston : Addison-Wesley, 2003. 560 p.
10. Martin R. C. Clean Architecture: A Craftsman's Guide to Software Structure and Design. Boston : Prentice Hall, 2017. 432 p.
11. Vernon V. Implementing Domain-Driven Design. Boston : Addison-Wesley, 2013. 656 p.
12. Banks A., Porcello E. Learning React: Modern Patterns for Developing React Apps. 2nd ed. Sebastopol : O'Reilly Media, 2020. 310 p.
13. Hahn E. Mastering Next.js. Birmingham : Packt Publishing, 2023. 412 p.

14. Cherny B. Programming TypeScript: Making Your JavaScript Applications Scale. Sebastopol : O'Reilly Media, 2019. 324 p.
15. GORM Guides. GORM. URL: <https://gorm.io/docs/> (дата звернення: 20.04.2026).
16. Dragonfly Documentation. Dragonfly. URL: <https://www.dragonflydb.io/docs> (дата звернення: 20.04.2026).
17. ClickHouse Docs. ClickHouse. URL: <https://clickhouse.com/docs> (дата звернення: 20.04.2026).
18. MariaDB Knowledge Base. MariaDB. URL: <https://mariadb.com/kb/en/> (дата звернення: 20.04.2026).
19. MinIO Object Storage Documentation. MinIO. URL: <https://min.io/docs/minio/linux/index.html> (дата звернення: 20.04.2026).
20. Flux Documentation. Flux. URL: <https://fluxcd.io/flux/> (дата звернення: 20.04.2026).
21. Telegram Mini Apps. Telegram APIs. URL: <https://core.telegram.org/bots/webapps> (дата звернення: 20.04.2026).
22. Protocol Buffers Documentation. Protocol Buffers. URL: <https://protobuf.dev/> (дата звернення: 20.04.2026).
23. gRPC. gRPC. URL: <https://grpc.io/> (дата звернення: 20.04.2026).
24. CIS Docker Benchmark v1.6.0. Center for Internet Security, 2023. URL: <https://www.cisecurity.org/benchmark/docker> (дата звернення: 20.04.2026).
25. Souppaya M., Morello J., Scarfone K. Application Container Security Guide : NIST Special Publication 800-190. Gaithersburg : National Institute of Standards and Technology, 2017. 63 p. DOI: 10.6028/NIST.SP.800-190.
26. Container Security Verification Standard. OWASP Foundation / GitHub. URL: <https://github.com/OWASP/Container-Security-Verification-Standard> (дата звернення: 20.04.2026).
27. ІВАНОВ Д. А. WEB-ДОДАТОК ДЛЯ ЗНАЙОМСТВ І СОЦІАЛЬНОЇ ВЗАЄМОДІЇ З ВИКОРИСТАННЯМ GO, TYPESCRIPT ТА MARIADB. «ЗАСТОСУВАННЯ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ В

ІНФОРМАЦІЙНО- КОМУНІКАЦІЙНИХ ТЕХНОЛОГІЯХ». Київ, 2026. С. 95–97.

28. Іванов Д. А. Застосування gRPC та DragonflyDB у розподіленій мікросервісній системі для цифрових знайомств. Програмні системи та технології цифрового розвитку : матеріали Всеукр. наук.-практ. конф. Київ, 2026.

ДОДАТОК А. ДЕМОНСТРАЦІЙНІ МАТЕРІАЛИ



ДЕРЖАВНИЙ УНІВЕРСИТЕТ ІНФОРМАЦІЙНО-КОМУНІКАЦІЙНИХ ТЕХНОЛОГІЙ

НАВЧАЛЬНО-НАУКОВИЙ ІНСТИТУТ ІНФОРМАЦІЙНИХ ТЕХНОЛОГІЙ

Кафедра Технологій цифрового розвитку

КВАЛІФІКАЦІЙНА РОБОТА · БАКАЛАВР · 121 ІНЖЕНЕРІЯ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ

Web-додаток для знайомств за допомогою мови програмування TypeScript та Go

ВИКОНАВ

Іванов Дмитро
Андрійович

студент 4 курсу · група ТЦР-44

КЕРІВНИК РОБОТИ

Читулян Вадим
Олегович

КИЇВ · 2026

РОЗДІЛ 1 ПОСТАНОВКА ЗАДАЧІ

02 / 15

Мета, об'єкт та предмет дослідження

МЕТА РОБОТИ

Спростити та прискорити пошук
партнерів

Шляхом розробки **масштабованого
відмовостійкого web-додатку** з геолокаційним
пошуком, обміном повідомленнями у реальному
часі та системою party-events, реалізованого
засобами TypeScript і Go з використанням
мікросервісної архітектури.

ОБ'ЄКТ

Процес комунікації між
користувачами сервісів
знайомств

Геолокаційний пошук партнера, обмін
повідомленнями, формування групових
заходів, інформаційно-аналітична підтримка
рішень про вибір партнера.

ПРЕДМЕТ

Програмне забезпечення на базі
мікросервісної архітектури

З використанням **TypeScript & Go, gRPC** із
поточною передачею, сховищ **MariaDB ·
DragonflyDB · ClickHouse · MinIO** та
оркестрування у **Kubernetes**.

MATCH · TYPESCRIPT & GO · 2026

02 · МЕТА, ОБ'ЄКТ, ПРЕДМЕТ

Задачі кваліфікаційної роботи

1

АНАЛІЗ ГАЛУЗІ

Систематизувати функціональні та нефункціональні вимоги до dating-систем

2

АНАЛІЗ АНАЛОГІВ

Порівняння існуючих продуктів та обґрунтування актуальності

3

ОГЛЯД ТЕХНОЛОГІЙ

Обґрунтування стека для real-time системи

4

АРХІТЕКТУРА

Мікросервісна, межі доменів, gRPC контракти

5

МОДЕЛЬ ДАНИХ

Геопозиц, 1-to-1 чати, групові чати, валюти

6

API ТА UI

REST + gRPC + Socket.IO; Mini App-прототипи

7

РЕАЛІЗАЦІЯ

Go-сервіси, TS-клієнт, gRPC-проксі чату

8

CI/CD ТА GITOPS

MiniIO-кеш BuildKit, scratch-образи, FluxCD

9

ТЕСТУВАННЯ

41 лінтер, юніт-, інтеграційне, безпекові показники

Аналіз аналогів

Жоден з лідерів ринку (Tinder, Bumble, Hinge, Badoo) не пропонує одночасно Telegram Mini App, режим party-events та сучасну DevOps-практику. Саме цю прогалину закриває Match.

КРИТЕРІЙ	MATCH	TINDER	BUMBLE	HINGE	BADOO
Інтеграція як Telegram Mini App	+	-	-	-	-
Окремий режим party-events	+	-	-	-	-
Real-time chat (WebSocket)	+	+	+	-	+
gRPC у міжсервісній комунікації	+	ЧАСТКОВО	-	-	-
Групові чати у форматі заходу	+	-	-	-	ЧАСТКОВО
0-CVE контейнерні образи	+	н/в	н/в	н/в	-
GitOps-розгортання (FluxCD)	+	н/в	н/в	н/в	-
Веб-версія як рівноцінний канал	+	-	+	-	+

Вимоги до платформи

29 функціональних та 14 нефункціональних вимог. Пріоритети MSC: **M** — must, **S** — should, **C** — could.

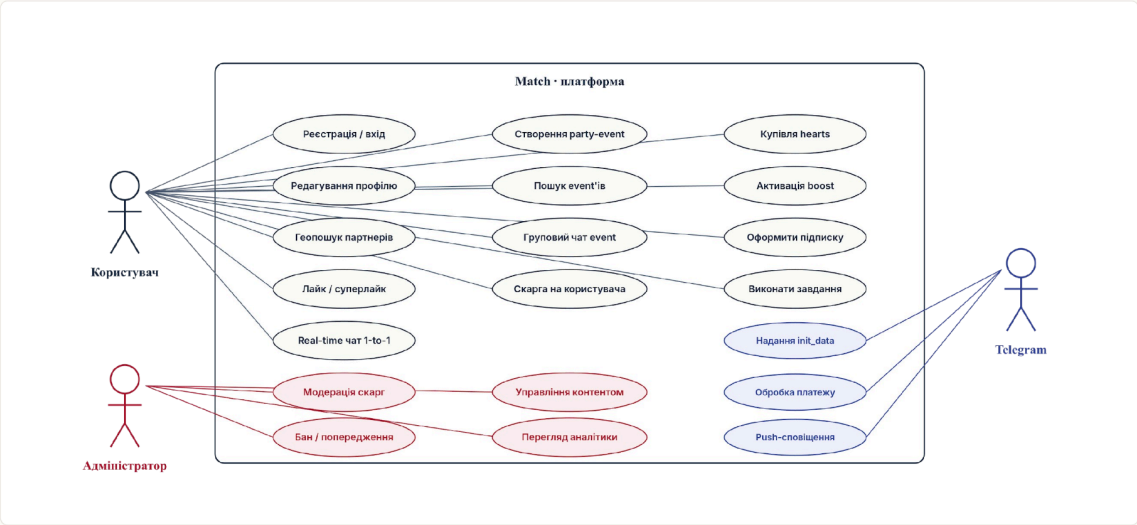
ФУНКЦІОНАЛЬНІ (ФРАГМЕНТ)			НЕФУНКЦІОНАЛЬНІ		
29 вимог · 8 доменів			14 вимог · 8 категорій		
Реєстрація через підпис init-data Telegram	M	≤ 2с	Performance: API p95 ≤ 100 мс; чат e2e ≤ 250 мс		p95
Геолокаційний пошук за двома радіусами	M	≤ 200мс	Scale: 100 000+ одночасних WebSocket / pod		100k+
Двосторонній лайк → автоматичний 1-to-1 чат	M		Fault-tolerance: rolling-update без простою		0 downtime
Real-time чат: текст, фото, відео, голос, реакції	M	≤ 250мс	Security: 0-CVE образи, TLS 1.3, rolling-хеші		0 CVE
Party-events: створення, пошук, груповий чат	M		GitOps deployment, env-injection, один артефакт		1 image
Hearts, boosts, 3 рівні підписки (Stars/Invoice)	S		Підтримуваність: 41 літер, Swagger, JSON-логи		≥ 50% cov
Завдання та реферальна система	S		Локалізація: ≥ 4 мов інтерфейсу з ConfigMap		4+ lang
Скарги, модерація, ban-система	M		a11y · WCAG 2.1 AA · ≥ 44×44 px hit targets		AA

Технології та обґрунтування вибору стеку

ШАР I	ШАР II	ШАР III	ШАР IV	ШАР V
Мови програмування	Серверні фреймворки	Клієнтський стек	Розподілені сховища	Інфраструктура
Go 1.26 · server	Gin v1.11 · HTTP	Next.js 16 · React 19	MariaDB + MaxScale	Kubernetes 1.31
TypeScript 5.x · strict	GORM v1.31 · ORM	Redux Toolkit · TanStack	DragonflyDB · cache	FluxCD · GitOps
SQL · MariaDB dialect	gRPC v1.79 · protobuf	Chakra UI v3	ClickHouse · OLAP	BuildKit + MinIO-kew
	Socket.IO · WSS	@telegram-apps/sdk v3.3	MinIO · S3 media	Trivy · CVE scan

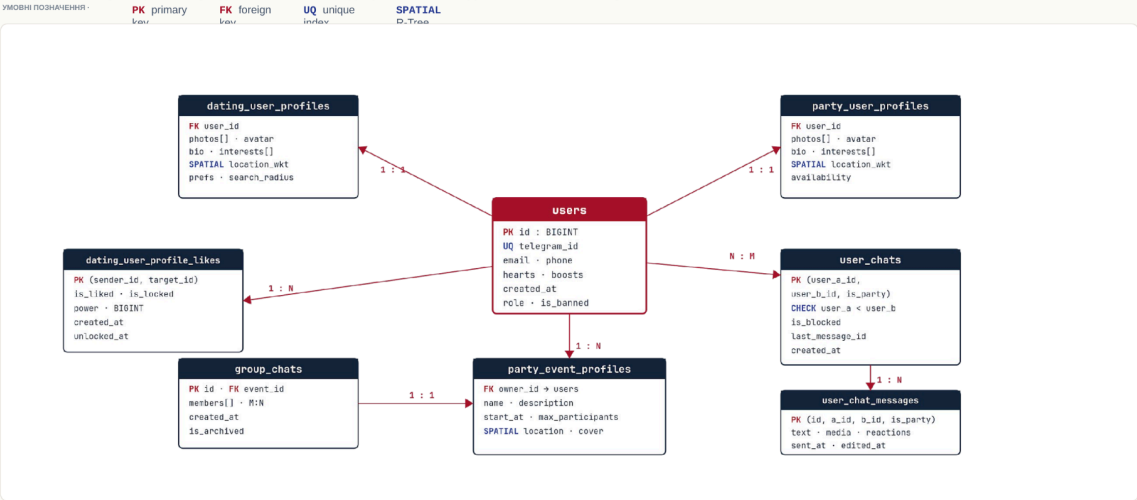
Go · goroutines 100k+ MS / pod	TypeScript + Next.js strict · protoc-gen-ts	gRPC замість REST/JSON ~4x менше трафіку
Шедулер M:N обслуговує 100 000+ одночасних WebSocket-з'єднань на одному pod; статичний бінарник ~10 МБ у scratch-образі.	Strict-типізація дає єдиний контракт з backend. Next.js 16 підтримує одночасно SSR для лендінгу та static export для Mini App.	Бінарний protobuf скорочує payload у 4 рази ; HTTP/2 multiplexing дозволяє bidi-streaming у тому ж з'єднанні зі строгою типізацією.
DragonflyDB замість Redis multithreaded	ClickHouse для аналітики OLAP · млрд рядків	Kubernetes + FluxCD GitOps · 0 downtime
Shared-nothing multithreading дає багаторазово вищу пропускну здатність на однакових ресурсах; сумісний з Redis-протоколом — drop-in заміна.	Колонкове сховище для потоку подій (OneTime, PerEvent, MoneyOps) — субсекундні OLAP-запити по мільярдах рядків без впливу на MariaDB.	HPA автоматично масштабує безліч подів за CPU/RAM без переписування коду; FluxCD виконує rolling-update без простою.

Діаграма варіантів використання платформи · фрагмент



Діаграма структури бази даних · фрагмент

Понад 50 таблиць покривають усі домени системи. На діаграмі показано фрагмент з основними сутностями та зв'язками.



Алгоритм пошуку

R-Tree spatial index на **POINT(SRID 4326)** зводить геопошук до 5–25 мс на 100 000 анкет замість O(N) повного перегляду.

root MBR

node A

node B

node C

node D

searcher · 50 km

Дерево MBR (Minimum Bounding Rectangles) — кожен внутрішній вузол містить мінімальний прямокутник, що покриває нащадків. Запит обходить тільки вузли, що перетинають коло пошуку.

SQL-запит та ранжування

```
-- 5-25 мс на 100k анкет SELECT dp.*, ST_Distance_Sphere(location_wkt, P) AS distance FROM dating_user_profiles dp WHERE ST_Distance_Sphere(location_wkt, P) <= 50000 AND dp.is_active = 1 AND dp.user_id <> :uid ORDER BY dp.boost_power DESC, distance ASC LIMIT 20;
```

01

Запит з координатами користувача в ST_Distance_Sphere

02

R-Tree фільтрує кандидатів за MBR за O(log N)

03

Ранжування: boost_power DESC → distance ASC

Система валідаційних rolling-хешів

Чотири незалежних валідатори за категоріями операцій (*privilege separation*). Кожен ключ — 32-байтова випадкова послідовність з Kubernetes Secret. Компрометація одного ключа не дає доступу до інших.

КОНСТРУКЦІЯ ТОКЕНА

```
time_bucket = [now_unix / window_size] token = HASH ( time_bucket || user_id || entity_id || ...metadata that affects validity || K ) → int64
```

```
validate: повторне обчислення для поточного time bucket та window_count попередніх. Окремий ts не передається – в хеші.
```

```
Захист від: enumeration (неможливо перебирати ID), replay (лише в межах rolling-вікон), CSRF (прив'язка до user_id).
```

K_media

Доступ до приватних медіа у заблокованому профілі

K_chat

Send / delete / react на повідомлення чату

K_like

Лайк профілю та розблокування заблокованих анкет

K_price

Платіжні URL · checkout-сесії Telegram Stars

NATCH · TYPESCRIPT & GO · 2026

10 · ROLLING-XEHI

Власний MinIO-кеш BuildKit прискорює збірку

01

git tag v*

push у репозиторій

02

MinIO-кеш

YaCodeDev/ActionMinIOCache + buildkit-cache-dance

03

multi-stage build

golang:1.26 -- scratch - CGO=0

04

Trivy scan

блок merge при CVE high/critical

05

Harbor push

приватний OCI-реєстр

06

FluxCD detect

image-reflector + image-automation

07

k8s rolling-update

0 downtime - -90c

15-25 ×

прискорення повторних збірок завдяки MinIO BuildKit-кешу

0

CVE high/critical у production контейнерних образах (Trivy)

~600 КБ

власний захищений базовий образ nginx з env-injection для SPA

92 %

cache hit rate · холодна 4-6 хв · тепла 12-20 с

Основні екрани застосунку

Telegram Mini App · одно-кроковий онбординг через init_data · optimistic UI · <2 с time-to-interactive.

01 · КОХАТИ

Геопошук та свайпи

02 · ПРОФІЛЬ

Деталі анкети · like / pass

03 · ТУСИТИ

Premium · приховані лайки

04 · ЧАТ

Real-time обмін · текст / медіа

05 · НАЛАШТУВАННЯ

Редагування анкети

Апробація

Результати дослідження апробовані у вигляді тез на двох всеукраїнських науково-технічних конференціях 2026 року.

01 VII ВСЕУКРАЇНСЬКА НАУКОВО-ТЕХНІЧНА КОНФЕРЕНЦІЯ · 2026

Мікросервісна архітектура платформи знайомств і подій з використанням gRPC та DragonflyDB
Іванов Д. // Матеріали конференції «Застосування програмного забезпечення в інформаційно-комунікаційних технологіях». Збірник тез. — 2026.

02 VII ВСЕУКРАЇНСЬКА НАУКОВО-ТЕХНІЧНА КОНФЕРЕНЦІЯ · 2026

Web-додаток для знайомств і соціальної взаємодії з використанням Go, TypeScript та MariaDB
Іванов Д. // Матеріали конференції «Технології цифрового розвитку: програмні системи та децентралізація». Збірник тез. — 2026.

Висновки

Усі поставлені у вступі задачі виконано в повному обсязі, усі нефункціональні вимоги досягнуто. Мету роботи реалізовано: розроблено та розгорнуто у Kubernetes повнофункціональний web-додаток **Match**.

✓ **Виконано аналіз галузі:** систематизовано 29 функціональних та 14 нефункціональних вимог, проведено порівняльний аналіз чотирьох провідних аналогів.

✓ **Обґрунтовано вибір стеку:** Go + TypeScript + gRPC + DragonflyDB + Kubernetes як оптимальне поєднання для масштабованої real-time системи.

✓ **Спроектовано архітектуру:** 13-сервісну мікросервісну систему з шардуванням real-time чатів через Socket.IO + gRPC bidirectional streaming.

✓ **Розроблено модель БД:** понад 50 таблиць, геопросторові R-Tree індекси та композитні ключі для нормалізації пар.

✓ **Реалізовано CI/CD:** власний MinIO-кеш BuildKit прискорює повторні збірки у 15–25 разів; контейнери з 0 CVE розгорнуті через FluxCD.

✓ **Досягнуто цільових показників:** API p95 62 мс, чат e2e 120 мс, 100k+ одночасних WebSocket / pod — всі NFR виконані.

**Дякую
за
увагу!**